The magazine of the ACCU

www.accu.org

{cvu}

Volume 18 Issue 4 August 2006 £3

Features

Regulars

{cvu}

Volume 18 Issue 4
August 2006
ISSN 1354-3164
www.accu.org

Editor
Paul Johnson
77 Station Road, Haydock,
St.Helens, Merseyside,
WA11 0JL
cvu@accu.org

Contributors
David Carter-Hitchin, Derek Jones, Lois Goldthwaite, Pete Goodliffe, Thomas Guest, Bill Rubin, Phran Ryder

ACCU Chair
Jez Higgins
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
David Hodge
membership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe and Alison Peck

accu

Software Engineering – the Greenest of the Professions

I was looking at some code of a friend a few weeks back (shortly before being hit for 6 by the 'flu) and I knew that I'd seen it before. It looked incredibly familiar. Too familiar. Then I realised that the code I was looking at was code that I'd written about 9 months back. He'd simply transplanted it into his work. Okay, not a problem, the code was released under the GPL, so I'm not worried (the copyright attribution had not been changed).

With that in mind, I started to look at some of my own code to see if this was a common occurance. Guess what – it is. Then I really started thinking about the implications of code reuse and how wide spread it was. First the implications.

On the positive side, you know what the code does, how it works, how to employ it best and how much abuse you can give it before the segment falls over kicking and screaming. The negative is that if the code is broken or insecure or causes a buffer overflow which had never happened before, then not only the current project is shot, but also all other projects which rely on your code. Now think about that last line. It's important.

With the increase in open source uptake and the number of large companies giving code freely (such as Sun, IBM and Novell), the chances of broken code being reused and therefore becoming more dangerous becomes a possibility and do you know what? I'm made up that this is the case! The more poor code there is out there, the more conscientious code authors should become. There is a critical point in every industry that says "okay, hold it there. This is broken. And so is this. And this. Crumbs, we'd better start to be more careful". Pigs may fly? No, it's already started. I've noticed in many projects on Sourceforge and Freshmeat that the code base is becoming far more particular with code reviews and what can be submitted as patches. It's good. I just wish the same transparency was available in the commercial world – imagine the fun and games that could be had if the source to big applications was made available!

PAUL JOHNSON,
EDITOR

The official magazine of the ACCU

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

The 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 ACCU members – by programmers, for programmers – and have been contributed free of charge.

To find out more about the 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

FEATURES

REGULARS

COPY DATES

C Vu 18.5: 1st September 2006
C Vu 18.6: 1st November 2006

IN OVERLOAD

Overload 74 contains high-quality articles on a wide range of topics, including The Kohonen Neural Network, C# Generics - Beyond Containers of T and Fine Tuning Boost's lexical_cast. Overload is available to all full ACCU members.

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

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

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

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

Coaching – the Art of Not Teaching

Phran Ryder explains the concepts behind coaching.

Mumbling...

"Many Agile methods emphasise the important role of the coach without going into detail about what it is to be a coach."

"Dad."

"Coaches use questions to raise **awareness**, ensure **ownership**,"

"*Dad*!"

"If you really want to be a good coach I think you need to experience coaching on a coaching course."

"DAD! - You're talking out loud again!"

"Mm, Oh. What son?"

"You're talking out loud - I can't do my homework with you mumbling to yourself."

"Oh – yeh. Sorry. How are you getting on with your homework?"

"Great. Nearly finished and I bet I get full marks this time. How about you?"

"I am trying to write an article about coaching for ACCU"

"Bless you"

I looked up at Jake, saw his smile and realised he had made a joke.

"ACCU stands for the Association of C and C++ Users."

Jake gave me one of his best teenage 'like I care' expressions.

"Well it worked on me, didn't it, Dad. I mean the coaching."

"In a way it did but really it just helped you to work on yourself."

The issue

I was sitting with my son Jake at the kitchen table.

This is how I now spent an hour or so of many weekday evenings. Jake would do his homework and I would, well, I'd try to make use of the time somehow. And generally I did. It was a cosy arrangement but certainly not a work environment I would have wanted for myself. You see, Jake had been having trouble doing homework. No, that's not correct, he had been getting into trouble for not doing homework. That's not accurate either, he was at the age where, where, …, well he was at that sort of age.

After discussion with Jake's Mother, I had agreed to come up with a way to get Jake on top of his homework. My job is one in which I spend great parts of the day helping people to become motivated, helping them to overcome problems, and helping them to achieve greater, well, achievements. So the natural thing was to translate these skills to home.

The session

I had been in communication with the school and discovered their concern about Jake's homework. The problem was how to broach the subject with Jake. Jake was of an age where he could be surly, confrontational, or uncommunicative. But he could be lucid and friendly. Many parents might have opening gambits like:

"I need to talk to you about your homework."

or

"What are we going to do about your homework?"

or

"The school is not happy with your homework"

The coaching approach that I use is all about helping the coachee to understand *their* issues, *their* goals and how *they* can reach them. The key tool is the use of the **open question** – a question that invites a response of plenty of words.

The coaching follows a structure known as the GROW model which stands for:

G – Goal

R – Reality

O – Options

W – Will

In summary this involves establishing a **goal**. Looking at the **reality** – the current situation. Exploring the alternatives **options** for reaching the goal. And finally understanding how committed the coachee is to overcoming the any barriers to reaching the goal – their **will**.

Coaching courses teach you a few things you should do as a coach. One is make the coachee aware of the coaching process. Another is that the coachee chooses the topic of the coaching session. This is because coaching largely about raising the coachee's **awareness** of the current situation and of the coachee's **ownership** of steps to change the current situation. However, surly teenagers, and many others, are unlikely to appreciate or care for the intricacies of coaching. In such situations I use coaching by stealth. I use the coaching techniques but the coachee is not aware that it is going on – at least not at first.

(G) The Goal

In tricky situations the first questions are important, so after some thought I went for my opening gambit.

"What would you like to get for your next homework?"

This question is not entirely open. It invites a short response but it does help to establish a goal. Thankfully, Jake took it in a positive way.

"Full marks – of course!"

So that's the goal. Now for a little reality.

(R) The Reality

This is where I, as a coach, help Jake to understand the current situation, that is: what has been achieved, what has been tried, who is involved.

I asked "What did you get for your last few pieces of homework?"

He told me – it was not an impressive collection. It included zeros for homework not handed in and averaged about 30%.

"So your average is 30% and you aim to get 100% for your next homework. Is that achievable?"

Jake shrugged the shrug of the bothered, "Probably not"

"What mark do you think is achievable?"

"I don't know! 70%?"

During these last few brief questions I have been refining and qualifying the goal. As I am listening to Jake's response, I do not at any point think about what my next question will be, I just **listen**. Only when Jake has finished speaking will I consider what has been said. I'll **summarise** what

PHRAN RYDER

Phran is chairman of AgileNorth.co.uk – a non-profit organisation for technical and business staff who wish to learn and share experience of becoming and being agile – details at www.agilenorth.org.uk. He aspires to be a code god and can be reached at Phran.Ryder@lloydstsb.co.uk

has been said and only then consider the next question. It is a simple cycle: listen, summarise, ask.

I summarized: "So for your next homework you want to improve on your average of 30% and get 70%?"

"Yes"

So we have considered the reality and as a consequence adjusted the goal.

(O) The Options

In my coaching session with Jake, the initial exploration of the goal and reality did not take long. This is not always the case. Neither is it the case that exploration of goal, reality, options and will is sequential. In practice, the questions move freely between the types but there is a general trend from goal through to will.

The 'could' word appears in many options questions. "What could you do to get more than 30%?"

Jake contemplated briefly and answered "I could work harder."

"What else could you do?"

"Cheat"

coaching is more than asking questions and listening

I couldn't help smirking at this point. A little light relief is a good thing. I chose to say nothing in the hope that more options would be forthcoming. Sure enough he made a suggestion and after a few prompting questions we had several options. The next aim is to prioritise these options. I summarized the options and asked:

"Which is most likely to help you improve on your marks?"

Jake considered the options briefly and answered, "Working harder"

Working harder is a bit vague. I wanted Jake to consider exactly what he means. "What is involved in working harder?"

Jake went quiet. He was clearly thinking, searching for ideas. It was very tempting to come up with some suggestions myself. After all I have a fountain of experience – I know lots of things. But making suggestions is one of things that coaches are encouraged to avoid. In this approach to coaching the goal, options and actions have to be **owned** by the coachee. They must come from within the coachee. That way they are more likely, indeed very likely to be followed. I waited and Jake responded.

"Spend more time on my homework?"

On the surface this answer seems like a good option. It is easy to measure and thus be specific in setting targets. But what does it mean to Jake. I asked,

"How would that help to improve your marks?"

Jakes expression was initially blank but then became bemused and frustrated. He had no answer and said so. So I took a different tack. "What prevents you from working hard?"

Jakes quick response "Its boring! I'd much rather watch TV." came with conviction.

Fair enough I thought. I thought of asking what could make it less boring but remembered to phrase my question positively. "What would make it more interesting?"

"Having the TV on?"

This was, to me, probably poor choice. But it is not really my place, as a coach, to **judge**. So, of course I asked for more options. "What else could make it more interesting?"

"Having the radio on?"

"Anything else?"

There was a pause. Jake seemed to be thinking. The pause continued, became pregnant, and gave birth. During moments of coaching silence it is very tempting to fill the gap, to rephrase the question, to ask another question, or, Q forbid, suggest an answer. But coaching is more than asking questions and listening. Good coaches use all senses available to them and this time my senses drew my attention to his body language. I felt he had an answer waiting to materialize; he just needed to narrow his angular confinement.

"I don't like being on my own?"

I could feel his relief at having said this, it clearly meant a lot to him. His answer wasn't really an option so I **rephrased** it to **clarify**. "So you would like some company when you are doing homework?"

"Yes"

We continued for a few minutes and came up with more options. I summarized these and then asked "Which of these would be most likely to make doing the homework more interesting?"

The answer came shyly but quickly – "I'd like some company"

"Who would you like to have for company?"

Jake looked up. Looked me in the eye and said – meekly, "You"

"Well." I said "I'll ask me but I am sure I will do it."

Jake was so obviously pleased that it knocked me off my stride. What does this mean? Am I not spending enough time with my son? Am I neglecting him? I put these thoughts to one side and continued with the coaching process. I used questions to take Jake through some of the other options and he chose some others that he could try.

(W) The Will

I then started to consider Jake's will. This looks at things that might prevent Jake reaching his goal. "What might prevent you from doing your homework with me?"

"You might be too busy"

"How can you make sure I am not too busy?"

"I could let you know when I am going to do my homework"

"What if that doesn't work?"

With a cheeky smile he came with a threatening reply "I'll give your PDA a bath."

"What else could you do if I am not available?"

"I could listen to the radio or I could see if mum will or…"

During the proceeding minutes Jake was clearly enjoying coming up with ideas and stating how he would overcome barriers. My questions where directed at making sure Jake had a good idea of how he **will** reach his goal. This involves, again, exploring the reality, considering options and deciding which he would use in a number of situations. But at no point did I make a suggestion or give direction. It had all been Jake's thinking.

The summary

So here I was sitting with Jake, mixing idle banter and work. And I was enjoying it. And Jake's homework was being done and being done very well.

I returned to the cause of my mumbling. My essay on coaching – I typed my summarizing paragraph.

Many Agile methods emphasise the important role of the coach without going into detail about what it is to be a coach. This article does not claim to provide that. There are a numerous angles to being a coach. This article looks at just one of them. It attempts to illustrate a style of coaching that should form a major part of a coach's box of tools.

The technique uses **open questions** that are structured around the **GROW model**. Coaches use questions to raise **awareness**, ensure **ownership**, augment **commitment**. During the process they avoid making judgements, offering suggestions or leading towards a particular solution. Instead they **paraphrase** to clarify, **summarise** for review and **listen**.

Effective Version Control #1

Pete Goodliffe describes how to manage your source code simply and effectively.

Version control is a necessary evil, the prerequisite for a robust development process. We all know that we should do it. Sadly many development shops get this foundational act wrong. Version control isn't rocket science, but it does require some careful thought and planning, and more than a little discipline.

This series of articles will lead you down the path to *effective* version control. Following a distilled form of Meyer's 'Effective' book structure, I present a series of items with simple headlines followed by deeper discussion. Read this as it suits you: save it for later and pick out the bits you're interested in, or start from the beginning and work to the end.

