the magazine of the accu

www.accu.org

# {cvu}

Volume 26 • Issue 3 • July 2014 • £3

## Features



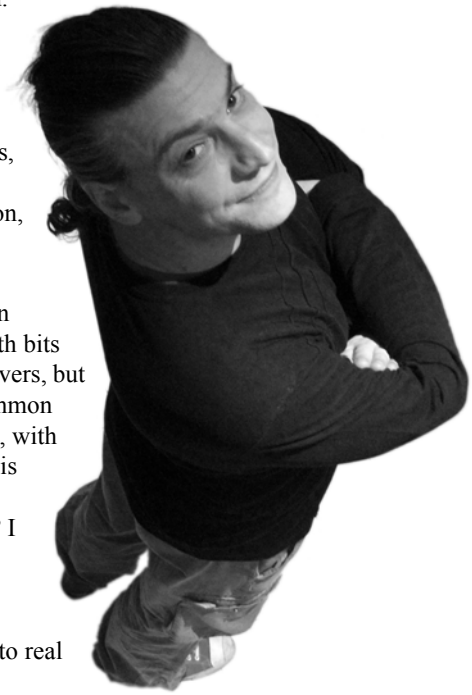
## Regulars

{cvu}
Volume 26 Issue 3
June 2014
ISSN 1354-3164
www.accu.org



accu

# Over and Under



I recently overheard someone criticising an in-house application for being massively over-engineered. It struck me that it's not often I hear this claim. Actually, as I thought about it, I realised I don't hear the term under-engineered either. It's certainly implied in some other disparaging descriptions: hacked, thrown-together, without tests, etc. All of these things imply software that's been created and released with little thought. By extension, then, *over*-engineered software has been...what? Created with too much thought?

I had a picture in my mind of some Heath Robinson contraption, a many-levered thing held together with bits of string. Of course, in software, there are indeed levers, but we call it 'configuration'. Moreover, it's not uncommon for the configuration of a system to get out of hand, with 'levers' and 'switches' for every part. Perhaps this is what over-engineering is. It got me wondering whether the term itself is specific to programming? I don't think it's fair to call even Heath Robinson's imaginary machines 'over engineered', because despite being obviously ridiculous and probably difficult to use, they were often practical solutions to real problems.

I have heard the term used before, of course, and sometimes to describe systems that merely had some thought put into their design. I doubt I'm the only programmer who has had the 'over-engineered' criticism levelled because their software had unit tests!

The system I was describing at the start did indeed have unit tests, and even a discernible 'architecture'. However, it was seemingly infinitely configurable, with an endless stream of parameters required for it to even operate. Furthermore, it took the requirement to be a distributed system as a challenge, and every calculation – however trivial – was sent over the network in order to, ahem, make best use of parallelisation. Just these two flaws indicate that *too little* thought was put into the design, with the result that it was hard to set up, and made less than optimal use of resources. Not, in fact, over-engineered at all.

STEVE LOVE
**FEATURES EDITOR**

# The official magazine of ACCU

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

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

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

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

# DIALOGUE



# REGULARS



# FEATURES



# SUBMISSION DATES

**C Vu 26.4:** 1st August 2014
**C Vu 26.5:** 1st October 2014

**Overload 123:** 1st September 2014
**Overload 124:** 1st November 2014

# WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

# ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

# COPYRIGHTS AND TRADE MARKS

# Feature Tracking

## Chris Oldwood considers different audiences for tools to track features.

A common question that comes up on the accu-general mailing list is about the tools teams can use to track the list of features being implemented for their product. Depending on the size of the team, the maturity of the product and the enterprise-ness of the organisation I've used a number a tools in the past. Other factors that have affected my opinion of the usefulness of a tool have often revolved around how it gels with the other tools such as the version control system.

For my own personal set of libraries and applications I naturally have very simple needs. Generally speaking all I want is a one line description of a bug or feature. Consequently every library and application has a simple text file called `TODO.txt` that is (often) an unordered list of features. One of the good things about keeping this file in the repo alongside the source code is that it changes as each feature is implemented. Generating the release notes is then as simple as picking out all the diffs of that file since the last release label.

More recently I've started using an online service (GitHub) to host my source code repos. Being a service aimed at project hosting, each repo automatically gains an Issues database and Wiki. The database of issues provides the next step up from a simple text file as you can track the status of features in more detail, such as who's assigned to it, which release it's destined for, etc. Because it's integrated with the source code repo including a number such as "#42" in the commit message creates a link in the GitHub UI to the issue with the same number which is good for software archaeology later.

At the other end of the spectrum are the Enterprise scale products that try to cater for big teams with heavy development processes that feel the need to try and track lots of 'important' metrics about each feature. They often come as part of a big suite that plays on their integration as a selling point. For example, Rational's ClearCase version control product supports triggers (that are incredibly annoying) ensuring a change can't be committed unless it's linked to an item in ClearQuest, their feature tracking database.

It's a noble pursuit to try and ensure that every code change, and therefore every commit, is related in some way to a formally tracked feature, whether instigated by the business or the development team itself, as that helps keep everyone honest. However, I find enforcing that kind of policy increases the pain of checking in out-of-band changes, such as to the build process, and only succeeds in enticing developers to bundle unrelated changes instead. If the tool makes adding a new feature request tedious then it's not going to be used the way you might prefer.

Outside the realm of dedicated 3rd party feature tracking products are the home grown alternatives. Naturally Excel, the Swiss-army knife of the corporate world, has been pressed into service on many an occasion. Despite it being painful to enter or read blocks of text with more than a couple of words, its ability to easily produce charts seems to be a big consolation prize in some circles. The agile project manager's fascination with burn-down charts and the like might fuel this choice.

What is probably the most derided part of the Microsoft Office suite – the Access database tool – is far more suited to the task, or at least was a decade or so ago. The only time I've ever actually created a database in Access was to knock up a simple bug tracking database for a very small team I was in. I'd seen someone do it before at another site I'd worked at and when the team grew large enough for the single-user problems to start becoming a drag it was easy to migrate the underlying database engine to something more heavy-duty, like SQL Server. With the plethora of free options available today the days of writing your own issue tracker have almost certainly long gone.

Recently, I've had the pleasure of using another of the Enterprise favourites – JIRA. I appear to have been quite lucky in that my team were using it how they wanted to, rather than how the project manager might want to. This meant we kept the ceremony fairly low by only tracking features at a high-ish level, linking them where appropriate and introducing sub-tasks when the larger features spanned multiple releases. JIRA can parse the check-in comments to relate source changes to a feature and supports links and attachments which are useful for relating associated documentation, such as design notes, email discussions, manual test cases, etc.

At the TICOSA conference [1] I mentioned in the last edition of this column [2] there was an interesting question about where to store the supporting documentation for a feature. Whilst the code might be a good place to document some specific details about the current implementation, it isn't necessarily the best place to 'Record Your Rationale' [3]. For this a wiki might be more appropriate or, if the document requires some richer content such as UML sketches, then storing them in the repo itself in a separate docs folder also keeps them close to hand.

The last 12 months has seen me working on a number of greenfield projects. Our very small team only needed a simple tool that would allow easy manipulation of stories rather than containing any notably lengthy text. In each case we used Trello and found it to be more than adequate. In fact after being forced briefly to use an in-house SharePoint-based imitation we soon realised how much we missed it! The text in cards can be written using Markdown for a little extra spice, can contain attachments, and checklists which are an excellent way to tick off the 'done, done' criteria. Cards also have a unique number which can be referenced in the commit messages. On one project board we had a column named after each release version and using the JSON export feature of Trello and JQ (a JSON command-line query tool) I produced some release notes automatically based on the card number and title.

Whilst a project manager's perspective of a feature tracking tool will likely be biased towards prioritisation and progress, as a developer I find I have different perspectives depending on the state of the feature. Prior to implementation I also want to know what the most important feature is along with details of acceptance criteria, etc. Post implementation I'm often looking for a way to trace a change in the source code back out to the driver for it so that I can better understand its context. This helps makes subsequent changes a little safer, especially when tests are scarce such as in legacy codebases. Sometimes, such as when a serious piece of technical debt has been accrued, I raise a ticket and embed the ticket number directly in the code to make it easier for a future maintenance programmer to track down the supporting notes in case I'm not around when they finally decide to pay it back. ■

## References

[1] http://www.ticosa.org

[2] *C Vu* 26-1, March 2014

[3] http://architect.97things.oreilly.com/wiki/index.php/Record_your_rationale

**CHRIS OLDWOOD**

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

# Nothing is Set in Stone

## Pete Goodliffe embraces change.

*They always say time changes things,*
*but you actually have to change them yourself.*
~ Andy Warhol

There is a strange fiction prevalent in programming circles: once you've written some code then it is sacred. It should not be changed. Ever.

That goes double for anyone else's code. Don't you dare touch it.

Somewhere along the development line, perhaps at the first check-in, or perhaps just after a product release, the code gets embalmed. It changes league. It is promoted. No longer riff-raff, it becomes digital royalty. The once-questionable design is suddenly considered beyond reproach and becomes unchangeable. The internal code structure is no longer to be messed with. All of the interfaces to the outside world are sacred and can never be revised.

Why *do* programmers think like this? Fear. Fear of getting it wrong. Fear of breaking things. Fear of extra work. Fear of the cost of change.

There is a very real anxiety that comes from changing code you don't know fully. If you don't understand the logic from the inside-out, if you're not entirely sure what you're doing, if you don't understand every possible consequence of a change, then you *could* break the program in strange ways or alter odd corner-case behaviour and introduce very subtle bugs into the product. You don't want to do that, do you?

Software is supposed to be soft, not hard. Yet fear leads us to freeze our code solid in an attempt to avoid breaking it. This is software rigor mortis.

> Do not embalm your code. If you have 'unchangeable' code in your product then your product will rot.

We see rigor mortis set when the original authors leave a project, and no one left fully understands their old business-critical code. When it's hard to work with legacy code, or to even make a reliable estimate for working with it, programmers avoid the code's core. It becomes an untamed code wilderness, where wild digital beasts roam unfettered. To work in a timely and predictable way, new functionality is added as new satellite modules around the edge.

We see rigor mortis set when a product is rolled onto production servers, and is used by many clients daily. The original system APIs stick because it will cost too much to change them; so many other teams or services now depend on them.

Code should *never* stay still. No code is sacred. No code is ever perfect. How could it be? The world is constantly changing around it. Requirements are always in a state of flux, no matter how diligently they were captured. Product version 2.4 is so radically different from version 1.6 that it's entirely possible the internal code structure *should* be totally different. And we're always finding new bugs in our old code that need to be fixed.

When your code becomes a straight jacket then you are fighting with the software, *not* developing it. You will be permanently dancing around necrotic logic and plotting ever more arcane courses around dodgy design.

**PETE GOODLIFFE**

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net or @petegoodliffe

> You are the master of your software; it's under your control. Do not let the code, or the processes around it, dictate how the code grows.

### Fearless change

*Be the change that you wish to see in the world.*
~Mahatma Gandhi

Of course, it is perfectly sensible to fear breaking software. Large software projects contain myriad subtleties and complexities that must be mastered. We don't want to introduce bugs through reckless modification. Only a fool would glibly make changes without actually knowing what they're doing. That's cowboy coding.

So how do we reconcile courageous modification with fear of error?

- Learn how to make good changes – there are practices that increase the safety of your work and reduce the chance of error. Courage comes from a confidence that your modification is safe.
- Learn what makes software easy to change, and strive to craft software with these attributes.
- Make daily improvements to your code that make it more malleable. Refuse to compromise code quality.
- Embrace healthy attitudes that lead to flourishing code.

Nothing is set in stone. Not the design. Not the team. Not the process. Not the code. Understand this, and the part you can play in improving your software.

> To modify code you need courage and skill. Not recklessness.

### Change your attitude

To 'enable' healthy change in your code the programming team have to adopt the right attitudes. They must be committed to code quality, and actually *want* to write good code.

Fearful, cowardly coding approaches don't make the grade. We shun: I didn't write this. It looks rubbish. I want nothing to do with it. I will venture into this code as little as possible. This attitude makes the coder's life a little easier now, but leads to design rot. Old code becomes stagnant whilst new driftwood washes up around its edges.

> 'Good code' is not somebody else's problem. It is your responsibility. You have the power to make a change, and to bring about an improvement.

Here are important attitudes, both for the team and for the individual, that contribute to healthy code growth:

- Fixing wrong, dangerous, bad, duplicated, or dis-tasteful code is not a distraction, a side-track, or a waste of precious time. It is positively encouraged. In fact, it is expected. You don't want to leave weak spots festering for too long. If you find code that is too scary to change, then it *must* be changed!
- Refactoring is encouraged. If you have a job that requires a fundamental code change to be done properly then do it properly: refactor. The team understands that this is required, and that some jobs may take a little longer when we find such problems.