I'm aiming at both software developers and version control administrators – I present rules for kings and knaves. We all know that rules exist to be broken, but the following items describe version control best practices. You'd better have a good (and justifiable) reason to avoid any of them. If you're currently implementing all of these items then well done – you're an above average version controller already!

This is an enormous topic, and there's a lot to say. Even though I present this as a series of articles rather than a single behemoth (I think you'd struggle to read that monster about as much as I'd struggle to write it!) we really only have time to skim the surface of version control. Large books could (and have) been written about this topic. Consider this a whirlwind tour. Caveat lector.

Basic principles

These first few items are the most fundamental – the groundwork upon which our version control story will build. You might imagine they're the most obvious, but some of these are quite often missing in development team cultures.

1. Use version control

Version control is not optional. It is not a nice-to-have. It's not something to add when you 'get around to it'. Version control is the backbone of your development process; without it you lack structural support. Version control is the mechanism by which multiple programmers collaborate on the same codebase, and your source code time machine. It allows you to see how the code changed over time, who made a particular modification, and why [1].

Without version control, your development efforts are at risk. You have no central backup, and no definitive 'master' source tree. You're left at the mercy of computer/hard disk failure, or the accidental deletion of those vital source files. You could try to do the same job manually, but humans make mistakes and you'll inevitably land in hot water. Why do by hand what others have helped you do automatically? (See item 21 for more of this kind of philosophy.)

As you can see, it's vitally important to use version control. You should use it from the very start of a project, even for small projects (small projects grow, and soon become unstoppable software organisms). You might not think that the code you're writing is 'important enough' for version control, but it is always worth the effort. A source code repository provides a structured working process, helps you correct mistakes, manages software release versions, and provides defined single point of backup – a safety net for even the least critical projects.

2. Deploy version control well

So you definitely need to install version control, and you need to deploy it *right*. What does this mean? It means several things, each of which we'll look at in this item. First of all: you need to *use the right tool*. This one

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@cthree.org

Coaching – the Art of Not Teaching (continued)

I have taught, coached, and mentored for many years. I am sure I am good at it. But having been on a coaching course I am painfully aware of how often I tell, i.e. hand out a solution and hope it would be understood.

This may sound simple but if you really want to be a good coach you could read about it – try Coaching for Performance ISBN 1 85788 303-9. But I think you need to experience coaching on a coaching course. See for example http://www.performanceconsultants.co.uk.

The punchline

"Dad, what is the connective noun for birds?"

"Excuse me?"

"The connective noun, you know like a herd of cows"

"You mean the collective noun?"

"Yeh"

"Oh, it's a ... flock."

"Dad, you're a coach... what is the collective noun for what you do?"

"I don't think there is one for coaches.

"What about a train of coaches?"

"Nice one."

"Dad, your business card says you are a consultant. What is the collective noun for consultants?"

"I don't think there is one"

"Shall we make one up?"

"How about a shaft."

"Why a shaft?"

"Well, that is what they generally do to people and it is where they should be kept – in a bottomless one."

"Dad you're a consultant."

"I prefer the term coach." ■

What is version control?

Before we get too far in this foray into version control, it's important to understand exactly what we're talking about – by defining a few key terms. Whilst this knowledge is essentially a prerequisite for the reader, it will be still useful to quickly review the basics.

Version control – The process of managing multiple revisions of a set of files. These are commonly the source files for a software system (so it is often called *source control*), but it could just as easily be revisions of a documentation tree, or of anything else you'd work on in a file system.

Repository – The version control server's central store of the versioned files. The repository records every revision of each file it manages, allowing you to revert to a particular version, or to investigate how the file looked at a specific date.

Working copy – Sometimes known as a *sandbox*, this is your copy of the source tree, checked-out from the repository and placed on your computer's local hard disk. You work on the files here, in isolation from anyone else working at the same time.

Check-out – The act of creating a working copy from a repository image. Some version control systems also require you to specifically 'check out' each file you want to open for editing, providing a form of edit contention control through file reservation.

Check-in – Or *committing*, the act of sending your new version of a particular file (or set of files) back to the server for inclusion in the repository.

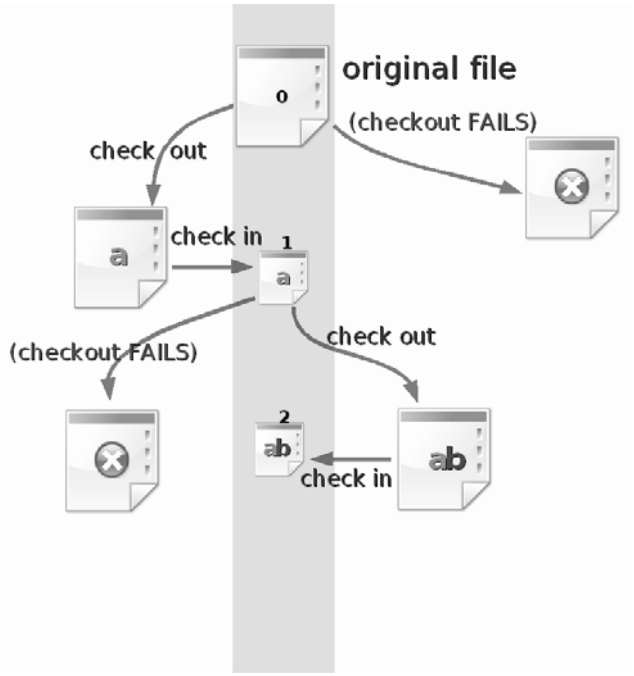
Branching and merging – A *branch* is a parallel copy of a set of files in a repository, acting as a 'fork' in development. You can create a branch to work on a feature in isolation, without affecting the main development effort. Once you've completed work on the branch, you can *merge* it down to the mainline development branch, or stop working and leave it as a development dead-end. The main code branch is often called the *trunk* (for obvious reasons).

Tag – A symbolic name associated with a set of file revisions. Used to mark important development milestones in the repository (for example: each major software release version). We'll look at branching and tagging in a later article.

Configuration management – Not to be confused with version control, software *configuration management* is a set of procedural and management processes to ensure that software is delivered correctly, on time, to budget, with minimum issues, in a repeatable and reliable way. CM certainly includes version control as one of its central tenets, but it is far, far, more. This article focuses solely on version control. ('Effective configuration management' could fill several books!)

VCS – Short for *Version Control System*. You might also see *SCMS* which stands for *Source Code Management System*.

Figure 1



simple choice opens up the religious world of version control advocacy. Don't ask anyone's else's opinion, or you might loose yourself in a So you definitely need to install version control, and you need to deploy it *right*. What does this mean? It means several things, each of which we'll look at in this item. First of all: you need to *use the right tool*. This one simple choice opens up the religious world of version control advocacy. Don't ask anyone else's opinion, or you might lose yourself in a heated VCS argument that lasts longer than your next development project.

How do you pick the right version control tool? There are many tools out there. They cover a vast range of maturity, scalability, features, modes of operation, and more. They range from the simpler (single server) systems like Subversion, CVS, and Perforce, through 'distributed' systems like BitKeeper, and Arch (with their multi-site hosted repositories and complex source tree models), to the enterprise-capable systems like ClearCase (with all the complexity and baggage enterprise deployment brings). Choosing is not a simple or obvious task. Good luck!

I guess you'd like a little help here? You've got to make a pragmatic choice based on your project needs. But first, let me introduce you to the golden rule of version control. It'll crop up many times, and is applicable in more fields than this one. Here it comes, so pay attention:

Simplicity is a virtue. When you have a choice, make the simplest one you can right now that will not adversely affect your choices later down the line.

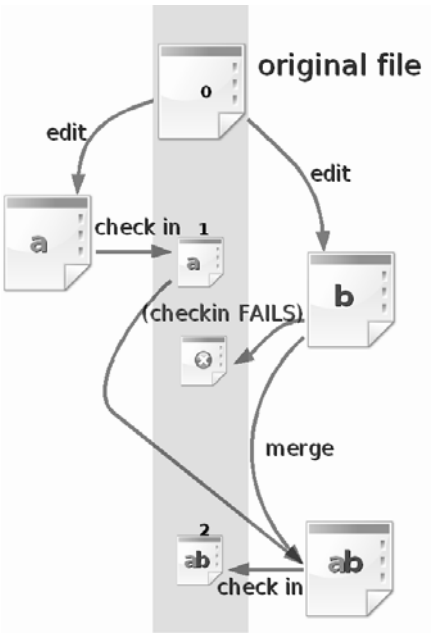
There, it's not complicated, is it? All sorts of version control problems can be neatly dealt with using this golden rule. Applying it to your choice of VCS tells you not to plan for the case when your company IPOs, gains 300% more developers and requires a vastly larger version control infrastructure. If right now you have five developers (and a pot plant) and you'll not grow beyond ten programmers (and two aspidistras) for the foreseeable future then base your tool decision on this information.

The more complicated tools offer many features. They not only come with a much higher price tag (which shouldn't be the deciding factor in your choice of VCS, but in some cases – especially when working on projects at home – might be important), they come with a swathe more complexity. They have higher admin overhead, a steeper learning curve, and there's more to go wrong. Unless you provably need that extra overhead, steer well clear, and select a more suitable – simple – tool.

It's beyond the scope of this article to run a feature comparison of the version control tools available, but there are a couple of notable systems that bear some mention. First of all: *CVS*. This is a venerable, well-proven VCS that is used worldwide for open source development, and is relied upon by many, many commercial projects. It works. It's simple. People know how to use it. It integrates into all good IDEs, and there are many third party CVS extensions around. It comes highly recommended [2]. However, CVS is beginning to show its age. It lacks atomic commits (guaranteed transactional server behaviour) and you can't rename files or move things around easily.

Subversion is a relative newcomer, and was designed to be CVS's successor. It fulfils this goal admirably, and provides the CVS-like development model with a more modern architecture and with excellent scalability. I have deployed Subversion on several sites and always been very satisfied with its performance.

Figure 2



I suggest that if you can't make a decision on any other grounds, start looking at Subversion and require any other tool to provide a compelling reason to move you away from it.

Now you've chosen a VCS, consider these other deployment issues:

- Only use one version control system per project. Ideally use only one version control system per company; it cuts down on complexity and confusion, and helps you to share code across projects. You'll also enjoy administration economies of scale. To achieve this you might want to migrate some of your source code out of a legacy VCS into the shiny new system. We'll look at that in item 11.
- Don't configure the system unusually. Use the simplest deployment possible. Don't overcomplicate VCS use with baroque processes or complex check-in strategies. When you deploy VCS, match it as closely as possible to the current development process of your organisation, with an eye to future best practices (unless, of course, you're working totally ad-hoc).
- Establish the right working practices (see the rest of this article!), and determine how version control fits into your overall configuration management strategy. Remember that version control alone will guarantee nothing. Alongside coding standards, a culture of solid units tests, and a managed software release procedure (see item 20) your work will be of an excellent standard.
- Address hardware concerns. Put the VCS on a scalable server with enough disk capacity, with some room for growth (in users and code size), with a reasonable plan for extensibility, and with an adequate network infrastructure to support day-to-day use.
- Document your version control strategy. Provide a five minute primer that describes your tool set, where the repositories are hosted, and how to get up and running with them. Provide pointers to other VCS manuals. Make sure that your development processes are clear.

3. Know how to use your version control tool

Having version control is essential. Selecting the right system is paramount. But once you've got a VCS, knowing how to use it properly is a prerequisite. I've seen more version control disasters caused by inexperienced developers trying to do something clever than by mechanical failure or VCS bugs put together. (It's remarkable how often these human errors are blamed on system problems.)

The law here is simple: don't do anything unless you know *what* you're doing, and exactly *how* to do it. Even if you do know how to perform some complex task, always take a practice run first (try it out on a test repository, or run the commands in 'test mode' to see what will happen).

Learn how to use your version control tool well. You don't have to necessarily know it inside-out (although that level of knowledge will never hurt you). But you must know the 50% you routinely use. Never use the tool without switching on your brain first.

Locking Models

One of the fundamental differences 'twixt version control systems is their file locking model. This profoundly affects one of the most basic developer activities: editing files. There are two models:

Locking (or *strict locking*) systems mark all the files in your working copy as read-only. You can't edit them. In order to fiddle with a file you have to ask the VCS nicely for permission to edit. This is known as *checking out* a file. The server marshals these requests and prevents two developers from opening the same file in the same branch at the same time.

As shown in Figure 1, this prevents you from making changes that might interfere with another person's work. It does, however, create a large procedural hurdle for even the smallest edits. It can also be intensely frustrating: if Dave checks out `foo.c`, and then goes into a meeting for three hours – or worse, on holiday for a month – you're left tapping your foot, waiting patiently until you can edit it.