- No one 'owns'' any area of the code. Anyone is allowed to make changes in any section. Avoid code parochialism; it stifles the rate of change.
- It is not a crime to make a mistake, or to write the wrong code (accidentally, at least!). If someone fixes or improves your code then it is not a sign that you are weak, or that the other programmer is better than you. You'll probably tinker with their work tomorrow. That's just the way it works. Learn and grow.
- No one's opinion should considered more important than anyone else's. Everyone has a valid contribution to make in any part of the codebase. Sure, some people have more experience in certain areas. But they are not code 'owners' or gatekeepers of the sacred code. Treating some people's work as 'more accurate' or 'better' than others' puts them on a false pedestal and demeans the contribution of the rest of the team.
- Good programmers *expect* change, because that is what software development is all about. You need nerves of steel and to not mind the ground changing underneath you. The code changes quickly; get used to it.
- We lean on the safety net of accountability. Again, we see reviews, pair programming and testing (both automated unit tests, and great QA/developer interactions) being key parts of ensuring our code remains supple. If you do the wrong thing, or introduce rigidity, it will be spotted before it becomes a problem.

## Make the change

An apocryphal story states that a tourist, lost in a country village, stopped a local and asked for directions to a town in a distant borough. The villager spent a moment in careful thought, and then answered slowly: if I were going there, I wouldn't start from here!

It sounds silly, but often the best place to start your journey from is not where you are, in a code quagmire. If you try to move forward you may sink. It may instead be best to work your way back to a sound point, leading the code on a route to a local highway, and once there press onto your destination at greater speed.

Obviously, it's important to learn how to navigate a route into code; how to map it, trace it, and understand where it hides surprising side-effects.

### Design for change

We strive for code that encourages change. This is code that reveals it's shape and intent, and encourages modification through simplicity, clarity and consistency. We avoid code with side-effects because it is brittle in the face of change. If you encounter a function that does two things, separate it into two parts. Make the implicit explicit. We avoid rigid coupling, and unnecessary complexity.

When an ugly, rigid codebase resists change then we need a battle strategy: we slowly improve the code day-by-day making safe, piecemeal improvements; we make changes to lines of code, and to the overall structure. Over a period of time, we watch it gradually slide into a malleable shape.

> Often it is best to make a series of frequent, small, verifiable adjustments, rather than one large sweeping code change.

Don't try to fight with the entire codebase at once. That may be a daunting, and perhaps intractable, task. Instead identify a bounded section of the code that you need to interact with, and focus on changing that.

### Tools for change

Pay attention now! This is really important: good tooling can help you make safe changes supremely fast.

A good automated test suite allows you to work fast and work well. It enables you to make modifications and get rapid, reliable, feedback on whether your modifications have broken anything. Consider introducing some kind of verifiable test for the sections of code you pick up, to avoid errors. Just as code benefits from accountability and a careful review processes, so do these tests.

> Automated tests are an invaluable safety harness that build confidence in your code changes.

The backbone of your development should be *continuous integration*: a server that continually checks out and builds the latest version of the code. If – heaven forbid – anything bad slips through to break the build, you will find out about it quickly. The automated tests should be run on the build server, too.

### Pick your battles

Nothing is set in stone, but not everything should be fluid.

Naturally, we pick our battles. We can't possibly change all of the code all of the time, whilst simultaneously adding more new work. We will always find unpleasant code that we can't fix right now, no matter how much we'd like to. The job may be too large. Or it may be past the scope of a mammoth refactor.

There is a certain amount of *technical debt* that we live with until we get a chance to make later improvement. This should be factored back into the project plan. Significant debt becomes work items that are placed onto the development roadmap, rather than forgotten and left to fester.

## Plus ça change

It sounds like a nightmare. Who could possibly work with code that is constantly changing? It's hard enough to track many simultaneous many changes, let alone join in with them!

# Dr Bjarne Stroustrup: An Interview

## Emyr Williams begins a new series of interviews in the programming world.

In 2013, I heard Pete Goodliffe talk about becoming a better programmer, and he lined up panel of experts about how to become a better programmer. Having heard the talk, I endeavoured to put as much of it as I could in to practice. During one of the intervals, I had a chance meeting with Bjarne Stroustrup, who was gracious enough to agree to be interviewed for my blog. I was also encouraged to publish it in the ACCU magazine, so here we are.

If you're a C++ programmer, then Bjarne Stroustrup won't need any introduction at all. However, if you are new to C++, then Dr Stroustrup is the designer and the original implementer of C++. He is currently a Managing Director in the technology division of Morgan Stanley in New York, a Visiting Professor in Computer Science at Colombia University, and a Distinguished Research Professor in Computer Science at Texas A&M University. His best known published work is *The C++ Programming Language* which is currently in its 4th edition.

How did you get started in computer programming? Was it a sudden interest in computing? Or was it a gradual process?

> During my last year of high-school, I had to decide whether to go to university and if so what to study. I decided to study mathematics because I was pretty good at that in high school, but I wanted a practical form of mathematics, some kind of applied math. That way, I would be able to make a living doing math after graduation, rather than becoming a teacher. So, I enrolled in 'Mathematics with Computer Science' in my home-town university (Aarhus University) because I mistakenly thought that 'Computer Science' was a form of applied math. It was good that I was wrong about that because I wasn't as good at math as I thought at the time (though being a poor mathematician is better than not being a mathematician) and I absolutely loved Computer Science – and especially programming – when I eventually was introduced to it in my second year at university.

What was the first program you ever wrote? And what language did you write it in?

> In my first Computer Science course, we learned several languages and wrote tiny programs in those. I don't remember those exercises, though. The primary language taught was Algol60. The first program I do remember was a small graphics program written in Algo60. It was probably my first program that was not a set exercise. It connected points on a super-ellipse to draw pretty pictures. The 'user interface' allowed me to specify the parameters for the super-ellipse, the number of points, and the number of lines. From a programming point of view, it combined math with graphics.

What would you say is the best piece of advice you've ever been given as a programmer?

> Just one piece of advice? "Test early and often." But maybe that's just my own variant of the old Chicago advice about elections. ["Vote early and vote often: http://en.wikipedia.org/wiki/Vote_early_and_vote_often]
>
> Try not to be too clever: Bugs hide in complex code. Be clear and explicit about what you are trying to build, and how. By 'explicit', I mean 'write it down in good clear English that others can read'. Articulating a design is important and helps you when it comes to construct test cases and write assertions. Always think about how a piece of code should be used: good interfaces are the essence of good code. You can hide all kinds of clever and dirty code behind a good interface if you really need such code.

If you were to start your career again now, what would you do differently? Or if you could go back in time and meet yourself when you were starting out as a programmer, what would you tell yourself to focus on?

> I think I would have taken a year off to travel the world and improve my interpersonal skills. Had I known that I'd spend most of my career writing in English and giving talks in English, I might have paid more attention in my foreign language classes. On the other hand, I have found topics with no apparent practical use (such as literature, history, and even philosophy) at least as useful as many specific technical skills. It is good not to have too narrow a focus.

**EMYR WILLIAMS**

Emyr Williams is a C++ developer who is on a mission to become a better programmer. His blog can be found at www.becomingbetter.co.uk

## Nothing is Set in Stone (continued)

However, we must embrace the fact that code changes: any code that stands still is a liability. No code is beyond modification. Treating a section of code as avoidably scary is counterproductive. ■

### Questions

1. What particular attributes makes software easy to change? Do you naturally write software like this?
2. How can we balance 'no code ownership' with the fact that some people have more experience than others? How does this affect the allocation of tasks to programmers?
3. Every project has code that changes frequently, and code that changes little. The latter code may be staid because it's not used, because it is healthily designed for extension by external modules, or because people actively avoid the nastiness within. How much of each of these kinds of rigid code do you have?
4. Does your project tooling support your code changes? How can you improve it?


**Becoming a Better Programmer: The Book**

Pete's new book – *Becoming a Better Programmer* – is published by O'Reilly. It's available from http://shop.oreilly.com/product/0636920033929.do

> Bertrand Russell: "The time you enjoy wasting is not wasted time."

**What was the biggest "ah ha" moment or surprise you've experienced when chasing down a bug?**

I don't think this question applies. I dislike debugging and my usual reaction to finally finding a bug is "How could I be daft enough to write that!" Alternatively, "What was he/she thinking?" Often, a simple invariant would have caught the problem early or a slight restraint on 'cleverness' would have avoided the problem altogether. What I enjoy is to design and to express my designs in code. Sometimes, the realization of a design can be amazingly beautiful. 'Getting it' with the STL (Alex Stepanov's handiwork) was an "Aha!" moment. Discovering how to express the STL better with concepts comes close.

**A lot is said about elegant code these days. What is the most elegant code you've seen? And how do you define what elegant code is?**

I'd say that one of the best answers I've seen for what makes elegant code, is something I've read from ACCU's own Roger Orr:

```
}
```

Just that closing brace. Here is where all the 'magic' happens in C++. Variables get destroyed, memory gets released, locks get freed, files get closed, names from outside the closed scope regain their meaning, etc. This is where C++ most significantly differs from other languages. It is interesting to see how destructors – an invention (together with constructors) from the first week or so of C++ – have increased in importance over the years. So many of the modern and most effective C++ techniques critically depend on them

**With the advent of C++ 14 upon us, where do you see C++ going in the future? Is there anything you'd like to see, or something you'd wish you'd done differently?**

For the future, I'd like to see better concurrency support, concepts (requirements for template arguments), and increased simplicity. I'd like to explore the idea of simplicity within C++ with features such as range-for loops, auto, and libraries that make simple things simple. "Within C++, there is a much smaller and cleaner language struggling to get out" (and no, that language is not C, D, Java, C#, or whatever). I'd like to explore what such a "much smaller and cleaner language" might look like in general and how it could be embedded into C++.

The essence of C++ is that it provides a direct map to hardware and offers mechanisms for very general zero-overhead abstraction. A future C++ should be better at both. This precludes simple imitation of many modern ideas and trends. We can learn a lot from other languages (and always have done so), but direct import of language features is non-trivial.

The 'time machine question' is easier to answer because it has no effect on reality. We can't change the past and even if we could, I'm pretty sure I'm no smarter than 1980s-vintage Bjarne and he had a much better feel for what was possible at the time. Even the best time machine would not allow me to compile C++14 on a 1MB, 1MHz, 1985 computer. If I could have dropped the "Concepts Lite" design on Bjarne's desk in 1987, we might have avoided a lot of problems. At that time, I was looking at ways of constraining template arguments, so I would have recognized the importance of the ideas. Furthermore, the complexity and compile-time overheads are minimal so I could have implemented "Concepts Lite" well using 1980s technology. Of course, working from first principles, I would not have chosen the C declarator syntax or two-way conversions between fundamental types, but to fix those, you would need to take the time machine a few years further back for a chat with Dennis.

**Try not to be too clever: Bugs hide in complex code. Be clear and explicit about what you are trying to build, and how**

With technology moving so fast these days, where do you think the next big shift in computer programming is going to be?

Hard to say; there are so many different kinds of programming. I'm not even sure what the last big shift was. Dynamic languages? Object-Oriented Programming? Functional Programming? XML? Virtualization? C? Generic Programming? I'm pretty sure I could point to areas where each of those answers would be quite reasonable – as well as areas where each would be ludicrous. The field of software development is just too huge and diverse for simple generalizations.

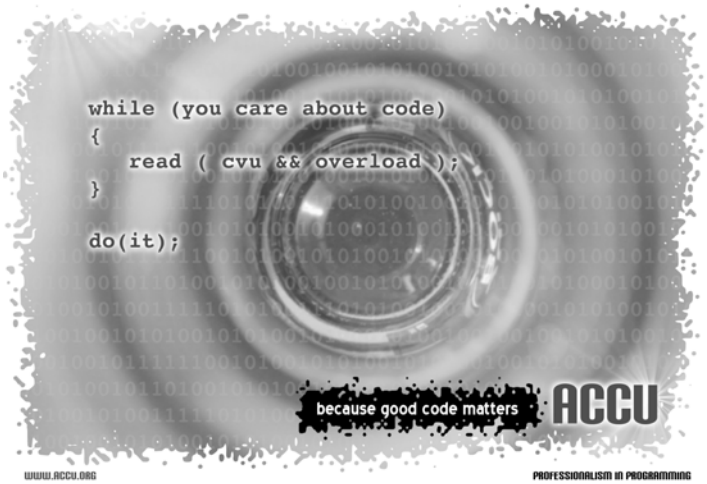
In the parts of the C++ world that I know best, my guess is that the improved support for concurrency in C++17 (various higher-level models of concurrency beyond the basic threads-and-locks level) will cause major changes and that concepts (starting with 'Concepts Lite') will complete generic programming's move into the mainstream. That combination, coming on top of the improvements provided in C++11, should completely change the way C++ is used.

I say "should" rather than "will" because I fear that many will hold back out of fear of novelty, for lack of intellectual flexibility, or because of constraints from old code bases. People are more likely to see the risks and complications of a change than to appreciate the risks and costs incurred by using outdated tools and techniques. Maybe, I'm being a bit less optimistic than I should be: people who have used C++11 tend to complain bitterly when they have to go back to C++98. Significant progress has been made and we can now write simpler and better code than we used to. Many people know that and they are not going to accept "the old ways" forever.