Concurrent (or *optimistic locking*) systems, on the other hand, free you from this locking hurdle by managing concurrent development for you. Your working copy is perfectly modifiable, and you can change it at will. The VCS sorts out the consequences of developers working on the same files.

Usually this is perfectly harmless; if two developers modify different parts of the same then file their changes get automatically merged together. Magic. However, if they both modify the same lines of code, the VCS will moan when the second developer attempts to check-in, marking their copy as *conflicted*. The developer has to resolve the conflicting changes and then attempt another check-in. This is a lot less painful than it sounds, and seldom happens in practice. This is shown in Figure 2.

This is a good advert for choosing a simple tool. You're more likely to avoid accidents by not having complex VCS behaviour to understand, and any problems you do encounter will be easier to recover from.

Stay tuned...

In the next instalment we'll look at managing our code repositories and how to track third party code, before investigating how to work with version controlled code. We'll conclude by looking at tags and branches, and how to make reliable source code releases. ■

Notes

1. This is more than software archaeology. It'll help you retrieve the exact code that built a 5-year-old release version of your product, which now contains a critical fault.
2. CVS also has good migration paths to other tools. It's the de-facto VCS, and every other vendor wants CVS users to move to their tool set. So many CVS repository conversion tools exist. You can start here and get more adventurous later.

Pete's book, Code Craft, is out soon.

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- -What are you doing right now?
- -What technology are you using?
- -What did you just explain to someone?
- -What techniques and idioms are you using?

If seeing your name in print wasn't enough, every year we award prizes for the best published article in C Vu, in Overload, and by a newcomer.

Gymnastics revisited

Last issue's series of brain teasers and gratuitous asides set a lot of you thinking. Thanks to everyone who replied! I promised to put the best answers in print, and true to my word, here they are...

The winners and also-rans

Tim Penhey deserves credit for getting the number series right, although he loses points because originally thought that I'd made a mistake in the puzzle. **Thomas Hawtin** also got the number series right. The winner for this puzzle, though, was **Anthony Williams**, who managed to send me the correct answer before anyone else. Just. Buy yourself a drink to celebrate, Anthony!

Anthony's 'Spot the difference' answer was good, but not the one I was looking for: *The Smalltalk sample is the odd one out, because all the others are idiomatic in their specified language for starting an iteration. Iteration in Smalltalk would use something like collect: or inject: into:, a bit like the Ruby version.*

Paul Smith fell into my obvious trap in the number puzzle, but deserves credit for his honest 'Spot the difference' answer: *Can I find all the differences? No.* That's probably the best answer that anyone gave! He also got the correct 'Odd one out' answer: *I guess Ruby (not having any knowledge of Ruby at all...) because it looks as if the others just create an* `iterator` *and initialise it to refer to the first value of* `stuff`*, but the Ruby one calls the contained block for each item in* `stuff` *as well as making* `i` *an iterator.* Spot on, Paul!

Jon Jagger guessed C was the odd language out as it was the only listing that didn't mention `stuff`. Close, Jon, but no cigar.

Reece Dun wins the Eager Beaver award for getting his answers in first (by a margin of less than an hour). It took him a run-up to get an answer to my number puzzle, but unfortunately, he didn't get the series quite right. His other answers were excellent, though, including a very comprehensive 'Spot the difference' answer:

The first example is C++ and the second is the same code in C.

1. Include the C++ (`<iostream>`) or C (`<stdio.h>`) output header.
2. C++ defines swap in `<algorithm>` whereas you have to write your own in C:
   a) C doesn't have templates, so you have to write each permutation of swap as needed, by hand;
   b C doesn't have references, so you need to pass the values by pointer if you want to modify their content.
3. The interface to `bubble_sort` follows the languages paradigms:
   a) C++ bubble sort operates on a `[first, last)` iterator pair, whereas C uses array, length. NOTE, the C version can be expressed as `[array, array + length)`.
   b C++ bubble sort can operate on any iterator type (i.e. any underlying container, and even subrange), whereas the C version is restricted to C-style arrays.
   c C++ bubble sort can operate on any data type that the underlying sequence provides, whereas the C version is restricted to integers.
   d C++ bubble sort is vulnerable to 2 programming errors:

```
bubble_sort( c.end(), c.begin());
// wrong way around
bubble_sort( c.begin(), d.end());
// different containers!
```

   but modern C++ libraries (e.g. GCC and VC8) can detect these in debug mode. C bubble sort is open to buffer overrun if used incorrectly:

```
int data[5] = { 5, 3, 7, 4, 2 };
bubble_sort( data, 10 ); // oops! too big!
```

4. The first C++ for loops follow STL best practices, e.g.:

```
for( first = begin(), last = end(); first != last; ++first )
```

   because you should always use `!=` to compare iterators. This is because you may be operating on objects and not pointers, whereas with the C version, you are operating on pointers, so `<` is used to compare them.
5. In the second loop, the C++ version uses `j < i` to compare iterators; I assume this is a mistake on the programmers behalf, and they intended to write `j != i` instead.

   NOTE: Using `j < i` **is** valid, but (in this case) unnecessarily restricts the C++ `bubble_sort` to random access iterators. In the next version of C++ (C++0x), there will be support for concepts, allowing something like:

```
template< typename Iterator >
  where ForwardIterable< Iterator >
void bubble_sort( Iterator first, Iterator last ){
... }
```

   so this bug (if indeed it is a bug) can be picked up immediately by the compiler.
6. Another difference with the for loops in the C++ and C versions is that the C versions loop over the indices into the array, whereas the C++ version iterates over the elements in the sequence.
7. When comparing the two values in the inner if statement, the C++ version dereferences the iterators to get at the values, whereas the C version accesses the index into the array using array[i].

   NOTE: `array[i]` is equivalent to `*( array + i )`, so the C++ version `(*i)` will be faster as it requires less computations.
8. When swapping the values, the C version uses the `swap` that was implemented,whereas the C++ one uses a special `std::iter_swap` method. Unlike `std::swap` (which would swap the iterator pointer values!), `std::iter_swap` will swap the values that the iterators point to: just like the hand-written C method.

   NOTE: If you are performing a `bubble_sort` on `std::list< std::string >` with the C++ version, the library implementer could have provided a version of `iter_swap` that will avoid temporary copying of the strings, using `std::swap` on them, e.g.:

```
template< typename Iterator >
void iter_swap( Iterator a, Iterator b )
{std::swap( *a, *b );}
```

   meaning that you get efficiency gains while maintaining the flexibility and readability of using the standard library. The Dave Abrahams and Alex Gutovoy book on C++ template meta programming has a section on the possible optimisations that can be made to the `iter_swap` function.

Reece also tried far too hard in 'Follow the leader': *Bjarne Stroustrup wrote a program in C++ that, while it made it harder to shoot himself in the foot, caused a* ***core dump*** *resulting in the machine blowing up, taking his leg in the process ;). Scott Meyers forgot his Effective C++ lessons when he wrote some code that resulted in an* ***infinite loop****. Herb Sutter earned* ***eternal fame and glory*** *by producing C++/CLI, the equivalent of fitting a square peg (C++) into a round hole (CLI) and (for the most part) succeeding! Andrei Alexandrescu devised a new policy for his smart pointer class – RandomErrorOfTheDay – that resulted in a* ***segfault****.* Thanks, Reece!

Answers

For the record, the correct answers to each puzzle are:

Number puzzle: By the end of `main` n=7. The sequence was 4,7,3,4,2,8,12,1,2,16,19,1 (this number lead to the obvious trap – did you fall foul of it?),5,3,0,1,17,8,9,1,6,7.

Spot the difference: There are far, far too many differences to describe here, but they are all eclipsed by one simple fact. The code on the left worked, and the code on the right didn't! I can't believe that no one spotted that!

Follow the leader: Herb won. Thankfully no one got that puzzle wrong. Even my daughter managed it.

Odd one out: Each of those code snippets idiomatically created an "iterator" in their language. But the Ruby code was the only snippet that would have attempted to perform an operation on the iterated range.

The Structure and Interpretation of Computer Programs Mentored Project

Thomas Guest writes a progress report for this project.

To date, half-a-dozen students, give-or-take, have been working steadily through the exercises in Abelson and Sussman's book *Structure and Interpretation of Computer Programs* [1] (SICP).

You may not be aware of the mentored developer projects, or if you are, you may be interested to learn a little more about how they operate. You may be curious to find out why anyone interested in studying computer programming would want to use Scheme, a dialect of Lisp—an old and notoriously uncompromising programming language. This article cannot provide definitive answers to these questions: for answers, you'll have to visit websites, digest emails, watch lectures, work through exercises, etc. Instead, it offers pointers to more information and an invitation: *It's still not too late to join in*.

Let me also mention at the outset that Mike Small leads this project. I'm just a student who has volunteered to write a report.

Mentored developer projects

To find out more about ACCU Mentored Developers' projects, visit the Members' Section of the ACCU website. Projects are typically set up by a group of ACCU members who wish to study a book, a technology, or similar. Each project will have a leader, who kicks things off, assigns tasks, and oversees the schedule; students, who actively work on the project; mentors, who provide expert advice; and observers, who are free to contribute as little or as much as they please.

Activity takes place on email lists. The accu-mentored-developers mailing list is a low-volume list mainly used to announce when new mentored projects start up—if you haven't subscribed, I recommend you do. When a new mentored project starts, it spawns off its own separate email list. This will be a high volume list, on which project work and associated discussion actually takes place.

The ACCU SICP project

Mike Small initiated this particular mentored project early in 2006, issuing an invitation on the accu-mentored-developers list.

As already stated, the project was based on Abelson and Sussman's book, *Structure and Interpretation of Computer Programs*—a book which has been used in undergraduate courses in many universities to teach students general principles of programming. Being a text book, it's packed with exercises, and these exercises are what form the basis of our study. The complete text of the book is freely available online, and, what's more, a full set of videos of the lectures can be downloaded.

Schedule

After much discussion and a few iterations, we seem to have settled on a sustainable schedule. Each iteration starts with Mike assigning exercises, one per student per day. As an example, here's the schedule for the last few exercises in Chapter One.

- Thu 23rd March, Tim, Exercise 1.41
- Fri 24th, Pal, 1.42
- Mon 27th, Jan, 1.43
- Tue 28th Mike, 1.44
- Wed 29th Thomas, 1.45
- Thu 30th Martin, 1.46

So, this particular round of exercises starts with Tim submitting his solution to Exercise 1.41 to the email list on Thursday 23rd March. Everyone then comments on this solution. On Friday, Pal's solution to the next exercise gets posted.

This schedule might seem rather ambitious, but, since the book contains over 350 exercises, we need to keep things moving. As you can see, we get the weekends off, and, if someone can't complete an exercise, he just says so and anyone else is free to step in. Besides, many of the exercises are trivial, and nearly all can be answered in just a few lines of code.

A few lines of code

As already mentioned, the book uses Scheme as a language for teaching. Yet Scheme itself is never really taught:

In teaching our material we use a dialect of the programming language Lisp. We never formally teach the language, because we don't have to. We just use it, and students pick it up in a few days.

Amazingly, this turns out to be true. I find myself learning the language by using it. Sure, some of the exercises are there to help me on my way, but most of them are aimed at discovering more fundamental things about programming.

Scheme is also a powerful language. You don't have to type a whole lot, and, by virtue of the interpreted environment, you can reshape and test your code as you go. There's no need for Makefiles or compilation.

Should you require more information on the language, the Standard is available online at http://schemers.org/Documents/Standards/R5RS. Scheme programs are concise and so too is the standard which, at just 50 pages, is more of a pamphlet than a doorstop.

Where are we?

As I write this (April 2006), we're well into Chapter 2 of the book, which starts to talk about layers of abstraction, data structures and so on. Chapter One, "Building Abstractions with Procedures" went into some detail on the Lisp evaluation model, before covering:

- recursive and iterative processes
- the space/time complexity of a program
- functional programming

Recursion and iteration

Scheme does not need any special looping construct [5] since recursive procedures (procedures which call themselves) can be used to implement iterative processes. The classic example is the factorial function, which we might consider implementing using a recursive procedure:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

THOMAS GUEST

Thomas Guest is an enthusiastic and experienced computer programmer. He has developed software for everything from embedded devices to clustered servers. His website can be found at http://www.wordaligned.org

This turns out to be an expensive implementation, since, to evaluate a particular factorial, the Scheme interpreter needs to build a chain of deferred multiplication operations, which can only be applied when it finally reduces the input argument to to the special case value, 1. We have here a recursive procedure which generates a recursive process.