Finally, do you have any advice for any kids or adults who are looking to start out as a programmer?

Don't just program. Know what problems you want to solve using programming. Don't rush into programming. Work on your communication skills. And don't forget to have fun – you are much better at things you enjoy doing than things you consider tedious. Learn to see the beauty in elegant and efficient code! ■

A Tour of C++, for people who want to quickly know what C++11 is, and his textbook for beginners: *Programming: Principles and Practice using C++*, which now uses C++11 and some bits of C++14 is available now from your usual book reseller.



# Checking Websites for Specific Changes

Silas S. Brown tries to improve developer productivity in a small way.

Some years ago I used a Perl program called 'Web Secretary' [1] to alert me to changes on particular web pages. This is a command-line tool that can be run in a daily 'cron' job or whatever, checking the pages you specify for changes and outputting its results usually to your email inbox. But sometimes it told me a page had changed and then I found the change wasn't very interesting, and I wanted some way of reducing this 'noise'.

So I wrote a Python script [2] which checks for changes to specific phrases on a page, ignoring everything else. It can alert you when a certain phrase or regular expression is absent (i.e. has been changed or removed), or when it is present, either in the rendered text of the page or in its source code. It's not always possible to point to a particular phrase that would make a change 'interesting', but in a surprising proportion of cases, it is. For example, if you want to know when the next version of some library is released but you can't keep up with all other information on that site, you could simply monitor its version number.

My script tries to be 'nice' to servers, limiting the rate of requests to each site, re-using connections when possible, sending If-Modified-Since headers so the server doesn't have to re-send a page if it hasn't changed at all, and of course re-using a fetched copy if there are two or more things to check on the same page. On the other hand, when dealing with multiple sites it does send requests in parallel. For each thing you want to check, you can also set the frequency at which you want to check it, and the page will be queried at no more than that frequency even if the script is run more often (useful if some pages change more than others; we don't want to drain webmasters' bandwidths unnecessarily).

Besides the obvious checks that your own sites (and/or redirects) are still functioning, you might want to selectively monitor changes for:

1. Staying up to date with software you care about. If the download page says the latest version is 5.4.3, simply monitor this phrase and you'll be alerted only when it changes. (If they might add digits to the existing number, use a regular expression. Of course if you're on Linux then you could just rely on the package manager to update things, but not everything is a package and you might sometimes want to be ahead of the packagers for a particular piece of software.)
2. Monitoring a wiki when you are interested in a small section of a page but don't really mind what happens to the rest of that page. I sometimes do this to see what happens when I made a small addition (not that I'd be a troll for putting it back if someone took it out for good reason, but I might still want to know why that happened); there are advantages to MediaWiki's 'watchlist' telling you about *all* changes to a page, but I don't have the mental bandwidth to fully 'adopt' every page I ever contribute to, so it's useful to have the option of at least monitoring a phrase or two that you edited even if you don't track what happens to the rest of the page. Of course it's always possible that someone will edit what you wrote in a way that negates the meaning but still includes your original words and therefore fails to trigger your alert, so you'd need to use good judgement about when it is and isn't reasonable to rely on this kind of monitoring. You can also use it to see if something you deleted gets reinstated, and this shouldn't give any false negatives.
3. Checking for replies to any comments you make on other people's blogs, when those blogs don't have a decent function for notifying you. If the thread is old then you could just monitor the part that says "36 comments" and see if that changes (although it's just possible that a new one will be added and another one deleted at the same time, leaving the total number unchanged), or you might be able to find some other phrase on the page that can reasonably be checked for changes.
4. Checking for the release of a 'coming soon' item, or for (changes to) specific commercial offers without having to sign up to receive everything from that company (note to companies: please don't think such partial monitoring is hurting your customer contact, because the alternative might be not monitoring you at all). If you're 'job hunting' it can also be useful for doing regular-expression searches on vacancy lists.
5. Avoiding 'link rot'. If you maintain a website with external links, you can set up checks that the pages you link to still contain the material you were pointing at. (There are tools to check the pages are still live, but that won't detect situations where the page is still live but no longer contains what you were linking to due to a site re-structure or something, and if a site you link to gets taken over by criminals then it might harm your search rankings if you don't fix that link.)

Checks can be annotated with comments to remind you what to do when something changes, for example "check the link on the such-and-such page".

I find this monitoring script makes me more productive, because I'm less likely to go off on a tangent thinking "I'll just check if X has happened yet" when I know my scripts are checking for X automatically. It also lets me keep up-to-date with more slow-changing resources than I would if I had to rely on checking them manually or on subscribing to all of their news (I still do this for some things but can now be more selective). It still needs me to maintain a list of things to check, but at least it seems to be a step in the right direction. ■

## References

[1] Web Secretary http://baruch.ev-en.org/proj/websec/

[2] WebCheck http://people.ds.cam.ac.uk/ssb22/setup/webcheck.html

**SILAS S. BROWN**

Silas S. Brown is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk


If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

# Being Original

## Chris Oldwood reflects on the content of talks and articles.

A few years ago I decided to take some time off after completing another long contract and having reached 10 years of commuting into London. At that point I had only been a member of ACCU for a couple of years and I was becoming friends with a few of the more prominent members. Although said partly in jest there were some niggling suggestions that I should consider writing about something. Naturally I downplayed the situation by suggesting that I had nothing worth saying. After all the stuff I read about in *C Vu* and *Overload* was far cooler than anything I did.

However, during my break the ACCU conference occurred and I reluctantly agreed to write up a review. This felt like a safe option, as being a review it was entirely subjective and so I couldn't really say anything that might be considered 'wrong' per-se. Up to that point the only thing I'd ever written were some lengthy diatribes to my team mates about the technical debt in the codebase and how we should probably go about tackling it. Each email had taken ages to pen and involved constant refactoring to try and get the tone right. The thought of writing anything longer wasn't exactly appealing.

I subsequently decided that the safest route was to self-publish via a blog. This way nothing could be rejected by an editor and so anything I wrote was definitely going to see the light of day. Being my own publisher also meant that I had no deadlines. And so after cranking out a few really safe posts about what I intended to get out of writing, who the ACCU was, etc. I started to consider tackling something meatier…

It was then that I realised I really didn't have anything original to say. As I looked back at my career as an Enterprise programmer it occurred to me that I spent my entire career standing on the shoulder of giants. Enterprise development is all about playing it safe. My job involves using mature technologies to solve the same problems as other enterprises using patterns and techniques that have already been discovered and documented by others years, if not decades, ago. If none of what I do is novel is it a surprise that I'd have nothing truly interesting to write about? After all, if everyone reads the same books and blogs, follows the same people on Twitter, etc. then they'll already know everything I do; probably more. Who exactly would be listening?

Seven months later after playing it safe I finally started to consider sticking my neck on the line. By then I had thought about a technique I had used a few years back which I pondered might have some merit in it. I put together a draft, sent it off to a fellow ACCU member who had recently asked me to look over a couple of their articles and waited for the feedback. The feedback was really useful but it seemed I still had quite a bit of work to do to knock the article into shape.

That article sat unfinished in my 'pending' folder for over three years! In the meantime I continued to bash out more safe reviews of the conference, London branch meetings and a few book reviews for some of the less technical tomes. I was also cajoled into writing a piece for Desert Island Books and Inspirational Particles, both still firmly in the more subjective writing camp. I know others find the Code Critique is a good way to get started, but there always seemed to be so many comprehensive answers that once again I didn't believe I could add anything extra.

So, if you haven't got anything original to say, what's the answer – plagiarise? That's not really a viable solution. And then it slowly dawned on me as I started to read more and more of the old classic books and papers that much of what is said is not necessarily that new anyway, but what is new is the way it's presented. As is hopefully apparent from my article in the previous *C Vu* about getting to grips with list comprehensions [1] it took a number of different books, articles, talks and thinking to eventually understand the concept.

My first non-review style article finally appeared almost 4 years after I initially plucked up the courage to even begin writing. It was about accessing more than 4 GB from a 32-bit Windows process and was pretty niche given that 64-bit was becoming prevalent even in the slow moving enterprise world by the time it got published. But I had finally bitten the bullet and stuck my neck out.

I'm a very defensive writer and speaker. I do not have the courage to suggest what you should or should not do, instead I prefer to present what I have done, backed up where possible by what my influences are with the only expectation being that it may be of some value to someone else. But my fear is that someone will point out what is obvious to everyone else and negate my entire article or talk by describing how I should be doing it. It's not that I'm afraid to be wrong – on the contrary, I want to learn – but I'd feel upset about wasting not only my time, but the time of the reviewers and readers.

At the conference this year I spoke to a few people who appear to be under a similar illusion that they have nothing original to say. For a start they may work in a different industry to me and have different constraints and cultures which is already interesting. However, they find that despite the plethora of content out on the internet their colleagues are still unaware of many of the concepts and techniques that those more experienced take for granted.

Maybe the reason many programmers have not grasped something is not because they are stupid or haven't bothered reading about it, but is because nobody has spoken to them in a way that they understand. I've tried to digest the Wikipedia page on monads [2] a number of times but have found myself lost in a sea of Computer Science babble. There is much said about Monads which makes me think I probably need to understand them, but to date no one has spoken to me in a language that I can comprehend.

So if you labour under the belief that everything has already been said and that no regurgitation of existing ideas is useful then ask yourself this – why did Kevlin Henney begin his talk on Immutability at this year's ACCU conference with a modern take on a 20-year-old article written way back in issue 8 of Overload [3]?

They say "those who cannot remember the past are condemned to repeat it" [4]. Perhaps you can help out those of us unaware of some aspects of the past by ensuring we don't waste time rediscovering it and allowing us to stand on your shoulders as well in the future. ■

## References

[1] *C Vu* 26-2 (May 2014)
[2] http://en.wikipedia.org/wiki/Monad_(functional_programming)
[3] 'Circle & Ellipse – Vicious Circles', Kevlin Henney, *Overload* 8, 1995
[4] http://en.wikipedia.org/wiki/George_Santayana

**CHRIS OLDWOOD**

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

# How to Deconstruct Compile Time FizzBuzz in C++ Without Using Boost

Malcolm Noyes looks under the hood at some C++ template metaprogramming tools.

Several years ago, Adam Peterson published an article [1] explaining how to implement `FizzBuzz` at compile time in C++. The code was very neat, but had a dependency on Boost and didn't go into great detail on how it worked, so I thought I'd write it again from scratch and try to explain my workings. In this article I'm only going to show examples that would work for very small `FizzBuzz` sequences because, well, this is a learning exercise not a typing exercise!

At the end of this article you'll find the source that I used to run the examples – up to 16 items in the sequence. Much of what follows is a variation on the code in chapter 5 of *C++ Template Metaprogramming* [2]), in particular the use of `mpl::vector` and the 'tiny' sequence.

## First, get an error message

To make this work, we first have to persuade the compiler to print an error message to the console when we build the program, something like Listing 1.

Listing 1

```
struct void_;

template <class T0 = void_, class T1 = void_,
   class T2 = void_, class T3 = void_>
struct vector
{
  typedef vector type;
  typedef T0 T0;
  typedef T1 T1;
  typedef T2 T2;
};
typedef vector<int,long,double>
   ::compilation_error_here res;
```

This is a simplified version of the `mpl::vector` example from *C++ Template Metaprogramming*. In particular note the `typedef` for 'type' that indicates the type of the class itself and the `typedef`s for template parameters; we'll be using these in our helper templates. When we compile this with VS2010, we get this:

```
1>ClCompile: 1> main.cpp
1>c:\projects\fizzbuzz_example\main.cpp(243): error
C2039: 'compilation_error_here' : is not a member
of 'vector<T0,T1,T2>' 1> with 1> [ 1> T0=int, 1>
T1=long, 1> T2=double 1> ]
1>c:\projects\fizzbuzz_example\main.cpp(243): error
C2146: syntax error : missing ';' before identifier
'res' 1>c:\projects\fizzbuzz_example\main.cpp(243):
error C4430: missing type specifier - int assumed.
Note: C++ does not support default-int
1>c:\projects\fizzbuzz_example\main.cpp(243): error
C2065: 'res' : undeclared identifier
```

**MALCOLM NOYES**

Malcolm Noyes is a c++ programmer and author of the goospimpl mocking library.

This is promising; it displays the types that we put in, and in the right order. That's fine, but the compiler works with types and we need to display numeric values, so we need a helper to convert integers to types. Andrei Alexandrescu's **Loki** [3] calls this `Int2Type` and **Boost.MPL** [4] has `mpl::int_` that does the same thing, but it's really easy to implement:

```
template <int N>
struct int_
{
  static const int value = N;
};
```

For each distinct value `N`, this template creates a distinct type, so `int_<0>` is a different type from `int_<1>` etc.