In contrast:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

uses a recursive procedure `iter` to implement an iterative process. At any point in the calculation of a `factorial`, the complete state of the function is held in just three variables, `product`, `counter`, and `n`.

Some points for the curly-bracket language readers.

1. Notice the nested inner function, `iter`, which has been scoped within the `factorial` function since it has no real use outside this scope. Scheme also supports unnamed, or *lambda* functions.
2. I could have used the usual mathematical symbol, `!`, for factorial, if I wanted.

   ```
   (define (! n)
      ... )

   (! 9)
   ;Value: 362880
   ```

   (Note that in Scheme, logical not is written `not`.)

   It appears that pretty much any contiguous combination of non-whitespace symbols can be used to name a function. For example, the built-in increment function is `1+`, which we could happily alias to `++` if we wanted:

   ```
   (define ++ 1+)
   ```

   Predicates often end with a question mark, which reads nicely.

   ```
   (if (even? n) ...)
   ```
3. Numbers aren't constrained to fit into a fixed number of bits, so we can calculate:

   ```
   (! 2006)
   ;Value: 2144794478704779 ... 000000
   ```
4. I was also pleased to find that Scheme builds in support for rational and complex numbers:

   ```
   (+ 7 (/ (sqrt -1) 3))
   ;Value: 7+1/3i
   ```

Space/time complexity

I don't suppose analysis of the space and time requirements of a program will come as anything new to a seasoned programmer, though we may choose to measure rather than analyse. There really is nothing like running an interpreter and directly experiencing the difference between an `O(n)` and an `O(log(n))` implementation of a function to bring the lesson home, though.

Functional programming

The functional style of programming may be a little more novel to curly-bracket programmers. By functional programming, I mean a style of programming where powerful abstractions can be built from higher-order functions—functions which apply to functions, that is—and where functions are first-class objects.

As an example, here is all we need to implement a function which will n-fold smooth another function. Note the higher-order helper functions, `compose`, which forms the composition of two functions, and `repeated`, which repeatedly applies a function. Both these helpers are completely general purpose building blocks.

```
(define (compose f g)
 (lambda (x)
   (f (g x))))

(define (repeated f n)
 (if (= n 1) f
     (compose f (repeated f (- n 1)))))

(define dx 0.1)

(define (average . items)
 (/ (apply + items)
    (length items)))

(define (smooth f)
 (lambda (x)
   (average (f (- x dx))
            (f x)
            (f (+ x dx)))))

(define (n-fold-smooth f n)
 ((repeated smooth n) f))
```

Can I Join in?

Yes—the email list is open to all ACCU members. Observers are always welcome. We'd really like some more mentors. It's still not too late to join in as a student either, and more students would be very welcome. By reading the email list archive you can get an idea of what all this involves, and also catch up with the exercises you've missed.

Why would I want to join in?

I can only provide a personal answer to this question. For me, it has much to do with a growing interest in computer languages, and in Lisp in particular. As Kevlin Henney writes [2]:

Many programmers develop an infatuation with Lisp at least once in their programming lives. If you haven't yet, now is your chance.

I'm quoting Kevlin out of context—he's not talking about the chance to study SICP with a group of like-minded people, he's introducing an article which provides a C++ implementation of Lisp-style list processing. Perhaps this is the conventional reason for studying Lisp: you may never use it in anger (how many job adverts are there for Lisp programmers?), but it will inform you and make you a better programmer. Thus Eric Raymond [4] advises would-be hackers:

Lisp is worth learning for a different reason—the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

Paul Graham [3], though, goes further. His book, *Hackers and Painters*, was what really urged me to find out more about Lisp. In this book, Graham describes Lisp not as a language to be learned as an end in itself, but as a superb language for designing and writing computer programs.

Not all languages are equal

Graham's starting point is that not all languages are equal. Few readers of this article would dispute this, I think, though many of us would qualify such comments with questions about context: *What problem are you trying to solve?*

Lisp is a multi-paradigm language. So too is C++, but the most powerful C++ paradigm, generic programming, is problematic. Generic C++ takes too long to compile, and what's worse, when it won't compile, all too often the compiler spouts nonsense. And often generic C++ looks clumsy: we want to write high level abstractions, but we end up holding down the shift

key to punch in angle brackets, colons, and `#define`s to work around compiler limitations.

When I found out how simple generic code was in Python (in a dynamically-typed language *all* code is generic), and how nicely Python supported every C++ paradigm I cared about, and more besides, I adopted it as my hobby-project language. At the same time, I wondered if there wasn't another step I could take: as C++ is to Python, perhaps so Python is to some other, yet more powerful language?

Graham argues Lisp is that language, the language which other languages are slowly catching up with.

Some languages are more equal than others

Why does Graham rate Lisp so highly? *Hackers and Painters* provides the authorative answer, but I'll attempt to summarise the main arguments here:

Lisp is good because of its power. Lisp is a high-level language which enables you to write concise, beautiful code. Programmers need a language which allows them to express themselves clearly and to constantly revise and test their code. Why, then, isn't everyone using Lisp? Partly because programming language choices are made by managers, not hackers, and partly because Blub programmers can only think in terms of Blub, and therefore cannot begin to appreciate what Lisp can do.[6]

I found these arguments provocative and entertaining, but what really impressed me was the quotation Graham cites as the single most important thing for people to remember when writing programs:

Programs should be written for people to read, and only incidentally for machines to execute.

Initially, this statement seems radical—isn't the whole point of programs that a computer can execute them?—but on reflection, I agree. I believe this principle should guide every aspect of our programming. It turns out that the quotation appears in the preface to *Structure and Interpretation of Computer Programs*.

Studying SICP

As you can see, my interest arose more from an interest in Lisp rather than a particular interest in studying computer programming, which is the real subject of SICP. I'll leave it until my next update to report more fully on how I'm finding Lisp: so far, I have been enjoying the language, but the development environments seem primitive—and I haven't begun to investigate library support.

The book itself is written with intelligence and wit. I read quite a few computing books but don't often have the time or motivation to work through the exercises. The ACCU project helps with the motivation, and my efforts are being repaid in full. ■

Resources

- The ACCU website is at http://www.accu.org
- The primary project resource is the email list itself: http://www.accu.org/mailman/listinfo/accu-mentored-sicp
- The book, *Structure and Interpretation of Computer Programs*, by Abelson and Sussman, is published by MIT Press. The full text is available online at: http://mitpress.mit.edu/sicp/full-text/book/book.html
- Videos of the accompanying lectures can be downloaded from: http://swiss.csail.mit.edu/classes/6.001/abelson-sussman-lectures/
- Schemers.org, an improper list of Scheme resources is at: http://www.schemers.org
- Finally, more information on this mentored developer project can be found on the project Wiki: http://wiki.wordaligned.org/sicp

Notes and references

1. Abelson and Sussman, *Structure and Interpretation of Computer Programs*, McGraw Hill, ISBN 0070004846.
2. "From Mechanism to Method: A Fair Share, Part 1", Kevlin Henney, http://www.ddj.com/dept/cpp/184403842
3. "Hackers and Painters", Paul Graham, http://www.paulgraham.com/hackpaint.html
4. "How To Become A Hacker", Eric Steven Raymond, http://www.catb.org/~esr/faqs/hacker-howto.html
5. In fact, Scheme does provide two special forms for looping: `do`, and the more general `named let`.
6. This is the Blub paradox. Blub is a hypothetical programming language, whose name has been chosen so as not to offend (or prejudice) users of a real programming language.

Thanks

Especial thanks to Mike Small for running this project, and for providing some of the raw material which went into this report. Thanks also to the other members of the mailing list.

XP Day 6

27th & 28th November, 2006, London, UK

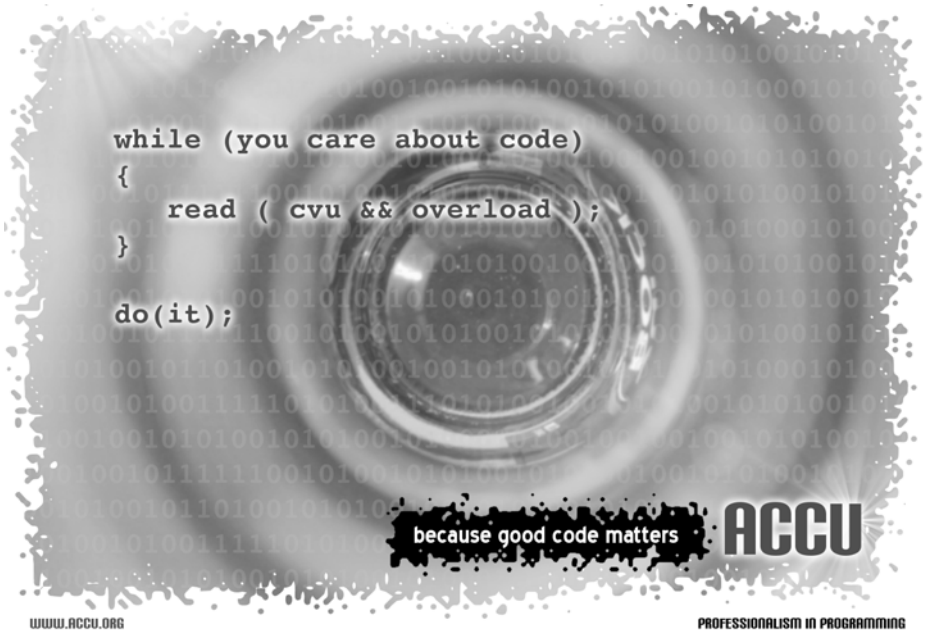
Two day international conference about Agile Software Development suited for practitioners at any level of experience, with a strong focus on practical knowledge, real-world experience and active participation of all attendees.

Keynote speaker: Joshua Kerievsky.

More information at www.xpday.org.

XP day

More than XP. More than one day.



Trip Switch Booleans in C++

Software with a circuit breaker? Bill Rubin explains how.

I've long held the view that advances in programming have come as much from adding constraints as from adding features. Computers were first programmed in machine language with no constraints. Instructions and data could be freely intermixed, either being used as the other. The advance of high-level languages (FORTRAN, COBOL) constrains instructions and data to be separate. The advance of structured programming (C, PL/I) constrains the flow of control among statements. The advance of object-oriented programming constrains the scope of functions and data. This short note proposes two modest C++ classes to constrain the way a boolean variable can be changed. These classes encapsulate what I call the *trip switch protocol*. When this protocol is applicable, the classes reduce software entropy, compared with using bare **bool** variables.

A *trip switch* is a two state device like a common household circuit breaker. A circuit breaker can be manually "initialised" to the "on" state, and can automatically be "tripped" into the "off" state, but it cannot automatically change back into the "on" state. Once off, it stays off. The protocol of a trip switch can be represented by a **bool** variable with "on" and "off" mapped into **true** and **false** values.

Unencapsulated trip switch

Consider how I first used the trip switch protocol without encapsulating it: I was doing file I/O with a class **File**. I defined a **bool** data member called **isValid_** and used it as a trip switch to manage file validity as shown in Listing 1. (The **File** class is pseudo-code, not really what I wrote but simply intended to illustrate implementation of the protocol).

Data member **isValid_** is initialized to **true** in each constructor. During each stage of accessing the file, **isValid_** is set to **false** if something goes wrong. Once **false**, it makes no sense to set it to **true**.

For industrial-strength code, it is good programming practice to include the code comment `Note 1` to document the intent of the design surrounding the **isValid_** usage. Without this code comment, future readers of the code can never be sure of the intent. They can search the code base and discover that **isValid_** is never explicitly assigned the value **true**, but that fact alone does not imply that the designer intended that it never should be. Thus, `Note 1` gives important guidance, not otherwise available, to maintainers and other code readers.

Problems with unencapsulated trip switch

There are at least three problems with the above approach of using a bare **bool** data member to implement the **isValid_** trip switch protocol. Firstly, the decision to employ the trip switch protocol is not made in one place, but is distributed throughout the **File** class. In a realistic setting, the statements involving **isValid_** would not be grouped together, but would be buried among all sorts of other concerns in both header and source files. The design principle of separation of concerns is not well served.

Secondly, it is easy for a maintenance programmer to overlook `Note 1`, especially when the **File** class is much more complex than shown here. If this occurs, the maintenance programmer may degrade the code base by introducing an assignment statement

```
isValid_ = true;
```

somewhere in the **File** class implementation. This statement degrades the code base for one of the following two reasons:

1. The maintenance programmer inadvertently violated the design intent, possibly introducing a bug.
2. The maintenance programmer correctly assumed (or was indifferent to) the design intent, and deliberately changed it. But because the programmer overlooked `Note 1`, it remains in the code, and is now wrong. A wrong comment is far worse than no comment. It will surely confound future maintainers.

Thus, the insertion of a code comment, while laudable, is by no means foolproof.

Thirdly, the bare **bool** approach doesn't scale well. I recently wrote a very small application which required half a dozen trip switch protocol instances in three enclosing classes. (Some instances used the inverse protocol, with "on" and "off" mapped into **false** and **true**.) Adding the constructor initializer for each constructor of the enclosing class, and for each **bool** data member, and adding the same code comment to each data member all violate the DRY principle (Don't Repeat Yourself).

Encapsulated trip switch

All the above problems are avoided by encapsulating the trip switch protocol in the simple class shown in Listing 2.

An object of type **BoolSettableToFalse** can be used almost exactly as a **bool** object, since it implicitly converts to and from **bool**. The main exception is that it cannot be assigned to. Only the **setToFalse()** member function can be used to change the value. There's no way to set the value to **true**, and that's the whole point of the class. There is no performance penalty – the run time and space are the same as for a bare **bool** variable.

Listing 1

```
class File {
 public:
  File(/*  some arguments */) : isValid_(true),
... {...}
    File(/* other arguments */) :
isValid_(true), ... {...}
    void access() {
      if(pathNotFound())    isValid_ = false;
      else if(openFailed()) isValid_ = false;
      else if(readFailed()) isValid_ = false;
     }

    bool isValid() const {return isValid_;}
    // Other member functions ...
  private:
    bool isValid_;   // See Note 1 below.
    // Other data members ...
};
/* Note 1: This bool is initialized to true, and
may be set to false, but is never set back to
true.
*/
```

BILL RUBIN

Bill is President of Quality Object Software, Inc., and a contract programmer spcializing in C++ development. When not unscrambling the latest software conundrum, he enjoys technical rock climbing. Bill can be contacted at rubin@contractor.net

Listing 2

```
class BoolSettableToFalse {
public:
    BoolSettableToFalse(const bool v = true) :
value_(v) {}
    operator bool() const {return value_;}
    void setToFalse() {value_ = false;}
private:
    BoolSettableToFalse& operator=(const
BoolSettableToFalse&);
    bool value_;
};
```

Listing 4

```
class BoolSettableToTrue {
public:
    BoolSettableToTrue(const bool v = false) :
value_(v) {}
    operator bool() const {return value_;}
    void setToTrue() {value_ = true;}
private:
   BoolSettableToTrue& operator=(const
BoolSettableToTrue&);
   bool value_;
};
```

The copy assignment operator is private to prevent protocol violations; one could otherwise change a false **BoolSettableToFalse** to a true one by assignment. The conversion constructor from **bool** is needed so that **bool** variables can be implicitly converted to **BoolSettableToFalse**. This constructor allows initializing to **false**, which is useful for **BoolSettableToFalse** instances which start out life already tripped.

The big advantage of the encapsulation approach is that *it enforces the trip switch protocol at compile time*. Variables of type **BoolSettableToFalse** clarify the design intent, and eliminate the need for descriptive comments and for initializers in enclosing class constructors. The decision to employ the trip switch boolean protocol is made in one place – the variable declaration – rather than being distributed throughout all references to the variable. The only disadvantage I am aware of is that the **setToFalse()** member function is less intuitive than a simple assignment.

Applying **BoolSettableToFalse** to the **File** class example yields Listing 3.

The new **File** class differs from the original implementation in the following ways:

1. The type of **isValid_** has been changed from **bool** to **BoolSettableToFalse**.
2. The **File** constructors are not shown, because they no longer participate explicitly in **isValid_** initialization. (They may still be needed, but not to implement the trip switch protocol.)
3. The `Note 1` code comment has been removed, because it's no longer needed. The design intent is encapsulated in `BoolSettableToFalse`.
4. The **isValid_** assignment statements have been replaced by calls to **setToFalse()**.

In this implementation, the decision to use the trip switch protocol is made in one place, by specifying the type of the **isValid_** data member. Less client code and no client comments are needed to support the protocol. The only code that must change is the type of **isValid_** and its assignment statements. A maintenance programmer cannot inadvertently violate the integrity of the code base because assignments to **isValid_** will not compile, and because there are no longer any code comments to become incorrect.

Listing 3

```
class File {
 public:
    void access() {
     if(pathNotFound())
isValid_.setToFalse();
     else if(openFailed())
isValid_.setToFalse();
     else if(readFailed())
isValid_.setToFalse();
    }

    bool isValid() const {return isValid_;}
 private:
    BoolSettableToFalse isValid_;
};
```

The dual of **BoolSettableToFalse** is **BoolSettableToTrue**, which is useful for the inverse protocol shown in Listing 4.

Of course, both these classes are useful not only for data members, but also for automatic variables and so on.

Summary

The **BoolSettableToFalse** and **BoolSettableToTrue** classes can be used in place of **bool**s to implement the trip switch protocol. The advantages are typical of those for encapsulation generally: Behaviour is enforced at compile time; code comments are not needed; the protocol is implemented only once; multiple instances scale up efficiently. Some may argue that the trip switch booleans are too trivial to bother with. My view is that simple classes like these are part of a style of programming in which a large number of "trivial" refinements accumulate to provide a much more robust code base. ■

Acknowledgements

Many thanks to Thomas Becker, Bill Collier, Leslie Hancock, Arthur Lewis, and Doug Lovell for helpful reviews of this article.

JOIN THE ACCU

You've read the magazine. Now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

Join ACCU to receive our bi-monthly publications *C Vu* and *Overload*. You'll also get massive discounts at the ACCU developers' conference, access to mentored developers projects, discussion forums, and the chance to participate in the organisation.

What are you waiting for?

How to join
Go to www.accu.org and click on Join ACCU

Membership types
Basic personal membership
Full personal membership
Corporate membership
Student membership

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Developer Beliefs about Binary Operator Precedence

Derek Jones carried out an experiment at the conference.

A common developer response to a fault being pointed out in their code is to say *What I intended to write was ....* A knowledge based mistake occurs when a developer acts on a belief they have that does not reflect reality. The act may be writing code that directly or indirectly makes use of this incorrect belief. An example may occur in the expression `x & y == z`, if its author, or a subsequent reader, incorrectly expects it to behave as if it had been written in the form `( x & y ) == z`.

This is the first of a two part article that reports on an experiment carried out during the 2006 ACCU conference investigating both knowledge of binary operator precedence and the consequences of a limited capacity short term memory on subjects performance in recalling information about assignment statements.

It is hoped that this study will provide information that can be used to build a model of how developers judge operator precedence and the impact that different kinds of identifier character sequences have on the cognitive resources needed during program comprehension. Such a model will be of use in analysing source code for possible, knowledge based, coding errors.

This article is split into two parts, the first (this one) provides general background on the study and discusses the results of the relative precedence selection problem, while part two discusses the assignment problem.

The hypothesis

Knowledge based mistakes might be broadly divided into those that don't seem to follow any pattern and those that do (or at least seem to). Here we are interested in finding patterns in the mistakes made by developers in choosing which of two binary operators has the higher precedence.

Studies have consistently found a significant correlation between a person's performance on a task and the amount of practice they have had performing that task [2]. Based on these findings it is to be expected that there will be a significant correlation between a developer's knowledge of relative binary operator precedence and the amount of experience they have had handling the respective binary operator pair.

A reader of the visible source only has to make a decision about the relative precedence of binary operators when an operand could have two possible binary operators applied to it. This paper uses the term *operator pair* to refer to the two operators involves in this decision. For instance:

```
x + (y * z);      // No reader choice to make
                  // because parenthesis make it.

x + a[y * z];     // No reader choice to make
                  // because of bracketing effect
                  // of [ ]

x + y * z << a;   // Two pairs of binary
                  // operators. A choice has to
                  // be made about which operator
                  // y and z bind to.

x + (y * z) | a; // One pair of binary operators
                  // The parenthesized expression
                  // could be the operand of +
                  // or |
```

As *table 1* shows, less than 2% of all expressions contain two or more binary operators.

This analysis assumed that subject performance is not influenced by the order in which the two operators in an operator pair occur.

Non-source code experience

There are two main situations where developers are required to comprehend expressions and thus gain experience in deducing precedence. These are formal education and reading/writing source code.

Within formal education, expressions are encountered when reading and writing mathematical equations. Many science and engineering courses require students to manipulate equations (expressions) containing operators that also occur in source code. Students learn, for instance, that in an expression containing a multiplication and addition operator, the multiplication is performed first. Substantial experience is gained over many years in reading and writing such equations. Knowledge of the ordering relationships between assignment, subtraction, and division also needs to be used on a very frequent basis. Through constant practice, knowledge of the precedence relationships between these operators becomes second nature; developers often claim that they are natural (they are not, it is just constant practice that makes them appear so).

Experience suggests that even people from an engineering background sometimes associate division from right to left. Division is often written in a vertical, rather than a horizon, direction in hand written and printed mathematics, so people have much less experience handling cases where it is written purely horizontally.

Measuring source

The following subsection discusses measurements of binary operator usage in the visible source of a number of large C programs (e.g., gcc, idsoftware, Linux, Netscape, OpenAFS, openMotif, and postgresql). The visible, rather than the preprocessed, source was used as the basis for measurement because we are interested in what a reader of the source sees and has to make decisions about, and not what the compiler has to analyse.

The contents of preprocessor directives were not included in the measurements.

Your author has not yet made sufficiently extensive measurements of the source code of programs written in any other languages for anything worthwhile to be reported here. Some broad brush measurements of FORTRAN [7] and PL/1 [1] source code have been made by other authors.

Binary operator usage in expressions

The languages C, C++, C#, Java, Perl, PHP, and some other languages share a large common subset of binary operators having the same relative precedence and associativity. These binary operators, with precedence decreasing from left to right are shown in Figure 1 (operators in the same column have the same precedence).

DEREK JONES

Derek used to write compilers, then got involved with static analysis and now spends his time trying to figure out developer behaviour. Derek can be contacted at derek@knosof.co.uk

Figure 1

High **Low**

```
[]  *  +  <<  <   ==  &  ^  |  &&  ||  assignment
()  /  -  >>  >   !=                    operators
.   %        <=
->          >=
```

Table 1

Percentage of expressions containing a given number of binary operators in their visible source code. Note that function call, direct and indirect member selection, and assignment operators are not included here as binary operators (although the arguments of function calls and array subscript expressions are counted as separate expressions).

Binary operators	% occurrence	Binary operators	% occurrence
0	92.82	3	0.62
1	5.28	4	0.14
2	0.86	5	0.13

Table 2

Occurrences, as a percentage of all such operators, of binary operators in the visible C source code. The second column is based on all occurrences of binary operators in expressions (890,423 operators). The third column is based on occurrences of of binaryoperator pairs in an expression (e.g., a + b & c * d contains the operator pairs + & and & *, so & is counted twice). There were 309,150 binary operator pairs

Operator	All occurrences of operator	Occurrences in an operator pair	Operator	All occurrences of operator	Occurrences in an operator pair
&&	8.55	22.17	>	3.82	3.60
+	12.46	13.41	>=	2.18	2.24
==	17.62	12.07	/	1.92	2.07
||	5.23	11.66	<=	1.34	1.33
*	4.32	6.34	&	9.59	0.70
|	5.19	6.25	<<	2.67	0.44
-	6.27	6.14	%	0.49	0.25
!=	8.73	5.65	>>	1.84	0.24
<	7.59	4.34	^	0.22	0.08

Table 3

Occurrences of pairs of binary operators as a percentage of all binary operator pairs (the total number of such operator pairs was 154,575). Table arranged so that operators along the top row have higher precedence. The '-' symbol denotes zero pairs.