Now we can put this into our 'error' template:

```
typedef vector<int_<0>,int_<1>,
   int_<2> >::compilation_error_here res;
```

and we get error output with increasing numeric values, which is what we want:

```
1>c:\projects\fizzbuzz_example\main.cpp(250): error
C2039: 'compilation_error_here' : is not a member
of 'vector<T0,T1,T2>' 1> with 1> [ 1> T0=int_<0>,
1> T1=int_<1>, 1> T2=int_<2> 1> ]
```

## Now generate the sequence...

That's fine, but now we need to generate the list of types so that `Fizz`, `Buzz` and `FizzBuzz` can be inserted at the right times – *C++ Template Metaprogramming* calls this list of types a sequence.

To get a new sequence, we have to append a new type (e.g. `int_<1>`) to an existing sequence (`int_<0>`), i.e. we start with a list of types containing `int_<0>`, append a new type `int_<1>` and end up with a new sequence containing `int_<1>`, `int_<0>`. If we were using **Loki** we could just append to its typelists but here we're modelling our sequence on `mpl::vector` so we need to do a bit more work.

Our aim is to `push_back` a new type to an existing sequence, e.g.:

```
...
template <class Sequence, class T>
struct push_back
  : push_back_impl<Sequence, T, ??? >
{};
```

It turns out that we can do this if we know the size of the exiting sequence, i.e. if we have an existing sequence with 0 elements then we can create a specialisation of `push_back_impl` to handle that case (we'll use a special `void_` type to indicate that the element is empty):

```
template <class Sequence, class T, int N>
struct push_back_impl;

template <class Sequence, class T>
struct push_back_impl<Sequence, T, 0>
  : vector<T, void_, void_, void_>
{};
```

This just 'adds' the new type to an empty sequence.

Similarly, if the sequence already contained 1 element then we could create a template that used the one existing type and 'added' the new type to that:

```
...
template <class Sequence, class T>
struct push_back_impl<Sequence, T, 1>
  : vector<typename Sequence::T0, T
  , void_, void_>
{};
```

Now the problem is to know how big our current sequence is, so we first need to specialise our template for every possible size of sequence (this is why we're only working with small sequences!), e.g. for zero size:

```
template <>
struct size_impl<void_,void_,void_,void_>
  : int_<0> {};
```

This **struct** will have a **static const value = 0** member that we can use later. For 1 element in the existing sequence:

```
...
template <class T0>
struct size_impl<T0,void_,void_,void_>
  : int_<1>   {};
```

Now we can apply this to get the size of the sequence:

```
...
template <class Sequence>
struct size : size_impl<typename Sequence::T0
  , typename Sequence::T1, typename Sequence::T2
  , typename Sequence::T3>
{};
```

If all the types of the sequence are empty (**void_**) then the specialisation **size_impl<void_,void_,void_,void_>** would get selected, which as we know has a value member equal to zero. If the first element had a type (e.g. **int<0>**) then the specialisation **size_impl< int<0), void_, void_, void_>** will get selected, with **value = 1**. This process continues for as many types as the sequence can support (four in this example).

This is a very much simplified version of the code in *C++ Template Metaprogramming*, in particular I've not implemented 'apply' metafunctions to make it clearer what's going on. Note that here the size is derived from the **int_<>** helper template that we defined earlier so that we can get the 'value' later (see Listing 2).

For an existing length, the 'size' template will select the 'best' specialisation of the **size_impl** template and add the new type as the last parameter. So now we can append a new type to another (Listing 3).

Listing 2

```
template <class T0, class T1, class T2, class T3>
struct size_impl : int_<4> {};

template <class T0, class T1, class T2>
struct size_impl<T0,T1,T2,void_> : int_<3> {};

template <class T0, class T1>
struct size_impl<T0,T1,void_,void_> : int_<2> {};

template <class T0>
struct size_impl<T0,void_,void_,void_> : int_<1>
{};

template <>
struct size_impl<void_,void_,void_,void_> :
int_<0> {};

template <class Sequence>
struct size : size_impl<typename Sequence::T0
  , typename Sequence::T1
  , typename Sequence::T2
  , typename Sequence::T3>
{};
```

Listing 3

```
template <class Sequence, class T, int N>
struct push_back_impl;

template <class Sequence, class T>
struct push_back_impl<Sequence, T, 0>
    : vector<T, void_, void_, void_>
{};

template <class Sequence, class T>
struct push_back_impl<Sequence, T, 1>
    : vector<typename Sequence::T0, T, void_,
void_>
{};

template <class Sequence, class T>
struct push_back_impl<Sequence, T, 2>
    : vector<typename Sequence::T0, typename
Sequence::T1, T, void_>
{};

template <class Sequence, class T>
struct push_back_impl<Sequence, T, 3>
    : vector<typename Sequence::T0, typename
Sequence::T1, typename Sequence::T2, T>
{};

template <class Sequence, class T>
struct push_back
    : push_back_impl<Sequence, T,
size<Sequence>::value >
{};
```

The same idea applies here; we use the **push_back** template to select the most appropriate **push_back_impl** template for the existing size. Note that we're not checking that the sequence is already full; this shouldn't be a problem with our limited use case for this example.

Now we can use the recursive parts of Adam's **RunFizzBuzz** to generate our sequence (Listing 4).

In this case the compiler will use the primary template and will recursively subtract 1 from the number and then instantiate itself until the number gets to zero, when it will use the specialisation for '0' to terminate the sequence, which gives us:

```
1>c:\projects\fizzbuzz_example\main.cpp(306): error
C2039: 'compilation_error_here' : is not a member
of 'vector<T0,T1,T2,T3>' 1> with 1> [ 1>
T0=int_<0>, 1> T1=int_<1>, 1> T2=int_<2>, 1>
T3=int_<3> 1> ]
```

Listing 4

```
template<int i>
struct RunFizzBuzz
{
  typedef int_<i> Number;
  typedef
    typename push_back<typename RunFizzBuzz<i-1>
    ::type, Number>::type type;
};

template<>
struct RunFizzBuzz<0>
{
  typedef vector<int_<0> > type;
};

int main()
{
  typedef RunFizzBuzz<3>::type
    ::compilation_error_here res;
}
```

Listing 5

```
template <class T1, class T2, bool c>
struct if_c_impl
{
  typedef T1 type;
};

template <class T1, class T2>
struct if_c_impl<T1, T2, false>
{
  typedef T2 type;
};

template<bool c, class T1, class T2>
struct if_c : if_c_impl<T1,T2,c>
{};
```

That looks a lot like what we want, but so far we haven't output **Fizz** or **Buzz**; we'll fix that now...

## Selecting Fizz

We need to select a different type (**Fizz**) if the number is divisible by 3. This calculation is a compile time constant for each template instantiation, so we can use it as a parameter to a template; **if_c** is a fairly simple type selection template that works by specializing for one value of the condition (in this case **false**) – see Listing 5.

Now we can add the condition for **Fizz** (Listing 6), which produces:

```
1>c:\projects\fizzbuzz_example\main.cpp(327): error
C2039: 'compilation_error_here' : is not a member
of 'vector<T0,T1,T2,T3>' 1> with 1> [ 1>
T0=int_<0>, 1> T1=int_<1>, 1> T2=int_<2>, 1>
T3=Fizz 1> ]
```

which is exactly what we want!

Listing 6

```
struct Fizz{};

template<int i>
struct RunFizzBuzz
{
    typedef int_<i> Number;
    typedef if_c<i % 3 == 0, Fizz, Number>
condition3;
    typedef typename push_back<typename
RunFizzBuzz<i - 1>::type,
                               typename
condition3::type>::type type;
};

template<>
struct RunFizzBuzz<0>
{
    typedef vector<int_<0> > type;
};
```

## Selecting Buzz

Adding the additional conditions for **Buzz** and **FizzBuzz** is trivial (as long as we remember to test for **FizzBuzz** in the right place!):

```
struct Fizz{}; struct Buzz{}; struct FizzBuzz{};
template<int i> struct RunFizzBuzz { typedef
int_<i> Number; typedef typename if_c<i % 3 == 0,
Fizz, Number>::type condition1; typedef typename
if_c<(i % 5 == 0), Buzz, condition1>::type
condition2; typedef typename if_c<(i % 3 == 0) &&
(i % 5 == 0), FizzBuzz, condition2>::type
condition3; typedef typename push_back<typename
RunFizzBuzz<i - 1>::type, condition3>::type type;
}; template<> struct RunFizzBuzz<0> { typedef
vector<int_<0> > type; };
1>c:\projects\fizzbuzz_example\main.cpp(305): error
C2039: 'compilation_error_here' : is not a member
of 'vector<T0,T1,T2,T3,T4,T5,T6,
T7,T8,T9,T10,T11,T12,T13,T14,T15>' 1> with 1> [ 1>
T0=int_<0>, 1> T1=int_<1>, 1> T2=int_<2>, 1>
T3=Fizz, 1> T4=int_<4>, 1> T5=Buzz, 1> T6=Fizz, 1>
T7=int_<7>, 1> T8=int_<8>, 1> T9=Fizz, 1> T10=Buzz,
1> T11=int_<11>, 1> T12=Fizz, 1> T13=int_<13>, 1>
T14=int_<14>, 1> T15=FizzBuzz 1> ]
```

Just for completeness, it works on gcc (MingW) too, although it is a little harder to see...:

```
$ gcc --version gcc.exe (GCC) 4.8.1 Copyright (C)
2013 Free Software Foundation, Inc. This is free
software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY
or FITNESS FOR A PARTICULAR PURPOSE. $ make g++ -c
-Wall main.cpp -o main.o main.cpp: In function 'int
main()': main.cpp:306:13: error:
'compilation_error_here' in 'RunFizzBuzz<15>::type
{aka struct vector<int_<0>, int_<1>, int_<2>, Fizz,
int_<4>, Buzz, Fizz, int_<7>, int_<8>, Fizz, Buzz,
int_<11>, Fizz, int_<13>, int_<14>, FizzBuzz>}'
does not name a type typedef
RunFizzBuzz<15>::type::compilation_error_here res;
^ make: *** [main.o] Error 1
```

You can find the code used in this article here [5]. ■

## References


[1] Adam Peterson's article on how to implement **FizzBuzz** at compile time in C++: http://www.adampetersen.se/articles/fizzbuzz.htm

[2] C++ Template Metaprogramming: http://www.amazon.co.uk/Template-Metaprogramming-Concepts-Techniques-Beyond/dp/0321227255/ref=sr_1_1?ie=UTF8&qid=1381403602&sr=8-1&keywords=c%2B%2B+template+metaprogramming

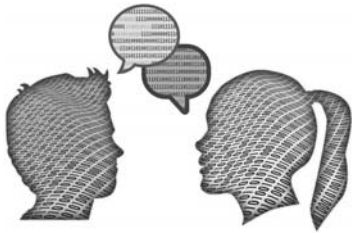
[3] Loki: http://loki-lib.sourceforge.net/

[4] Boost.MPL: http://www.boost.org/doc/libs/1_54_0/libs/mpl/doc/index.html

[5] http://www.graoil.co.uk/downloads/fizzbuzz_example.zip

# Code Critique Competition 88

Set and collated by Roger Orr. A book prize is awarded for the best entry.



Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

## Last issue's code

I was adding items into a map only if they weren't already there and the code reviewer said I shouldn't be using operator[] - see how I was doing it in the function called `old_way`. So I tried to do it better using the code in `new_way`. Now the code sometimes works and sometimes doesn't – can you help?

The code is in Listing 1.

Listing 1

```
#include <iostream>
#include <iterator>
#include <map>
#include <string>

std::map<int, std::string> theMap;

void old_way(int key, std::string value)
{
  if (theMap[key].empty())
  {
    theMap[key] = value;
  }
}
void new_way(int key, std::string value)
{
  std::map<int, std::string>::iterator it
    = theMap.lower_bound(key);
  if (it->first < key)
  {
    theMap.insert(it,
      std::map<int, std::string>::value_type
      (key, value));
  }
}
namespace std
{
  ostream& operator<<(ostream&,
     map<int, string>::value_type const &rhs)
  {
    return cout << rhs.first
      << "=" << rhs.second;
  }
}
int main()
{
  new_way(1, "test");
  new_way(2, "another");
  new_way(1, "ignored");

  std::copy(theMap.begin(), theMap.end(),
   std::ostream_iterator<
    std::map<int, std::string>::value_type>
    (std::cout, "\n"));
}
```

## Critiques

### Paul Floyd <paulf@free.fr>

Here's a code critique without any blatant errors, at least in the sense that it compiles cleanly and seems to produce the results I expected.

Let's start by analysing `old_way`

```
void old_way(int key, std::string value)
{
  if (theMap[key].empty())
  {
    theMap[key] = value;
  }
}
```

Firstly, at least for C++98/03, I'd pass in the string argument by `const` reference.

In the `if` condition, the `std::map` bracket operator will search for key. This will be a binary search, of complexity O(logn). There are two possibilities:

1. key is found, and the bracket operator returns a reference to it. If the mapped string is not empty, the `if` condition evaluates to false, and flow passes to the end of the function. The mapped string could be empty, in which case the `if` body will be executed.
2. key is not found and the map default constructs a new string at that position and returns a reference to it. The default constructed string will evaluate `empty()` to true, and the body of the `if` statement will execute.