	*	/	%	+	-	<<	>>	<	>	<=	>=	==	!=	&	^	|	&&	||
*	1.222																	
/	1.218	.116																
%	.006	.005	.001															
+	6.214	1.125	.104	5.344														
-	1.271	.479	.026	3.345	1.405													
<<	.032	.007	.001	.005	.031	.010												
>>	.037	.004	-	.005	.019	.018	.006											
<	.328	.479	.013	1.001	.822	.026	.009	-										
>	.373	.147	.020	1.188	.774	.013	.026	.003	-									
<=	.068	.032	.001	.349	.137	.003	.004	-	-	-								
>=	.126	.068	.004	.421	.298	.010	.006	-	.001	-	-							
==	.086	.049	.160	.352	.469	.028	.052	-	.001	-	-	.001						
!=	.057	.032	.074	.207	.178	.008	.017	.001	-	.001	-	-	-					
&	.003	.007	.001	.006	.005	.083	.115	.001	.001	-	.001	.001	.004	.144				
^	.018	-	-	-	-	.003	.003	-	.001	-	-	-	-	-	.057			
|	.016	.005	.003	.003	.012	.558	.114	-	-	-	-	-	-	.022	-	5.829		
&&	.093	.043	.045	.272	.222	.006	.012	3.466	2.577	1.692	2.530	13.157	7.959	.516	.001	-	5.392	
||	.038	.019	.019	.183	.098	.007	.006	2.487	2.054	.362	1.005	8.924	2.685	.245	-	.001	.021	2.827

While technically they are binary operators array subscript, function call, direct and indirect member selection, and the assignment operators are often not thought of as such by developers. For the rest of this paper the term *binary operator* should be read as excluding these operators unless stated otherwise.

The source code contexts included in these measurements were all the ones in which the C grammar permitted an expression (a *full expression* to use C Standard's terminology) to occur (excluding preprocessor directives).

For learning, or reinforcement of existing knowledge, of binary precedence to occur a binary operator pair needs to be encountered. Measurements of occurrences of binary operators in such pairs, rather than all binary operators, is what our hypothesis suggests subject performance should be correlated with. Table 2 lists both sets of measurements.

Table 3 lists the percentage occurrence of each operator pair as a percentage of all operators pairs in all expressions.

Use of parentheses in expressions

If the authors of source code always used parenthesis to specify the intended binding of operands to operators, then neither they or subsequent readers would need to have any knowledge of the operator precedence actually specified by a language; the information they needed would always be visible in the source.

For instance, the expression `a * b + c` could be written as `(a * b) + c`. The first expression is defined to be the operator pair `* +`, while the second is defined to be a parenthesized binary operator `(*) +`. In `(a + b << c) * d` there is no parenthesized binary operator because there are two binary operators within the parenthesis. However, there is an operator pair, i.e., `+ <<`.

Measurements (see Table 4) showed that several operator pairs (see * entries) almost always appear with redundant parenthesis to explicitly specifying the intended precedence.

When parenthesis are used to specify the intended binding of the operands to operators, readers of the source do not need to make use of their precedence knowledge. Thus there is no learning experience and these occurrences are not included in any count used here.

The experiment

The experiment was run by your author during a 45 minute lunchtime session of the 2006 ACCU conference (http://www.accu.org) held in Oxford, UK. Approximately 300 people attended the conference, 18 (6%) of whom took part in the experiment. Subjects were given a brief introduction to the experiment, during which they filled out brief background information about themselves, and they then spent 20 minutes working through the problems. All subjects volunteered their time and were anonymous.

One person wrote on the answer sheet that they did not understand the parenthesis problem. This person did not answer any of the parenthesis problems (a few seemed to have been answered, but not in a way that allowed the intent to be unambiguously deduced). Results from 17 subjects were used.

Table 4

Ratio of occurrences of parenthesized binary operators where one of the operators is enclosed in parenthesis, e.g., (a * b) - c, and the other is the corresponding operator pair, e.g., (*) - occurs 0.2 times as often as * -. The table is arranged so that operators along the top row have highest precedence and any non-zero occurrences in the upper right quadrant refer to uses where parenthesis have been used to change the default operator precedence, e.g., * (-) occurs 0.8 times as often as * -. The total number of these parenthesized operator pairs was 102,822 and the total number of operator pairs was 154,575. When there were no corresponding operators pairs in the visible source the entries are marked with a *.

	(*)	(/)	(%)	(+)	(-)	(<<)	(>>)	(<)	(>)	(<=)	(>=)	(==)	(!=)	(&)	(^)	(|)	(&&)	(||)
*	0.1	0.3	9.5	0.2	0.8	2.3	4.4	0.0	0.0	-	-	0.1	0.1	64.8	0.1	-	-	-
/	0.6	1.2	3.8	0.4	1.6	10.5	4.7	-	-	-	-	-	-	1.6	*	-	-	-
%	5.4	10.5	6.0	4.2	4.2	22.0	*	-	-	-	-	-	-	5.0	*	-	-	-
+	0.2	0.2	1.3	0.0	0.3	337.1	101.7	0.0	0.0	-	0.0	0.0	0.1	104.3	*	13.2	0.0	0.0
-	0.2	0.3	3.1	0.1	0.3	11.5	8.5	0.0	0.0	-	0.0	0.0	0.0	45.4	*	0.5	-	0.0
<<	2.8	10.5	34.0	43.1	20.9	1.5	2.2	0.1	-	-	-	0.1	0.2	10.4	2.8	0.1	-	-
>>	3.4	2.0	*	68.9	22.6	1.3	1.3	-	-	-	-	-	-	8.3	2.2	0.0	-	-
<	0.2	0.2	0.5	0.3	0.4	5.6	5.4	-	-	-	-	-	2.0	96.0	*	-	-	-
>	0.2	0.2	0.2	0.3	0.3	3.9	1.6	-	-	-	-	-	*	45.5	2.0	*	0.0	-
<=	0.3	0.4	1.5	0.2	0.4	4.4	5.0	-	-	-	-	-	1.0	*	-	*	-	-
>=	0.2	0.1	1.0	0.2	0.4	4.4	2.9	-	-	-	-	-	-	55.0	-	*	-	-
==	0.2	0.6	0.9	0.2	0.2	1.7	2.0	*	6.0	*	*	*	9.0	4689	*	*	0.0	-
!=	0.2	0.4	0.5	0.1	0.2	4.2	3.1	8.0	*	-	*	*	*	487.8	*	*	-	-
&	7.2	2.7	0.5	108.8	168.9	15.1	20.9	3.0	1.0	*	2.0	3.0	1.5	0.3	*	55.9	-	-
^	0.5	*	*	*	*	12.8	38.0	*	2.0	-	-	*	*	*	0.1	*	-	-
|	5.1	2.3	1.2	13.0	6.9	6.6	1.7	*	*	-	*	*	*	67.7	*	0.0	-	1.0
&&	0.1	0.1	0.9	0.2	0.2	3.0	2.1	0.3	0.3	0.4	0.3	0.4	0.5	6.1	18.0	*	0.0	26.0
||	0.1	0.2	0.7	0.1	0.3	2.7	2.9	0.4	0.4	0.3	0.3	0.6	0.7	4.6	*	27.0	25.5	0.0

The problem to be solved

The problem to be solved followed the same format as an experiment performed at a previous ACCU conference and the details can be found elsewhere [6].

Figure 2 is an excerpt of the text instructions given to subjects.

Figure 3 is an example of one of the problems seen by subjects. One side of a sheet of paper contained three assignment statements while the second side of the same sheet contained the five expressions and a table to hold the recalled information. A series of X's were written on the second side to ensure that subjects could not see through to identifiers and values appearing on the other side of the sheet. Each subject received a stapled set of sheets containing the instructions and 40 problems (one per sheet of paper).

The parenthesis insertion task can be viewed as either a time filler for the assignment remember/recall problem, or as the main subject of the experiment. In the latter case the purpose of the assignment problem is to make it difficult for subjects to track answers they had given to previous, related operator pair, problems.

Subject's performance on the parenthesis task is discussed in this article (the first of two).

The parenthesis problems

Based on the results of the 2004 ACCU experiment [6] it was anticipated that on average subjects would answer around 20 questions. By practising on himself the author concluded that answering five parenthesis problems would take approximately the same amount of time as answering the **`if`**-statement problem used in the 2004 experiment. This gave an estimated 100 answers per subject (subjects actually averaged 123.5 answers).

To simplify the analysis, it was decided subjects would encounter every operator pair in a problem before they say the same pair again. If subjects were expected to answer around 100 problems then some operators needed to be removed from consideration. The binary operators used had 10 levels of precedence (some shared the same precedence levels), giving 90 possible combinations of pairs of operators if we require the operator pairs to have different relative precedence. It was decided to include a small number of problems where the operator pairs had the same precedence and the same precedence pairs **+**/**−** and **==**/**!=** were chosen. The operators **`*`**, **`/`**, and **`%`** share the same precedence and span the range of frequency of occurrence in source code (see Table 5) and were all included for this reason.

The **<** operator was chosen to represent the relational operators because it is the most commonly occurring of the four (just over 50% of all relational operators), and the **<<** operator because it is slightly more common than **>>**.

If one set of two operators having the same precedence could appear there are 110 possible pairs, for two sets of two operators having the same precedence there are 121 possible pairs, and so on. The problems were automatically generated using an awk script which was parameterized by a data file that specified when to use pairs of operators having the same precedence.

All possible operator pairs (from which ever set of operators was selected by the data file) were generated and their order randomised, as was the ordering of the individual operators of a pair. The generation process was then repeated until enough precedence problems had been generated for 40 subjects (8,000 problems). Two hundred problems were successively taken from this list for each subject.

The binary operators that appeared in the problems seen by subjects, with precedence decreasing from left to right are (operators in the same column have the same precedence):

```
*    +    <<    <    ==    &    ^    |    &&    ||
/    -               !=
%
```

Figure 2

The task consists of remembering the value of three different variables and recalling these values later. The variables and their corresponding values appear on one side of the sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. Read the variables and the values assigned to them as you might when carefully reading lines of code in a function definition.
2. Turn the sheet of paper over. Please do **NOT** look at the assignment statements you have just read again, i.e., once a page has been turned it stays turned.
3. At the top of the page there is a list of five expressions. Each expression contains three different operands and two different operators. Insert parenthesis to denote how you think the operators will bind to the operands (i.e., you are inserting redundant parenthesis).
4. You are now asked to recall the value of the variables read on the previous page. There is an additional variable listed that did not appear in the original list.
   - if you remember the value of a variable write the value down next to the corresponding variable,
   - if you feel that, in a real life code comprehension situation, you would reread the original assignment, tick the "would refer back" column of the corresponding variable,
   - if you don't recall having seen the variable in the list appearing on the previous page, tick the "not seen" column of the corresponding variable.

Figure 3

---------- first side of sheet starts here ----------

```
p = 13;

q = 72;

r = 29;
```

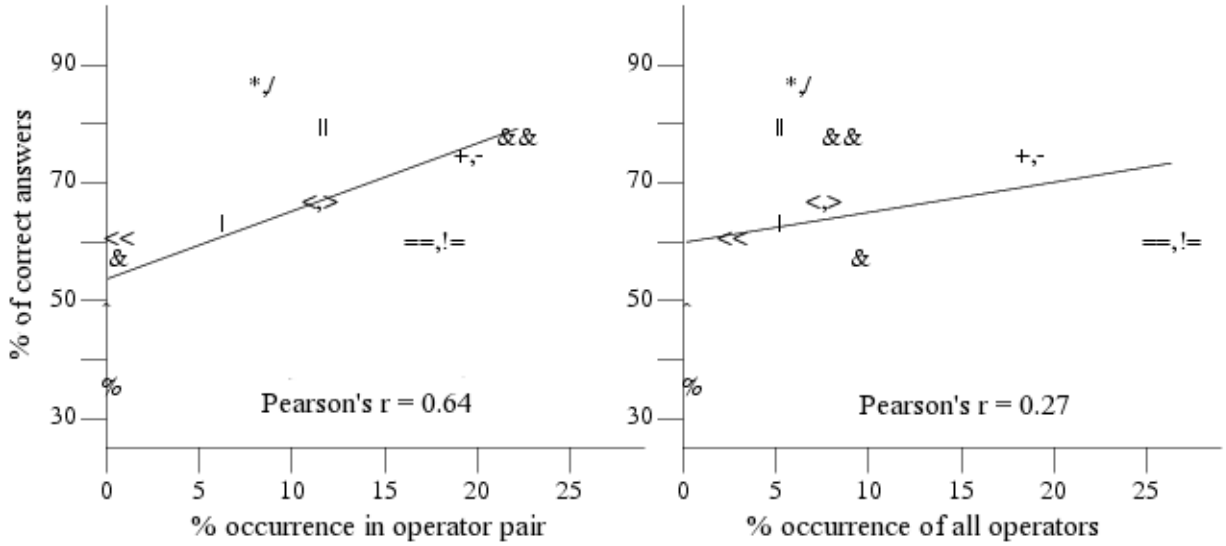
---------- second side of sheet starts here ----------

```
x + y * z

x / y == z

x != y - z

x % y ^ z

x << y - z
```

	remember	would refer back	not seen
q =	____	__	__
p =	____	__	__
s =	____	__	__
r =	____	__	__

Figure 4

The left graph is the number of precedence problems answered (average 123.5) against number of years of professional experience for that subject (average 14.6). The right graph is the percentage of incorrect answers given by a subject against years of professional experience (the line is a least squares fit on the percentage of incorrect answers).