When the if body executes, there will be a second call to the bracket operator (so another O(logn)). This will succeed, and value will be assigned to the returned reference.

It isn't clear whether empty strings are allowed in the map, but as it stands `old_way` will overwrite existing elements keyed by an empty string.

```
old_way(3, std::string());
old_way(3, "overwrite");
```

Here the second call will find the empty string inserted by the first call. Since `old_way` does not distinguish between a missing element and an element keyed with an empty string, the second call overwrites the first insert.

Now lets consider `new_way`.

```
void new_way(int key, std::string value)
{
  std::map<int, std::string>::iterator it
     = theMap.lower_bound(key);
  if (it->first < key)
  {
     theMap.insert(it,
            MyMap::value_type (key, value));
  }
}
```

**ROGER ORR**

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

Again, I would pass value by **const** reference.

First there's a call to **lower_bound**, which will return the first element greater than or equal to key (or **end()** if all elements are less than key or the map is empty). Let's go through these 3 possibilities.

If the key is already in the map, the following test for less than will be false, and there will be no attempt to insert into the map.

If the key isn't in the map, but there is at least one element with key greater than the key being added. In this case the test following **lower_bound** will be false and the element is not inserted. This is wrong, and it doesn't show up because there in the example, there is no attempt to insert a new key that is not the largest in the map (the successful insertions have keys 1, 2 then 3). This can be fixed by swapping the order of the arguments to less than in the test.

The last possibility is that the map is empty or that the key being inserted is greater than all keys in the map. In this case **lower_bound** returns **end()**. Then the following test checks **it->first** when it is pointing to **end()**. Oops.

**dbx** has this to say:

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0x8068e68
    which is 144 bytes into a heap block of size 1328 bytes at 0x8068dd8
This block was allocated from:
      [1] operator new() at 0xe910670b
      [2] std::__node_alloc<true,0>::_S_chunk_alloc() at 0xea55d1c2
      [3] std::__node_alloc<true,0>::_S_refill() at 0xea55d0c4
      [4] std::__node_alloc<true,0>::_M_allocate() at 0xea55cfe7
      [5]
std::basic_string<char,std::char_traits<char>,std::allocator<char>
 >::reserve() at 0xea569c68
      [6] std::__copy() at 0xea5bd475
      [7] std::_Init_timeinfo() at 0xea5ba92c
      [8] std::_Locale_impl::make_classic_locale() at 0xea5be306
stopped in new_way at line 37 in file "cc87.cpp"
   37      if (it->first < key)
```

Changing the test to include a check for **end()** fixes this,

```
if (it == theMap.end() || key < it->first)
```

When I looked at these uninitialized values, they were always 0, which means that the **it->first < key** was always true. However, it would be impossible to insert an element with a key having a negative value. It also probably explains the non-deterministic behaviour described.

When the insert does work correctly, it reuses the iterator as a hint for the insertion position, which will hopefully save on an O(logn) search. I would recommend adding some debug check or assert to the return from the insert, as we are expecting the insert to always succeed. This overload of **std::map::insert** returns an iterator pointing to either the inserted element or the existing element.

```
MyMap::iterator ret = theMap.insert(it,
                    MyMap::value_type
                    (key, value));
assert(ret->first == key)
```

It's worth noting that **new_way** has no problem handling empty strings, so

```
new_way(3, std::string());
new_way(3, "overwrite");
```

does not overwrite the element with key 3.

Now for the one thing that did stick out even at first glance.

```
namespace std
{
  ostream& operator<<(ostream&,
    map<int, string>::value_type const &rhs)
  {
    return cout << rhs.first
      << "=" << rhs.second;
  }
}
```

What's that? Polluting namespace **std**, taking an (unnamed) **ostream** reference, but sending output to **cout**. There's no easy way to define an **operator<<** for types that reside in the **std** namespace. Instead, I just wrote a boring function

```
void printMyMap(std::ostream& ostr,
                const MyMap& myMap)
{
  for (MyMap::const_iterator iter =
    myMap.begin(); iter != myMap.end(); ++iter)
  {
    ostr << iter->first << "="
         << iter->second << "\n";
  }
}
```

(would be a bit nicer with C++11 and **foreach**).

A couple of style issues. I wouldn't use a global for **theMap**, but I suppose it's OK for illustrative purposes. I would use a **typedef** to keep down the typing, like this

```
typedef std::map<int, std::string> MyMap;
```

### Stefan Schiffler <stefan.schiffler@scils.de>

This code contains several mistakes, I will address the most important one only. First of all the **old_way** inserts the new value not only if the key does not exist, but also if it exists but the value string is empty. Unfortunately **new_way** makes it worse. The **lower_bound** function returns an iterator pointing to the first element which is not considered to go before key. It might return **theMap.end()**, so calling **it->first** in the **new_way** function is the most important problem here. It can be fixed by

```
auto it = theMap.find(key);
if( key==theMap.end() ){
//... insert to the map
}
```

But actually what you want to use instead of **new_way** is

```
theMap.insert( std::make_pair(key,value) );
```

which is inserting the new elements if they are not already in the map, otherwise returning an iterator to the existing element. So **std::map::insert** exactly does what you want to implement in **new_way** by yourself. And the best thing: You do not have to write a test for it :)

So the complete code would be

```
#include <iostream>
#include <map>
int main()
{
  std::map<int, std::string> theMap;
  theMap.insert( std::make_pair(1,"test") );
  theMap.insert( std::make_pair(2,"another") );
  theMap.insert( std::make_pair(1,"ignored") );

  for( auto item : theMap )
    std::cout << item.first << " = "
              << item.second << std::endl;
  return 0;
}
```

### Dave Cridland <dave@cridland.net>

The code doesn't work because it's hitting undefined behaviour, and misusing **lower_bound** as well. But let's stop for a moment and look at why **old_way** was the wrong method.

The old way, of course, mostly works – and moreover, the knowledge of why it works is clearly visible in the code. The **operator[]** of a **std::map** will instantiate a value with the default constructor automatically, and so with a sensible class like **std::string**, you get an empty string. This is visible in the code, the test for "Is this key in the map?" is to look at the value and see if it's an empty string.

If an empty string is actually put there, though, it'll also be treated as a new entry, and so a subsequent set would erroneously work:

```
old_way(1, "");              // Essentially works.
old_way(2, "Test");          // Also works.
old_way(2, "Another test"); // Doesn't insert -
                             // so works.
old_way(1, "Oooops");        // This *does*
  // insert, because the existing value is
  // empty() - fail.
```

So switching away from using **operator[]** to test for the key's presence is certainly the right choice here – it can also make the code a little faster.

But in the **new_way**, **lower_bound** always returns an iterator, just as **operator[]** always returns a value. And again, the iterator might have been invented for the purpose – for an empty **std::map** for instance, **lower_bound** will return the same marker iterator as the **Map.end()** does. This, like a **NUL** at the end of a string, is purely a marker and isn't a valid iterator at all, so the moment you dereference it – as in the conditional in **new_way** – all bets are off. Because the key is a POD – a Plain Old Data type rather than an object with a constructor – it may actually exist and have memory, but it'll be uninitialized, and the result of the conditional is undefined behaviour. In other words, nobody can tell you what it'll do – it'll sometimes work and sometimes fail.

So let's consider the case where the call to **lower_bound** gives you a valid iterator. This will be an iterator pointing to the first element that's 'not less' – ie, equal or greater for an **int** – than the argument. So if the map contains only a key of 2, and you ask for the **lower_bound** of 1, it'll point at the 2, and your conditional expression will be false. Ask for 3, and it'll give you **end()** – and give you undefined behaviour. Ask for 2, and it'll give you 2, and your conditional expression evaluates, finally, the way you want it to.

When it actually works, the insert is a good one – it'll use the hint correctly, though probably ignore it, and thankfully it won't ever overwrite existing elements, in part because **insert()** will only insert a new key into the map (contrasting with **multimap**).

Wait... What was that? Your brief was to add "items into a map only if they weren't already there"? Ah – that's what **insert()** does.

So how about:

```
void third_way(int key, std::string value) {
  theMap.insert(std::make_pair(key, value));
}
```

In fact, it's not really clear why you'd want to have a function here at all, is it?

If you did, you'll be wanting to have an argument of **"const std::string &"** rather than simply **"std::string"**; this will remove a copy. With C++11, you can also use a move here in a different overload. But really, wrapping one relatively simple line of code in a set of overloaded functions seems like overkill.

The other thing that struck me about the code was the use of the **ostream_iterator** – clever stuff, but I note that in the newly defined **operator<<** the **ostream** argument is actually ignored, and **cout** used instead. This works in the example given, but it'll be impressively confusing if you try to write to a file instead – not only will this write be directed to standard output, but anything subsequent in the chain of **"<<"** operators will be as well. If it's sufficiently buried, that'll be an interesting bug to trace.

Finally, I'd note in passing that if you did want to check for the existence of the key prior to constructing a **std::string** from the string literal you have, you might want to do things with **find**, not **lower_bound**, since it's generally easier:

```
void fourth_way(int key, const char * value) {
  auto it = theMap.find(key);
  if (it == theMap.end()) {
    theMap.insert(std::make_pair(key, value));
  }
}
```

You'll see I've used a C++11-ism of **auto** instead of the full iterator type name.

But you were right in that you'll gain a bit of performance when the hint it present, so perhaps:

```
void fifth_way(int key, const char * value) {
  auto it = theMap.lower_bound(key);
  if (it == theMap.end() || it.first != key) {
    theMap.insert(it,
      std::make_pair(key, value));
  }
}
```

As I say, this only makes sense if you feel constructing a **std::string** before the insert is going to be an issue. Ironically, it will never be in C++11, because that defines a new templatized overload that'll only perform the type conversion if the insert will succeed, so the simplest solution ends up being the best, in C++11 at least:

```
#include <iostream>
#include <iterator>
#include <map>
#include <string>
std::map<int, std::string> theMap;
namespace std {
  ostream& operator<<(ostream& os,
    map<int, string>::value_type const & pair) {
      return os << pair.first << "="
                << pair.second;
  }
}
int main() {
  theMap.insert(std::make_pair(1, "test"));
  theMap.insert(std::make_pair(2, "another"));
  theMap.insert(std::make_pair(1, "ignored"));
  std::copy(theMap.begin(), theMap.end(),
    std::ostream_iterator<std::map<int,
    std::string>::value_type>(std::cout, "\n"));
}
```

### Martin Janzen <martin.janzen@gmail.com>

Where to start? First things first: let's create a **typedef** for the map. Writing **std::map<int, std::string>** everywhere is error-prone, and clutters up the code needlessly:

```
typedef std::map<int, std::string> map_type;
```

Next, the single instance **theMap** is a global variable, with all of the usual associated problems: Functions are inflexible, being usable only with **theMap**. Test cases are more difficult to write because we can't count on starting with a clean map. We may encounter naming conflicts and linkage errors because **theMap** isn't static, nor even in a separate namespace; and so on.

An improved design might be object-based, encapsulating the map and its related functions:

```
struct TheMap
{
  typedef std::map<int, std::string> map_type;
  typedef map_type::value_type       value_type;
  map_type theMap;
  void insert(int key, std::string value);
  // ...other interface functions...
};
```

Now we can instantiate one or more maps anywhere we want.

We might further prefer to hide the implementation by making **theMap** and its related **typedef**s private. However, we will then have to write a potentially large number of forwarding functions if we want to emulate part or all of the **std::map** interface.

Alternatively, we could create a class which is derived publicly from std::map, wrapping only the required constructors (perhaps using C++11's constructor inheritance to save typing). However, inheriting from STL containers, while possible, is usually considered to be best avoided – possibly a subject for a future Code Critique?

If we're really sure we need only one instance, and only in this single compilation unit, we can at least put it into an anonymous namespace, together with its related functions:

```
namespace
{
  typedef std::map<int, std::string> map_type;
  map_type theMap;

  void map_insert(int key, std::string value) {
   ... }
}
```

For simplicity we'll use this approach for the rest of this response.

* * *

Let's go on to the insertion function. The code reviewer was correct to say that the old way is wrong, because **std::map::operator[]** will insert a new entry every time you try to search for a key that isn't already in the map. This is a memory leak. Also, it's likely to lead to surprises later; say, when iterating through a map which now contains a large number of empty values. And it precludes the possibility of storing an entry whose string really is supposed to be empty.

Unfortunately, the new way is also wrong. The **std::map::lower_bound()** function is not the right tool for the job, since it doesn't test for equivalence. Instead, it returns an iterator which points to the first element whose key is equivalent to or goes after the specified key; with the result that the code may or may not work depending on the order in which items are inserted.

The **std::map::find()** function is probably what the author had in mind:

```
void map_insert(int key, std::string value)
{
  if (theMap.find(key) == theMap.end())
    theMap.insert(
      map_type::value_type(key, value));
}
```

Passing an iterator as a hint in the first argument to **std::map::insert()** is pointless here, so I've removed it. The hint may improve performance if we know that the new entry can be inserted just before the iterator, but if our search just failed then the iterator will point to **theMap.end()**, which doesn't help.

However, it's really not necessary to look for the key at all, since the **std::map::insert()** function already does exactly what we want. If key is not already in the map, the entry is inserted; if it is, the map is unchanged:

```
void map_insert(const int key,
  std::string value)
{
  theMap.insert(
    map_type::value_type(key, value));
}
```

In addition, the **std::map::insert()** function will return a pair consisting of an iterator and a **bool**, which respectively point to the entry for that key and indicate whether an insertion took place. The example code doesn't bother to report the outcome, but it might be useful to have the function return **true** if an insertion took place:

```
bool map_insert(const int key, std::string value)
{
  return theMap.insert(
    map_type::value_type(key, value)).second;
}
```

* * *

Having fixed the insertion error, let's turn our attention to the test code, which raises some rather more interesting issues.

The **test operator<<()** function has a small bug, in that it accepts an **ostream&** parameter but always writes to **std::cout**.

Worse, though, is the fact that this function has been placed into namespace **std**. This is 'undefined', according to 17.4.3.1. Template specializations in namespace **std** are fine (eg., for **std::hash<T>**), but new declarations or definitions are not allowed.

How can we fix this?

If the rhs parameter were a user-defined type, we could put the **operator<<()** function into the same namespace as that of the type, relying on argument-dependent lookup to do the right thing.

Unfortunately, rhs really is in namespace **std**, being a **std::map::value_type**, so this won't work here.

There are any number of alternatives:

1. If we'd replaced the global **std::map** with a class that contains a **std::map**, as shown earlier, we could write an **operator<<()** function which is qualified by the class name, and ADL will work:

   ```
   std::ostream& operator<<(std::ostream& os,
     const TheMap::value_type& rhs)
   {
     std::cout << rhs.first << "=" << rhs.second;
   }
   ```

   Note that this function may need to be declared as a friend of the class, depending on the access control we've specified.

2. For a plain **std::map**, though, it's possible to write a wrapper class containing only a reference to our map's **value_type**, and an **operator<<()** function that operates on the wrapper instead:

   ```
   struct map_wrapper
   {
     map_type::value_type const &ref;
     map_wrapper(map_type::value_type const &val)
     : ref(val) {}
   };
   std::ostream& operator<<(std::ostream &os,
      map_wrapper const &rhs)
   {
     return os << rhs.ref.first << "="
               << rhs.ref.second;
   }
   std::copy(theMap.begin(), theMap.end(),
     std::ostream_iterator<map_wrapper>(
       std::cout, "\n"));
   ```

   This works, but is a bit opaque: the purpose of the **map_wrapper** class is probably not going to be immediately evident to someone reading the code for the first time. (A literate comment would help, but how often do we see that?)

3. We could also provide a function which converts our map's **value_type** to a **std::string**, for which a suitable **operator<<()** is already defined. Then, use **std::transform()** to write it to a stream iterator:

   ```
   std::string format_map_entry(
     map_type::value_type const &val)
   {
     std::ostringstream os;
     os << val.first << "=" << val.second;
     return os.str();
   }
   ```

```
std::transform(theMap.begin(), theMap.end(),
  std::ostream_iterator<std::string>(
    std::cout, "\n"),
  format_map_entry);
```

This is slightly clearer, but at the cost of a lot of possibly expensive string manipulation.

4. We could write a function object that does the formatting, then replace `std::copy()` with `std::for_each()`:

```
struct format_map_entry
{
  std::ostream &os;
  format_map_entry(std::ostream &s) : os(s) {}
  std::ostream& operator()(
    map_type::value_type const &rhs) const
  {
    os << rhs.first << "="
       << rhs.second << "\n";
  }
};
std::for_each(theMap.begin(), theMap.end(),
  format_map_entry(std::cout));
```

5. Assuming that we're all using C++11 by now, it'd be clearer to replace the function object with a lambda function:

```
std::for_each(theMap.begin(), theMap.end(),
  [](map_type::value_type const &val)
  {
    std::cout << val.first << "="
              << val.second << "\n";
  };
```

(To write to a stream other than `std::cout`, replace `[]` by `[&]` or `[&os]` to capture the stream by reference.)

6. On the other hand, we could keep it simple and just use a plain old `for` loop – or, depending on taste, a fancy new range-based for, perhaps even with `auto`:

```
for (auto const &val : theMap)
  std::cout << val.first << "="
            << val.second << "\n";
```

The choice probably comes down to the likelihood that we will need to reuse the formatting function. In this case it's trivial, so it doesn't make much difference; but if it were more complicated, or needed to be standardised, or were likely to change frequently, it'd be good to provide a formatting wrapper or function.

* * *

Finally, all of this demonstrates that the original tests were completely inadequate. Because the test keys were added in sorted order, the function appeared to work despite the presence of a critical bug.

(A TDD purist might point out that, to be fair, the author did write just enough code to pass the test cases. I suspect that neither the users of this code nor the author's boss would be impressed by this line of argument.)

At the very least, we need a few more test cases, in unsorted order, including some with keys which are negative or zero, as well as some empty strings:

```
int main()
{
  map_insert(   0, "an");
  map_insert( -17, "this");
  map_insert(  42, "problem");
  map_insert(  -1, "is");
  map_insert(   9, "ex");
  map_insert(  42, "parrot");
  map_insert(   0, "not");
  for (auto const &val : theMap)
    std::cout << val.first << "="
              << val.second << "\n";
}
```

Use of a unit test framework such as Google Test, cppunit, or Phil Nash's Catch would be a further improvement, making it easy to add automated checks, and to report failing test cases and a summarized pass/fail result.

### James Holland <James.Holland@babcockinternational.com>

The code reviewer was correct when saying that `operator[]` should not be used. It is just a pity the reviewer did not say why. After all, the `old_way` function code seems to work correctly (although, if elements with null strings are acceptable, then `old_way` does not quite work). It is, perhaps, not as efficient as it could be. It fact, as we shall see later, the function body can be simplified to the extent that its body becomes a single statement.

The lack of efficiency comes about because the map is being traversed twice; once within the predicate of the `if` statement and once within the body of the `if` statement. The predicate first traverses the map to search for an element with a key of `key`. If no such element is found, one is inserted with a value of a default constructed `std::string`, namely, an empty string. Now there is definitely an element in the map with a key of `key`. Either it had already existed or it has just been inserted by the `if` statement. In either case, the empty member function then returns whether the element's value (a string) is empty or not. If the string is empty, it was because its element was just inserted by the `if` statement and therefore did not previously exist. The body of the `if` statement will now be executed to 'insert' the new element. This is where the second traversal takes place. Once the element is found, its value is changed to 'value' and the `old_way` function exits.

In an attempt to heed the reviewer's advice, the coder constructs a second function named `new_way`. Unfortunately, an error was made in the process. The problem is with the `if` statement of the new function, shown below, that is meant to determine whether a particular element exists within the map.

```
if (it->first < key)
```

After initialisation, the iterator (`it`) points to the location in the map where an element with a key of key would be inserted. The location may already be occupied by an existing element (if there is an existing element with a key equal to or greater than `key`) or the location may represent the end of the map (if there are no existing elements with a key equal to or greater than `key`). The latter will certainly be the case if the map is empty. It is fine to dereference the iterator and obtain the element's second value if the iterator points to an existing element. It is not valid to dereference the iterator if it points to a non-existent element. Unfortunately, this is what is happening in the `if` statement of the `new_way` function. The result is undefined behaviour. This is why the code sometimes works and sometimes doesn't.

To correct the problem the `if` statement has to be modified. One thing to notice is that if an element with a key of `key` does not exist; its lower bound and upper bound iterators will be equal. If this is the case, the element can be inserted in the map as required. The iterator already has the lower bound of the element and so all that needs to be done is to compare the iterator with the element's upper bound. The `if` statement then becomes as shown below.

```
if (it == theMap.upper_bound(key))
```

The program will now consistently work as expected.

Although `new_way` function now works correctly, it is no more efficient that the `old_way` function. This is because the function is still performing two traversals; one for `lower_bound()` and one for `upper_bound()`. These two functions could be combined into one function, namely, `equal_range()`. This will require only one traversal of the map. Fortunately, however, there is no need to pursue this approach as there is a simple and more efficient way of achieving the desired effect.

The map member function `insert()` first checks to see whether an element already exists within the map. If the element is not in the map, it is inserted. If it is in the map, no further action is taken. This is exactly what is required. The resultant `new_way` function is shown below.

```
void new_way(int key, std::string value)
{
  theMap.insert(std::make_pair(key, value));
}
```

This version of the **new_way** also caters for the situation where the insertion of elements with null strings is valid. This is because insert only inserts the new element if it cannot find an element in the map with a key of key; not whether an existing element has a null string.

### Marcel Marré <marre@links2u.de> & Jan Ubben

Any sporadic bug hints at undefined behaviour. In this case, the first two lines of **new_way** result in undefined behaviour. Querying **lower_bound** from a map is not guaranteed to return a dereferenceable iterator. It will return the map's end when the key provided is greater than all keys already in the map, which is, of course, the case when the map is empty.

The returned iterator is unconditionally dereferenced in **new_way**. A naive way of fixing this would be to change the condition to:

```
if(theMap.end() == it || it->first < key)
```

However, we can do better by utilising **std::map::insert** as the interface suggests, which alone covers the requirements completely. While it returns whether or not the value was actually inserted into the map, this is not in fact used in the code, so the whole insertion can be written as:

```
void new_way(int key, std::string value)
{
  theMap.insert(std::map<int,
    std::string>::value_type(key, value));
}
```

The reason why the reviewer rejected **old_way** is also likely due to misusing the map interface. Using **theMap[key]** in two lines means that the lookup for that key is performed twice. Furthermore, if no entry has been made for key, a default-constructed value for this is added automatically. This means that in **old_way**, an empty string would not be considered a valid entry.

However, there are additional problems with the code.

Overloading functions in namespace **std** is undefined. Additionally, the code abuses the interface of **operator<<** by ignoring the provided **ostream** parameter. It forces output to **std::cout** and returns it, which can lead to surprising results when chaining output operations into a different **ostream**. Therefore, a named function is more appropriate:

```
void printWithNewlineToCout(std::map<int,
  std::string>::value_type const & keyValue)
{
  std::cout << keyValue.first << "="
            << keyValue.second << "\n";
}
```

This is used slightly differently:

```
std::foreach(theMap.begin(), theMap.end(),
  printWithNewlineToCout);
```

If this were not merely sample code, one would also not want to use a global variable for **theMap**.

### Raimondo Sarich <rai@sarich.co.uk>

Stepping back for a moment, I would question the requirement to add to the map only if the key cannot already be found, regardless of the value. Let us proceed on the assumption that the requirement is valid.

Firstly, the problem with using **operator[]** has already been highlighted, but we should understand the reason: The operator will create the entry in the map if it does not already exist (using the element's default constructor); clearly undesirable in this application.

The observed problem is the intermittent failure. It is interesting to note that a release build (using VC2010) indeed sometimes works and sometimes produces no output, whilst a debug build always fails with an assertion. The failures are caused by the dereferencing of the iterator returned from **lower_bound**. **map::lower_bound** returns an iterator to the first element in the container whose key is not considered to go before **k** (the lookup key), or **map::end** if all keys are considered to go before k[1]. Furthermore, **map::end** does not point to any element, and thus shall not be dereferenced[2]. The first call of **new_way** sets it to **map::end**, and the subsequent dereferencing (**it->first**) results in undefined behaviour. This means the dereference may take any value, and this is why the code sometimes works; occasionally the essentially random value is less than **key**. In the debug build **operator->** is checked for validity, hence the assertion.

**new_way** needs only a simple fix to make it optimal, as was first pointed out to me by our distinguished columnist. The find/check/insert-with-hint idiom first finds the insertion location, checks whether to go ahead with the insertion, and then performs the insert with a hint, avoiding the need for a second search (unless the map changes before the insert). Changing the conditional to **if (it == theMap.end())** will do the trick.

The more insidious problem does not cause a failure and allows us to copy **theMap** to an **ostream_iterator**, another popular idiom for container types. The enabling code is the definition of **operator<<**, which demonstrates two problems.

Firstly, it ignores the first parameter and instead always outputs the value to **cout** and then returns **cout**. Although it works in this code, it might cause some surprise when **theMap** is output to an **ofstream** (or indeed any other **map<int,string>** is sent to any output stream other than **cout**).

The second problem is that the definition is made in namespace **std**, which results in undefined behaviour[3]. Unfortunately, the copy-to-**ostream_iterator** idiom cannot be applied to a **map** because the contained type is a **pair**. Since both **pair** and **ostream** are in namespace **std**, the argument dependent lookup for **operator<<** remains in namespace **std**, which does not contain an appropriate definition. Since we are restricted from changing namespace **std** an alternative must be found, reasonable choices being **for_each** or **transform**.

A final few niggles:

- A sprinkling of pass by const reference. In the given code, temporary **string**s are created from the **const char*** strings and then passed by value into **new_way** resulting in the construction of another **string**. It would be interesting to investigate whether a clever compiler optimises away the superfluous **string** construction, but pass by **const** reference certainly avoids it, and is also preferable in case an actual **string** is passed.
- Removing the global variable.
- A couple of **typedef**s, and referring to **std::pair** directly, which I find clearer.
- A namespace so we don't pollute the global namespace. Not strictly necessary in this simple example, but a habit worth forming.

And we are done.