The number of problems generated for the % operator was significantly less than for the other operators and the `!=`, `-`, and `/` operators did not occur as frequently as the other operators (see Table 5). This difference was driven by the data script used to control the generation of the problems. At the time it was thought to be a good idea.

Results

Subject experience

Traditionally, developer experience is measured in number of years of employment. In practice the quantity of source code (lines is one such measure) read and written by a developer (interaction with source code overwhelmingly occurs in its written, rather than spoken, form) is likely to be a more accurate measure of source code experience than time spent in employment. Developer interaction with source code is rarely a social activity (a social situation occurs during code reviews) and the time spent on these social activities may be small enough to ignore. The problem with this measure is that it is very difficult to obtain reliable estimates of the amount of source read and written by developers. This issue was also addressed in studies performed at a previous ACCU conference [5][6].

It has to be accepted that reliable estimates of lines read/written are not likely to be available until developer behavior is closely monitored (e.g., eye movements and key presses) over an extended period of time.

A plot of problems answered against experience for both the 2004 or 2006 (see Figure 4) experiments did not show any correlation between years of experience and number of problems answered. While a least squares fit of years of experience against percentage of incorrect answers shows a slight upward trend, the Pearson r correlation coefficient is not significant at the 0.05 level.

Subject motivation

When reading source code developers are aware that some of the information they see only needs to be remembered for a short period of time, while other information needs to be remembered over a longer period. For instance, when deducing the affect of calling a given function the names of identifiers declared locally within it only have significance within that function and there is unlikely to be any need to recall information about them in other contexts. Each of the problems seen by subjects in this study could be treated in the same way as an individual function definition (i.e., it is necessary to remember particular identifiers and the values they represent, once a problem has been answered there is no longer any need to remember this information).

Subjects can approach the demands of answering the problems this study presents them with in a number of ways, including the following:

- seeing it as a challenge to accurately recall the assignment information (i.e., minimizing *would refer back* answers) and not being overly concerned about accurately answering the parenthesis questions,
- recognizing that *would refer back* is always an option and that it is more important to correctly parenthesize the list of expressions.
- making no conscious decision about how to approach the answering of problems.

Experience shows that many developers are competitive and that accurately recalling the assignment information, after parenthesizing the expression list, would be seen as the ideal performance to aim for.

At the end of the experiment subjects were asked to turn to the back page and list any strategies they used when answering problems. The answers given all related to remembering information about the assignment statements. There was no mention of the parenthesis problem. One person wrote on this page that: "Not usually programming in a language that supports shift operations." which suggests that they are using a language that is not among the set being considered here, and that the language they use assigns different precedences to the operators it supports (but then, whether the precedence actually defined by a language has any effect on developer performance is one of the questions raised by the results of this study).

Analysis of results

The 17 subjects gave a total of 2,100 answers to the operator precedence questions (average 123.5 per subject, standard deviation 44.0, lowest answered 75, highest answered 200). If subjects answered randomly then 50% of answers would be expected to be correct. In this experiment 66.7% of answers were correct (standard deviation 8.8, poorest performer 45.2% correct, best performer 80.5% correct).

If the hypothesis is correct, then subjects will give more correct answers for some operators than for others. A first approximation is given, for each operator, by the percentage of problems containing the operator that were answered correctly (see Table 5). It is an approximation because there are two operators in each problem and incorrect knowledge about one of the operators will affect impact the perceived performance for the other operator.

If subjects guessed all answers we would expect 50% to be correct. A correct percentage either side of 50% is evidence that subjects are

Table 5

For each operator, the percentage of problems containing that operator that were answered correctly. Third column gives the total number of problems containing that operator.

Operator	Percentage correct	Number of problems
/	86.18	123
*	85.91	291
||	78.89	379
&&	77.21	408
+	76.63	291
-	72.39	163
<	66.43	414
|	62.72	397
<<	60.19	417
==	60.14	291
!=	59.20	125
&	56.70	455
^	47.64	403
%	34.88	43

consistently making use of some knowledge about the operator. When this knowledge agrees with that given in the language specification the correct percentage will be greater than 50% and when it is different the percentage will be less than 50%.

Some operators share the same precedence. If developers are aware of this, then it is possible that essentially the same piece of existing knowledge is reinforced in developer memory when either operator is encountered. If this is the case we would expect that subject performance for these *same precedence* operators will be similar. Table 5 shows that the percentage of correct answers for the operators **+** **-** and **==** **!=** ***** and **/** are very similar. Based on these results we might also expect subject performance for the *same precedence* operators **<<**, **>>** and **<**, **>**, **<=**, **>=** to be very similar, had they all appeared in problems.

The **%** operator has the same precedence as the ***** and **/** operators, but are developers aware of this connection? The very low percentage of correct answers containing **%** operator suggest that subjects were not aware of this fact.

The reinforcement that seems to take place when some same precedence operators appear in source means that an occurrence of either operator should count towards a learning experience for the same piece of knowledge. When plotting correct answers against operator occurrence, those operators which subjects know have the same precedence should have their occurrences summed (e.g., the 12.41% **+**'s and the 6.14% **-**'s are summed to give 18.55%).

The percentage of correct answers for the **||** and **&&** operators was very similar (see Table 5). Do developers believe they share the same precedence level, or do they know the precedence is different (it actually differs by one level) and the performance similarity is caused by subjects randomly selecting one of the operators to have the high precedence? The percentage of correct answers for the **|** and **&** operators was not as similar (again the precedence levels differ by one). Answers to the questions raised by the results for these operators will have to wait until more experiments are performed.

In Figure 5, the Pearson r correlation coefficients are 0.64 and 0.27. There are 11 operators (or combination of operators), which gives 9 degrees of freedom. Looking in statistical tables [4] for a one-tail test we find that the results in the left plot are significant at the 0.05 level of significance, while the straight line fit of the results in the right plot is not significant.

Operator pair-wise analysis

Does each developer have their own, incorrect, model of operator precedence, or do many developers share a common incorrect model? For instance, do a significant number of developers believe that one particular operator has a much lower precedence than it does in reality?

The Bradley-Terry model is one technique for analysing results that consist of paired comparisons. A common example of the use of the Bradley-Terry

Table 6

For each operator pair, the percentage of problems for which a correct answer was given for that operator pair. The value in parenthesis is the total number of problems containing that operator pair.

	*	/	%	+	-	<<	<	==	!=	&	^	|	&&	||
*	-													
/	90(10)	-												
%	-	-	-											
+	94(19)	69(13)	0(2)	-										
-	92(14)	100(1)	0(4)	80(10)	-									
<<	74(31)	84(13)	40(5)	58(31)	61(18)	-								
<	93(31)	100(13)	40(5),	90(33)	88(17)	64(50)	-							
==	100(19)	100(12)	100(24)	100(9)	83(31)	90(32)	-	-						
!=	100(12)	-	33(3)	100(6)	100(8)	92(13)	91(12)	57(7)	-					
&	71(35)	69(13)	40(5)	74(35)	47(21)	40(49)	41(51)	27(36)	11(17)	-				
^	63(30)	69(13)	25(4)	30(30)	33(15)	30(43)	16(43)	12(32)	8(12)	30(49)	-			
|	80(31)	84(13)	20(5)	70(30)	60(15)	45(46)	42(42)	22(31)	0(11)	79(44)	75(44)	-		
&&	100(31)	100(11)	60(5)	93(30)	94(18)	73(45)	75(44)	53(30)	71(14)	79(53)	88(45)	73(42)	-	
||	96(28)	100(11)	60(5)	92(28)	92(13)	69(42)	85(41)	53(28)	70(10)	87(47)	83(43)	86(43)	50(40)	-

Table 6

Coefficients obtained from applying a Bradley-Terry model to the pairs of binary operator precedence answers

Operator	Coefficient	Operator	Coefficient
/	3.37	|	1.48
*	3.33	%	1.21
+	2.59	&&	0.58
-	2.4	==	0.15
&	1.87	||	0.09
<<	1.57	!=	0.0

model is in ranking sports teams from the results of individual matches played by those teams.

Table 7 gives the output of using the Bradley-Terry model to rank operators in precedence order based on subject answers. The second column for each operator can be interpreted as providing an estimate the probability that one operator will be chosen to have a higher precedence over another. For instance, the coefficient for the operator **^** is twice as great as the coefficient for **|** and so when both occur together in an operator pair the **^** operator is twice as likely to be assigned a higher precedence.

However, be warned the error analysis is complex and the confidence in the calculated probabilities low; this issue is not discussed here.

The Bradley-Terry analysis was performed using the R system [8] with the addon package written by David Firth [3].

The coefficients for the same precedence operators *****, **/** and **+**, **-** and **==**, **!=** are very similar. However, there is little difference between the coefficients of many adjacently ranked operators. The rank assigned to the **%** operator is significantly different from the ***** and **/** operators (with which it shares the same precedence); this difference in precedence assignment is discussed later.

Discussion

The very large number of errors (approximately 33%) made by the subjects in this experiment is surprising. Had they guessed the answers the error rate would have been 50%. While experience shows that developers do make errors on the relative precedence of binary operators when writing and reading code, an error rate of this magnitude would generate many more faults (around 1% of all expressions would be in error) than appear to be encountered in practice.

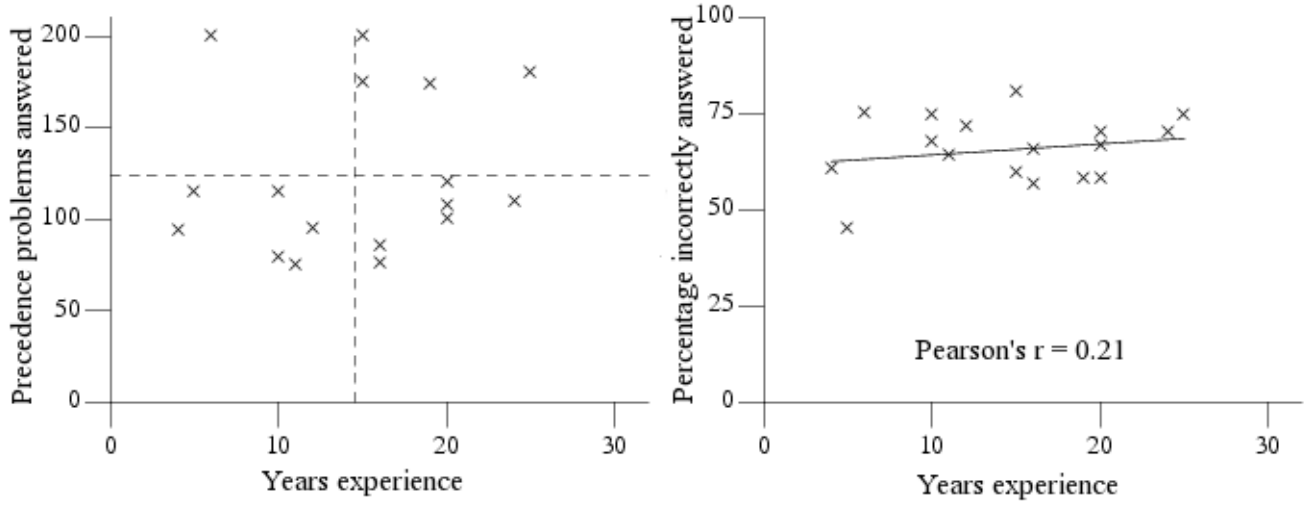
Possible reasons for the discrepancy between the error rates found in this experiment and those seen in commercial software development include:

- Some aspect of the experiment's design resulted in subject's performance being significantly poorer than it is when they comprehend source code professionally. For instance, subjects may have been overly distracted by the desire to correctly remember information about the assignment statements (subject's performance on the 'filler' problem, an if-statement analysis task, in the 2004 experiment was so good that insufficient error data was obtained for analysis).
- Developers make use of local context (e.g., the semantics suggested by the names of variables), rather than knowledge of operator precedence, to deduce how operands bind to binary operators. For instance, the semantic associations of the identifiers used in the expression:
  num_apples % num_baskets > max_left_over
  suggest that **num_apples** and **num_baskets** are in a calculation that checks whether there is an excessive number of apples remaining after some task is performed.
- Developers make use of knowledge of what is being calculated to choose the way in which operands bind to operators.
- Developers are aware of their own lack of knowledge of operator precedence and use parenthesis to indicate the intended precedence when they feel sufficiently unsure of the behavior that will occur. The ratios in **<tableref href="op_paren">** show that parenthesis are not always used for rarely occurring operator pairs. However, the high ratio of operator usage for some operators (e.g., see the **^** row) suggests that the overall impact may be significant.