```
#include <map>
#include <string>
#include <iostream>
#include <algorithm>
namespace {
typedef std::map<int, std::string>
  map_int_string;
typedef std::pair<int, std::string>
  pair_int_string;
void insert_if_not_found(map_int_string& theMap,
    int key,
    const std::string& value) {
  map_int_string::iterator it =
    theMap.lower_bound(key);
  if (it == theMap.end())
    theMap.insert(it,
      pair_int_string(key, value));
}
```

```
void print_pair(const pair_int_string& a) {
  std::cout << a.first << "="
            << a.second << "\n";
}
} // namespace Anonymous
int main() {
  map_int_string theMap;
  insert_if_not_found(theMap, 1, "test");
  insert_if_not_found(theMap, 2, "another");
  insert_if_not_found(theMap, 1, "ignored");
  std::for_each(theMap.begin(), theMap.end(),
    &print_pair);
}
```

Or are we? The concept of adding to a **map** only if the key cannot be found is more generally applicable, at least to other types of **map**. Perhaps we should use templates to capture the general concept, certainly if it might be used elsewhere. However, this does result in a little overhead of having to identify the template specialisations we need created, either by calling the function with the right type or by specifying the template parameters. The template functions probably belong in a different header and namespace, but this review is already long enough.

[Ed: complete sample code was provided, but was elided to save space]

#### References


[1] http://www.cplusplus.com/reference/map/map/lower_bound/

[2] http://www.cplusplus.com/reference/map/map/end/

[3] [C++11: 17.6.4.2.1/1]: The behavior of a C++ program is undefined if it adds declarations or definitions to namespace **std** or to a namespace within namespace **std** unless otherwise specified. A program may add a template specialization for any standard library template to namespace std only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.


### Giuseppe Vacanti <giuseppe@vacanti.org>

I'm not too sure what the **old_way** was trying to achieve. Using **operator[]** on a map will either fetch an existing element corresponding to the key, or create one if it does not exist (assuming a suitable default constructor is available). Additionally, **map[key].empty()** does not test whether there is no element corresponding to key, but in this case (the value type is **std::string**) checks whether the string corresponding to key is empty. If we could consider an empty string as indicating that there is no value corresponding to key, then **old_way** would achieve its goal, but in this case only because the type of the value has a method called **empty()**.

The **new_way** has a vague notion that iterators have something to do with the problem, but why use **lower_bound**? This method returns an iterator to an element of the map whose key is no less than the key passed, and the less-than test is used to determine that there is no key equal to the one passed to the method. I use **lower**/**upper_bound** and **equal_range** only if I want to select elements in a map, not if I want to insert a new element.

In order to insert an elements without replacing an existing one, use, well, **insert** (I specify the type fully, but one would use auto here to save some typing):

```
std::pair<
  std::map<int, std::string>::iterator, bool> p
  = theMap.insert(std::make_pair(key, value));
```

The method insert inserts only if the element does not already exist in which case **p.second==true**. **p.first** points to the newly inserted element, or the existing element with the same key.

As a side remark, those strings should be passed by constant reference and not by value.

## Commentary

The code in this critique is based on a very common pattern that I have encountered a number of times over the years. The fundamental problem is that people over-use the convenience function **operator[]** provided in **map** – in some languages this may be the only, or best, choice in most cases but this is not the case in C++.

It is also sadly all too common to see people writing their own code to perform what an existing library call – in this case **map::insert** – already does.

One 'meta' comment about this critique is with the original reviewer's remark: "the code reviewer said I shouldn't be using **operator[]**'. It doesn't appear that the reviewer gave a clear enough explanation of what the problem was that needed fixing. I have seen similar problems with "FIXME" or "TODO" comments in code bases…

## The Winner of CC 87

This critique attracted quite a few entries and the problems in the code were very well covered. There were also some good suggestions for alternatives to the slightly more subtle problem of defining our own streaming operator in the **std** namespace. Marcel and Jan provided an explanation for why this is a bad thing (other than "it's illegal").

However, I was particularly taken with Martin's discussion about why the existing tests were poor and to my mind this gave him the edge – and the prize – for this issue.

## Code Critique 88

(Submissions to scc@accu.org by August 1st)

I'm trying to write a simple program to read and process lines of text from the console. I've got a problem – and have stripped the program down to a small demonstration of the it. If I run the program and type

```
add 1 2
```

it prints, as I'd expect:

```
add( 1 2)
```

and is back ready to read the next line and I can, for instance, type **subtract 2 3** and it echoes back **subtract( 2 3)**.

But if I type just

```
add
```

then the program prints

```
add(
```

and although it seems to read more lines it no longer seems to process them.

Additionally, with one compiler (MSVC), I get this warning and don't know why:

**cast between different pointer to member representations, compiler may generate incorrect code** – which worries me. To be honest I don't really know why the **static_cast** is needed but I can't get it work any other way.

Can you help fix the problems presented and perhaps suggest some other improvements?

The code is in Listing 2 (next page).

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

# Standards Report

## Mark Radford reports the latest developments in C++ Standardization.

Hello and welcome to my latest standards report.

The ISO C++ Standards Committee are meeting again soon. The meeting will be in Rapperswil, Switzerland, 16th–21st June. Given that the final copy deadline for *CVu* this time around is the 21st June, in theory I could cover the first half of the meeting. I did this with last February's meeting (Issaquah, WA, USA), but, unfortunately, my current work commitments make it impractical this time around. Therefore I will confine myself to looking at some of the papers in the pre-Rapperswil mailing, which can be found at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/#mailing2014-05.

As usual, I have much to say about the C++ standards process, but in this report I first get to write about another standards process, namely one for the C language. Many thanks to the convenor of the BSI C Panel for updating me on their progress.

### C Standardisation

Last January I reported that the C Panel are working on a floating point bindings TS, which has five parts to it. Part 1 (binary floating point

**MARK RADFORD**

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

## Code Critique Competition 88 (continued)

Listing 2

```
#include <iostream>
#include <map>
#include <sstream>
#include <string>
class Processor;
typedef void (Processor::*Pmf)(
  std::istream &is);
class Processor
{
public:
  void run(std::istream & is);
  void addCmd(std::string const &cmd, Pmf pmf)
  {
    cmds[cmd] = pmf;
  }
protected:
  Processor() = default;
private:
  std::map<std::string, Pmf> cmds;
};
void Processor::run(std::istream &is)
{
  std::string line;
  while (std::getline(is, line))
  {
    std::istringstream iss(line);
    std::string cmd;
    iss >> cmd;
    Pmf pmf = cmds[cmd];
    if (pmf) (this->*pmf)(iss);
  }
}
class Identity
{
  int id = getNext();
  static int getNext()
  {
    static int seed;
    return ++seed;
  }
public:
  int getIdentity() { return id; }
};
// (Stripped down code)
class Calculator : public Processor,
  public virtual Identity
{
public:
  Calculator();
private:
  void add(std::istream &is);
  void subtract(std::istream &is);
  // etc
};
Calculator::Calculator()
{
  addCmd("add",
    static_cast<Pmf>(&Calculator::add));
  addCmd("subtract",
    static_cast<Pmf>(&Calculator::subtract));
}
void Calculator::add(std::istream &is)
{
  // stub ...
  std::cout << "add("
    << is.rdbuf() << " )\n";
}
void Calculator::subtract(std::istream &is)
{
  // stub ...
  std::cout << "subtract("
    << is.rdbuf() << " )\n";
}
int main()
{
  Calculator calc;
  calc.run(std::cin);
}
```

Listing 2 (Cont'd)

arithmetic) was at the draft technical specification (DTS) stage. Following the DTS ballot, part 1 has now been approved for publication (but is not yet published, although publication is just a matter of time). Part 2 of the TS (decimal floating point arithmetic) was at the preliminary DTS (PDTS) stage, but has now moved to the DTS stage, while part 3 (Interchange and extended types) and part 4 (supplementary functions) will advance to PDTS ballot soon. Work is being done on part 5 (supplementary attributes) but there is no public draft yet.

## Concurrency and Parallelism

Moving on to C++, once again there is a buzz of activity on the concurrency and parallelism front. A couple of SG1 (Concurrency and Parallelism study group) papers (in the pre-Rapperswil mailing) caught my eye: first was 'Comments on continuations and executors' (N4032) by Anthony Williams. In passing note that document D3904 (mentioned in the first line of N4032) is the working draft of the concurrency extensions TS, and appears in the mailing (with revisions) as N3970. In N4032 Anthony expresses some reservations about the executers section of the concurrency TS which draws on 'Executors and schedulers, revision 3' (N3785). This is not the first time reservations about N3785 have been expressed: the first incarnation of this paper (N3378) prompted some lively debate, and some reservations, within the BSI C++ Panel. However, there have been no competing proposals in the area of scheduled/threaded execution, until now. In 'Executors and Asynchronous Operations' (N4046), Christopher Kohlhoff has done considerable work in presenting an alternative (note he has also provided a reference implementation of his proposal). Before leaving the topic of concurrency, I'll quickly draw attention to another paper by Anthony Williams entitled 'synchronized_value<T> for associating a mutex with a value' (N4033). This paper describes a simple and useful idea: a template that encapsulates the coding for synchronizing access to a single value using a mutex.

## Monads

Moving on to things other than concurrency, a work colleague pointed out to me a paper entitled 'A proposal to add a utility class to represent expected monad' (N4015) by Vicente J. Botet Escriba and Pierre Talbot. Monads are a concept from the world of functional programming, and this paper is a proposal for a utility class called **`expected<E,T>`**, where **`T`** is the value type, and **`E`** is an error type. **`expected<E,T>`** is similar to **`optional<T>`** except that **`expected<E,T>`** also holds error information. Note, in passing, that this idea is not new: it appeared back in 1994 in Barton and Nackman [1], where it was called **`Fallible<T>`**. Barton and Nackman's **`Fallible<T>`** was a little less general in that it simply held a boolean value to indicate if the value was valid, or if an error had occurred. The class **`expected<E,T>`** described in N4015 is, to my knowledge, the first time the idea has been proposed for standardisation.

## Destructive Move

With the advent of C++11, move operations are a major addition to C++. Three years later, 'Destructive Move' (N4034) by Pablo Halpern identifies a problem with the way move operations are specified and proposes a solution. The problem is, the requirement that move constructors leave the object moved from in a valid (but unspecified) state, means it isn't always possible to implement the move constructor in such a way that it can give a no-throw guarantee. Those who want to know the reasons why, can find them in the paper. The point is that 'destructive move' – a move operation that does not require the object moved from to be left in a valid state – can be implemented in such a way that it can give a no-throw guarantee. The paper proposes the addition of a couple of function templates to the library in support of destructive move operations.

## Uniform Copy Initialisation

Proposals that make aspects of C++ simpler or at least less surprising, are always welcome. C++11 introduced the uniform initialisation syntax using curly braces for initialisation rather than parentheses (or an equals sign). This, among other things, serves to combat programmers being caught out by C++'s "most vexing parse" [2] (where a declaration such as **`std::vector<int> vec(std::istream_iterator<int>(filename), std::istream_iterator<int>());`** is parsed as a function declaration rather than the declaration of a **`vector<int>`** object, the latter being what was intended). However, with brace style initialisation, the declaration **`C c2 = { "Steve", "Brown" };`** will not compile if the relevant constructor in class **`C`** happens to be declared as explicit. This is addressed in Nicolai Josuttis' 'Uniform Copy Initialization' (N4014). The author says that programmers "...waste a lot of time looking the reason of the error", and asserts that the syntax with the equals sign is not in any sense more dangerous than the syntax without the equals sign. His proposal is simple: allow the equals sign, and let it have no effect.

## Finally

I can only comment on a few of the papers in the mailing, which contains many more papers (I counted 86!). For example, each group has its issues/defects documents. Also I'll just slip in a quick mention of Andrew Sutton's 'Working Draft, C++ Extensions for Concepts' (N4040), which has an updated version. It would have been nice to have reported up to the minute events from the ISO C++ meeting in Rapperswil which is, at the time of writing, only a few days away. However, as I said in my introduction, it really wouldn't be practical for me to give it real time coverage this time around. I'll catch up with events from Rapperswil, but that's for next time.

## References

[1] *Engineering and Scientific C++: An Introduction with Advanced Techniques and Examples* by John J. Barton and Lee R. Nackman, Addison Wesley, 1994

[2]http://en.wikipedia.org/wiki/Most_vexing_parse

# Bookcase

## The latest round-up of book reviews.

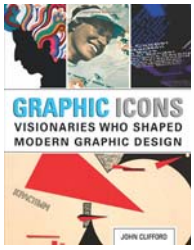
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.

Astrid Byro (astrid.byro@gmail.com)

### Graphic Icons




**By John Clifford, published by Peachpit Press. 240pp. ISBN 978-0-321-88720-7**

**Reviewed by Alan Lenton**

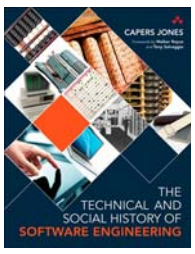
A rapid (two to four pages each) illustrated look at the art movements and innovators that have inspired modern graphic design. A must for budding and experienced graphic designers, not to mention digital user experience programmers and designers. The pages are chock full of illustrations guaranteed to provide inspiration and example for your day to day work.

Obviously, any book like this must to a certain extent be a personal choice of the author, but there was one glaring omission which surprised me. That of the surrealists, whose influence on modern design has been massive. In fact a number of the designers featured cite Man Ray, for instance, as a major influence. A very strange absence.

Personally, I would have also included typographer Matthew Carter who produced the first digital fonts properly designed for screen display – the sans-serif Verdana and the gorgeous serif Georgia. But these are nit-picks. John Clifford has done an excellent job of providing something which is fun to read, educating, and inspiring of new ideas. Go for it!

Recommended.

### The Technical and Social History of Software Engineering




**By Capers Jones, published by Addison Wesley in 2013. ISBN 978-0-321-90342-6 452pp**

**Reviewed by Alan Lenton**

This book has been sitting on my desk for more than a month since I finished reading it. The thorny question was, how to review it. So, what's the problem? Well, on the one hand it's quite interesting, and would be useful to someone trying to write a history of computing.

On the other hand the title is a complete misnomer. It's nothing much to do with the social history of software engineering, and a somewhat lopsided view of the technical history, concentrating on business applications and the rise of function point metrics, which the author champions.

After a brief nod towards aspects of pre-digital computing, the book is basically a linear description of the commercial market.The earlier chapters clearly make use of a lot the information contained in the Wikipedia. This was not a wise choice. The Wikipedia's striving for academic respectability has resulted in vast swathes of material relating to personal computing in the 1980s and 90s being removed. They were either oral history, or from long defunct computing magazines, and therefore had no 'proper' citations, according to the Wikipedia. In social history terms this is a critical omission.

Allied to this is the complete absence of any discussion about the role of computer games in the history of computing, both as an introduction to using computers, and as an influence on software practitioners. The author briefly mentions this problem later on in the book, but makes no attempt to rectify it. It is understandable that the author is not familiar with the games industry, coming as he does from a commercial background. However, he should have made himself familiar with the industry if he wanted to write a book on the history of computing (social or otherwise). The attitude shown is redolent of a common theme in certain parts of the industry until the start of the current century. It is an attitude that considers games to be a waste of otherwise useful computing power.

Almost completely absent from the book is any attempt to discuss the social history of computing – either its effects on society, or the social development of its practitioners. For instance, the rise of open source software is as much about the politics and sociology of computing, as it is about technical development, and yet the topic is barely touched in the book. Neither are the very early struggles of programmers of the very first computers to become recognized in their own right, instead of being considered mere lab technicians by the academics who wanted to study 'computing'.

From the point of view of a technical history there is no feel of an overall concept, leading to a large proportion of the book being one or two page summaries of selected companies in the industry. Even at this level there seems to be no understanding of the extent to which the big 'non-computing' businesses have become, over the period covered, software houses specializing in whatever was their business before the rise of cheap computing power. The classic case for this is, of course, the big banks who have gone from being banking houses with a software department to being software houses with a banking licence. The fact that most boards of directors of these bodies have not yet caught up with the reality of their business does not absolve the author of a book on software engineering history from noticing the metamorphosis!

All in all, a rather disappointing read...

The reviewer is a professional programmer, and holds a degree in Sociology from Leeds University.

### C++ For the Impatient

**By Brian Overland, published by Addison Wesley, ISBN: 978-0-321-88802-0, pages 657**

**Reviewed by Andrew Marlow**


Not recommended.

The book consists of 18 chapters on various aspects of the language, starting with data, operators and control structures and ending with various library facilities such as streams, the string class, STL containers and algorithms, random numbers and regular expressions. There are three appendixes: Rvalue references, C++11 feature summary and (rather strangely) tables of ASCII codes. Each chapter has some exercises at the end. There are numerous examples and some diagrams to explain certain things such as

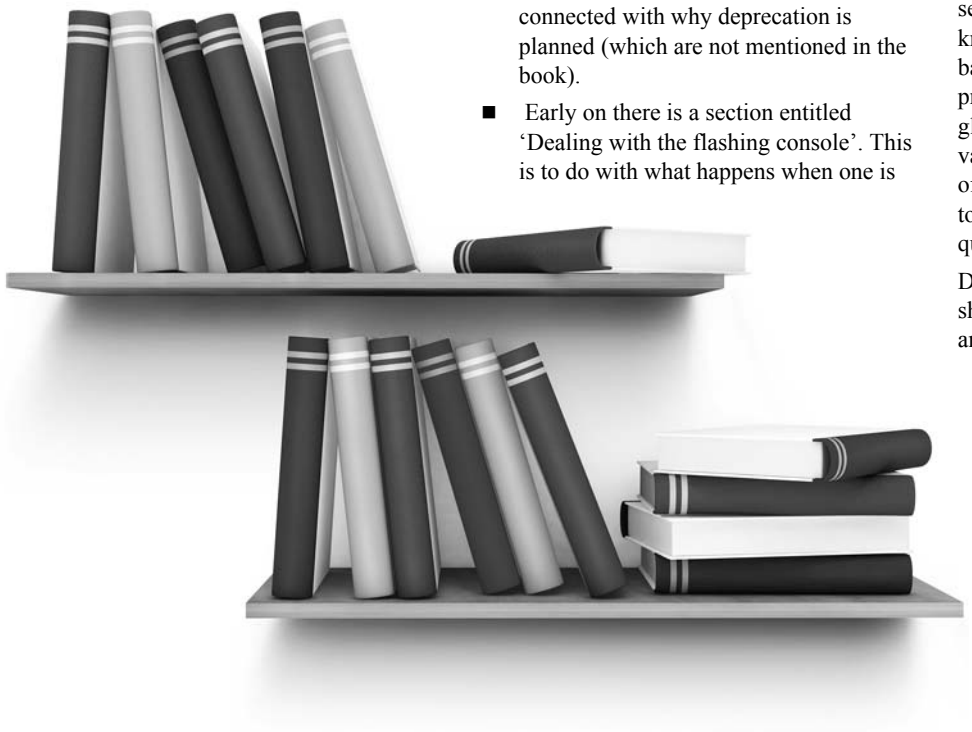
memory layout issues (trees, hash tables, move semantics etc).

The title probably derives from the well-known book *TeX for the Impatient*, which says of itself "It will enable you to master TeX at a rapid pace through inquiry and experiment, but it will not lead you by the hand through the entire TeX system".

The trouble is, the C++ book does seem to treat C++ as the primary interest. This shows in the way the book is laid out; it is organised by the classic divisions that one uses to teach programming languages. This gives the impression that it will take the reader through most of C++. This approach does not serve the impatient reader well.

I have several criticisms of the book:

- Despite claims of aiming at a particular audience, the book seriously lacks focus. Several parts seem aimed at complete beginners, other parts look like reference summaries for people experienced in C++.
- The book contains far too much C and code that is C-like for a book on C++.
- Many of the code examples and exercises are presented in an order that seems concerned with language feature coverage. This causes many of them to require knowledge of material that is presented later (without any forward referencing).
- The chapter and appendices division seems odd, almost arbitrary. There are entire chapters devoted to the old C way of doing things, e.g. the POSIX **`datetime`** functions and C I/O using **`printf`**/**`scanf`** which I would not even have mentioned or would have put in an appendix. The chapters even occur before the better C++ way of doing the job (e.g. C strings are discussed before the STL string class). The chapters are disparate and there is nothing to link them together. Several of them could have been presented in any order (with the C stuff in appendices). This has led to a large number of chapters (20 in all).
- The exercises vary in difficulty to an amazing degree which makes me think that the author does not seriously anticipate that anyone will actually do them.
- The book was very loose with terminology. It calls exceptions 'Structured exception handling', it refers to lambdas as 'lambda technology', it describes **`#include`** as a labour-saving device, it conflates exceptions with miscellaneous runtime errors from the language or its **`std`** libraries, it refers to associative containers as 'associated containers', it uses preprocessor macros where modern C++ would use **`enum`**s or **`const`** variables, exceptions are not caught by reference, it says that function prototyping is optional.
- There are inaccuracies:
  The description of vector reserve says that the specified amount of space is reserved; it actually requests that the vector capacity be at least enough to contain *n* elements so in theory it could allocate more. The **`std`** header to include to get **`std::pair`** is **`<utility>`**, not **`<map>`**. The vector template discussion says that **`vector<bool>`** has been deprecated. Strictly speaking this is not true. There have been several attempts to deprecate it and this is ongoing so it is best avoided for that reason and for the reasons that are connected with why deprecation is planned (which are not mentioned in the book).
- Early on there is a section entitled 'Dealing with the flashing console'. This is to do with what happens when one is using Windows and launching from the IDE instead of using a long-running DOS prompt and invoking the program from that. The program will terminate and take the transient DOS window with it causing a window that briefly flashes on the screen. A non-portable 'solution' is given which is almost immediately retracted and followed with a portable 'solution'. In my opinion it would be much better to say that one is advised to say **`cin.ignore();`** to prevent the DOS prompt launched from the IDE from immediately terminating so that you can see any console output that may have been generated.
- Some aspects of C++11 were not covered, e.g. attributes, defaulting and deleting special member functions, emplace functions, **`Array<>`**, **`forward_list<>`**, **`noexcept`**, **`datetime`**, **`date`**, **`timepoint`**, duration, clocks, threads, synchronisation mechanisms. Now, I realise that it was not the intention to cover every feature but I think that there should have been something at the beginning to mention which features would be omitted. I think it was a mistake to exclude the C++11 date time facilities whilst having a chapter that discusses the old C way of doing things.



On the plus side, the book is very easy to read. The style is slightly informal but not too over-familiar. The explanations and diagrams are quite good. The reference-like sections are more readable than they appear at first glance because they each contain a small example. These examples are very helpful and typically missing from similar reference sections in other books. The book is also quite short, especially considering how much is covered. Perhaps this is why it is called *C++ for the Impatient*. It does seem to be making an effort to impart the knowledge without extensive preamble and background explanations, just enough for practical purposes. The book has received glowing reviews from other reviewers on various web sites and this easy reading was one of the main reasons. The book is also produced to a high standard with a lay flat binding, high quality paper and high quality fonts.

Despite these good points, in my opinion the sheer number of bad points outweigh the good and result in a book which I cannot recommend.

## View from the Chair in Waiting

**Alan Lenton**
**chair@accu.org**

Well, my hopes of being forced to fight an election for the post of chair were dashed when the only person considering standing against me had to pull out due to a change in his personal circumstances. You can still vote for me, and for Malcolm Noyes as the secretary, online, or, if you want, at the Special General Meeting on 2nd August 2014. If you haven't already received the details in your email by the time *CVu* is published, you should be receiving them shortly.

Obviously, ending up with no chair or secretary was pretty embarrassing. So, the committee will be looking into the possibility of taking nominations for officers at the AGM, in the event that there are no nominations before the AGM. We will have to change the Constitution to do that, so if we go down that route, the decision will be at next year's AGM.

There's not a lot to tell you about, because, without a full complement of officers, the committee is only allowed to keep things ticking over, which it has done. Obviously the committee is concerned about the steady trickle of membership losses, and resolving this will be one of the major items on our agenda this coming year. One of the things we will definitely be doing is getting draft committee and general meeting minutes up in the web site's members' area as early as possible, rather than waiting for the minutes to be agreed before publishing them.

On a more personal note, it seems to me that over the next couple of years ACCU is going to reach the point where it is going to have to make a decision about what it really is, and what it wants to be. Arguably it has suffered since it decided not to concentrate on C/C++ – although it never did just concentrate on C/C++. I suspect the decline in membership is related to its lack of clarity on this front. In some ways we are still living off our heritage. The things we are best known for are the Conference and our publications – both launched a long time ago by Francis Glassborow when he was the chair of ACCU.

We have many very clever people who are members. The problem is, though, that they have many other calls on their time – work, family, their own projects, to name just the most obvious. Nonetheless, we need the input of members if we are going to start growing again. The committee can facilitate that, but it can't substitute for the membership.

There are good signs on the horizon – in particular the slow but steady growth of local groups. The committee is already looking at ways to help out, and to provide resources advertising ACCU membership to those at the meetings.

But enough of the heavy stuff! I also have a couple of appeals to make. We need some one with publicity experience to help out on the committee in this field. I realize that being an ace spin doctor is not one of the skills normally associated with computer programming, but I'm sure there must be somebody out there with the right combination!

Secondly, if anyone has any comments or ideas about how we can improve the website then the committee would like to hear from you. Rest assured we will take comments on board, and we promise we won't try to intimidate you into implementing the idea for us...

In the mean time, happy programming, and may all your bugs be small ones. :)