Will measurements of binary operator usage in other language reveal different sets of common operator pairs? The most significant factor driving the choice of operators in source code is likely to be low level implementation details of the algorithms used. Various claims are made about how object oriented languages affect the choice and implementation of algorithms. Your author does not believe there is any significant operator usage variation between C++, C#, and Java. However, until detailed measurements are made it is not possible to claimed with any certainty that developer operator precedence performance for these languages is consistent with our hypothesis.

The percentage of correct answers for the ***** and **/** operators is well above that predicted by the least squares fit and it is also above that for the **+** and **-** operators. Of all the operators appearing in the problems the ***** and **/** operators have the highest precedence. Perhaps developers are aware of

Figure 5



The left graph is a plot of the occurrence of operators, as a percentage of all operators in operator pairs, in visible C source (see table 2), against percentage of correct answers to problems containing an instance of the operator (see table 5). The line is a least squares fit assuming the percentage correct deviates from the regression line. Operators separated by a comma have had their percentage occurrences summed and the correct percentages averaged. The right graph plots operators as a percentage of all operators occurring in all expressions.

coding guideline often contain a recommendation specifying that operator precedence be made explicit through the use of parenthesis

this *endpoint* information and it provides them with a decision algorithm that is simpler to use than for other operators, where information on an upper and lower precedence bound may need to be applied.

There is no obvious pattern to the other operators in the operator pairs that the multiplicative operators appeared with when a correct answer was given.

The percentage of incorrect answers for the `==` and `!=` operators is well above that predicted by the least squares fit. The Bradley-Terry analysis suggests that subjects believe, incorrectly, that the equality operators have the lowest precedence of all the operators seen in the study. While the results imply that subjects are using a much lower precedence for these operators than they actually have, the percentage of correct answers is not low enough to suggest that subjects are consistently using a lower precedence.

The results for the `^` operator are consistent with random guessing and this operator's precedence is sufficiently middling in the rankings that end-effects will be small.

Subject performance on the % operator is consistent with them believing it has a precedence level that is completely different than the one it actually has. More experiments are needed to uncover information on the model developers have for the `%` operator. These experiments might even inquire about the extent of developer knowledge about this operator (e.g., what operation to they believe it performs?).

Subject performance does not appear to be effected by the operator with which the `^` and `%` operators are paired.

Further work

What performance characteristics would we expect from subjects having much less experience than the subjects in this experiment? If subject performance is strongly correlated with operator pair reading experience, then we would not expect to see much less correlation between number of correct answers and source code occurrences.

Experiments using subjects who are just about to graduate and subjects a year or so after graduating (with and without extensive software development experience during that time) could provide the data needed to calculate the impact on performance of formal training and experience with source code.

A replication of the same experiment would be very useful, perhaps with some changes to the format. For instance, using different relational and shift operators, and more `%` problems; or changes the format of the problem to be solved so that a task other than remembering information about assignment statements is used.

What operations do developers think that the `^` and `%` operators perform? Perhaps the poor subject performance on these operators can be explained by the fact that many developers don't actually know what operation they perform, and hence where they might possibly fit in to the precedence hierarchy.

Conclusion

The results, Figure 5, are consistent with the hypothesis at the 0.05 level of significance. This means that there is a 5% chance that the null hypothesis (i.e., random answering) is true.

This is the first experimental evidence (perhaps partially indirectly) that software developer performance is effected by the number of times language constructs are encountered in source code.

If further experiments confirm the very poor binary operator precedence knowledge seen in this experiment, then developers must be using a significantly different technique for comprehending expressions (i.e., they are not relying on a knowledge of operator precedence).

Developer training is unlikely to be a viable option for solving the problem of faults caused by incorrect knowledge of operator pair precedence. While it is possible for people to learn the precedence of all language operators sufficiently well to pass a test, this knowledge will degrade over time through lack of use. Many languages contain operators that are rarely used in practice and it is knowledge of the precedence of these operators that developers are likely to forget the quickest (through of lack of practice).

While coding guideline often contain a recommendation specifying that operator precedence be made explicit through the use of parenthesis, experience shows that developers are often loath to insert what they consider to be redundant parenthesis (because ***Everybody*** *knows what the precedence is and if they don't they should not be worksing on this code*). Subject operator pair performance should be all the evidence needed to convince people that redundant parenthesis do provide a useful service. Under 2% of all expressions would need to make use of redundant parenthesis. ■

References

[1] J. L. Elshoff. *A numerical profile of commercial PL/I programs*. Software-Practice and Experience, 6:505-525, 1976

[2] K. A. Ericsson, R. T. Krampe, and C. Tesch-Romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review* 100:363-406, 1993, also University of Colorado, Technical Report #91-06.

[3] D. Firth. Bradley-terry models in R. *Journal of Statistical Software*, 12(1):1--12, Jan. 2005.

[4] P. R. Hinton. *Statistics Explained*. Routledge, 2nd edition, 2004.

[5] D. M. Jones. *I mean something to somebody*. *C Vu*, 15(6):17-19, Dec. 2003.

[6] D. M. Jones. *Experimental data and scripts for short sequence of assignment statements study*.http://www.knosof.co.uk/cbook/accu04.html, 2004.

[7] D. E. Knuth. *An empirical study of FORTRAN programs*. Sofware-Practice and Experience, 1:105—133, 1971.

[8] W. Venables, D. M. Smith, and R Development Core Team. *An introduction to R*. Apr. 2006

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?

Maintaining Legacy Code

David Carter-Hitchin gets to grips with legacy code.

As an infrastructure manager and programmer of a 20 year old 2+ MLOC C/C++ system, I am reasonably qualified (in theory) to give some advice about working on such a project. Before discussing any of the technicalities of the project, I thought it worthwhile to offer some general advice and encouragement for people out there dealing with totally different technologies. I have written this article with the view it may be read by people who don't know much about legacy code, and as such please forgive me for the occasional didactic tone.

1. Learn to be patient

If you're patient already, then fine. If you're not, then you're going to be disappointed, since old systems require hard work and time to turn around. The bigger a system is, the longer it will take, so be prepared to get results over a period of time, not all at once. Patience is a key life skill and it's all too important when working with old code!

2. Build a knowledge base

Information is power. Again, like patience, this will be a key factor in deciding the future life of your old code. Start documenting everything which seems significant, but don't hope to understand everything at once, and don't get swamped by detail. Find a sensible place to gather similar facts, like a Wiki. For example, if your software relies on a large number of third party of public domain libraries, then put these into an inventory.

Schematic diagrams showing the inter-relationships between parts of the code are very useful, as are diagrams showing relationships with the outside world. If your software takes in real world data, from external feeds for example, make sure these are documented (*frequencies, content, format*). In general understand what its input, processing and output are, but don't worry about being too high-level early on; it's better to have a complete, but vague overall picture than a highly detailed but incomplete one. The reason for this is that although you could be an expert on a particular subset of the code, if you are unaware of how it relates to the whole, then any changes to that subset are potentially dangerous, as your changes could lead to unexpected behaviour somewhere else.

If you don't see it as your role to maintain documentation, then find out what documentation has already been done, and whose role it is to keep it up to date. Even if you are lucky enough to be working in a team with full time technical writers, then try to make some time to read their output and give them some feedback.

3. Ensure your tools are in place to tackle the project

I'll say more of this below but a few things I would a have thought were essential would be a good source code indexer, source revision system and bug tracking tool. If there's a little nagging doubt in the back of your mind that you think that you should have a certain tool to do a certain job, then it's probably a good idea to make some time and follow that up. Tools are key to breaking down the problem of maintaining software into more manageable chunks.

DAVID CARTER-HITCHIN

David is currently a senior infrastructure developer for a bank in the City of London. His major interest is Mathematics and he is currently studing for a BSc and a CQF. David can be reached at david@carter-hitchin.clara.co.uk

4. Tame the beast

Sometimes old software is monstrous in nature – it's ugly, it's unpredictable, it's big (sometimes), it's dangerous ("Hey, where's all my data gone?!") and so on and so on. However monsters are not known for high IQs, and your average programmer is going to be much smarter than any system. Now to tame the beast, you need to study it (with patience!) to find out its strengths and weaknesses – again this will take time, it won't happen overnight (unless you're very lucky). Once you've understood it's weakest areas, then you can fix those. Fixing one problem often reveals another, so be prepared for that. Sometimes a fix doesn't work either – don't get disheartened – have patience!

Eventually, if you keep trying, you will make it.

5. Performance versus function, hardware vs software

First thing to do before you even think about writing or changing a line of code, is to ask yourself (and your sponsors) what improvements are most important. Broadly speaking these will fall into two camps 'performance' and 'functional' enhancements. By 'performance' I mean everything from speed of execution to safety and security considerations – security is normally dealt with separately to speed considerations, but I'm lumping the two here for ease of discussion. So, if your primary concern is performance, you should first conduct a thorough environment review – this should not take long, but it might take a long time to remedy any ill-findings. You need to audit the quality and number of machines (if known) your code is currently running on, type of network used and so forth.

Patience is a key life skill and it's all too important when working with old code!

Obviously if you are dealing with a third party or public domain app that gets shipped everywhere, then this is not so applicable to you. If, on the other hand your working environment is constrained then it's essential to look at that. If you (or your sponsors) have lots of money to spend then upgrading key machines could temporarily fix processing bottlenecks. Of course, it is better (and possibly cheaper) to fix the software, but this often takes longer than the time it takes to order and install a new piece of hardware. You need to understand the issues, and weigh up the cost/benefits.

6. Marry methodologies

Getting people to agree about the best way of doing things is essential. If you can't agree, it's better to elect a draconian ruler who will lay down the law, as opposed to letting things get out of hand.

Our project is twenty years old, and so there is a conflict between the old C gurus and the C++ youngsters. The arguments are very interesting, and I can see both sides, but at the end of the day, a unified approach has to be adopted for success. Here that approach is C++.

Tool chain

Well, so much for general considerations – now for some specifics. I'll start with the tool-chain we use.

To navigate our code we use Doxygen, LXR + glimpse for fast searches. I've also tried with Understand for C/C++, which was very good and Synopsis which I haven't managed to get working yet, but it looks very good.

Before your code gets anywhere near a compiler, there are a raft of tools to help improve coding safety and standards. We use Gimpel's Flexelint which takes about 6 hours to run on 2+ MLOC of C/C++.

there was no consistent method of database access, which can be a significant problem in terms of support and maintenance

It takes quite a while to clear out the noise from this tool, but when you start to see the real errors, it's well worth the price tag. We have also looked at Parasoft's C++Test which includes some unit testing software (more about that below) and CodeWizard, which is functionally rich, but to set up a tool like this for maximum effect on an existing codebase would take a lot of time. We have not completely out ruled using a tool like this, or similar tools like QAC++, but we can do plenty of things 'ourselves' first – in other words without the need for third party tools.

Keep it tidy

Code clean-ups are very useful, including formatting improvements, removal of unnecessary code/includes and generally rewriting code to be clearer. Doxygen with code annotation and include dependency and class graphs (you need to enable GraphViz to get those) are very enlightening. If includes are included twice, for example, this are displayed in red as a warning.

Anything to make the code clearer is welcome. A simple example is in the forward definition of enums/structs:

```
typedef struct _my_struct
{
 int a;
} MY_STRUCT;
```

This used to be required under old some C compilers, for use within the same translation unit, but now we can simply write:

```
struct MY_STRUCT
{
 int a;
};
```

Use STL

We are compiling all our C as C++ so we can do this now. In addition any opportunity to use the STL where appropriate is welcome. For example we had some horrid code which declared a fixed array of size 200, populated the array with data, then called a function which re-arranged the entries, throwing away duplicates. This was horrid primarily because of the arbitrary 200 size limit. It wouldn't have been too bad had there been some bounds checking and extension code there, but there was not – all we had was the result of a lazy programmer's assumption about the nature of the data – which in time proved to be false and this caused production problems. This was a classic case for a use of a `std::set`, which extends itself for me safely and without me having to write code, and as a bonus provides uniqueness. This is the strength of C++ – replacing clunky verbose C which was broken with a few lines of robust C++. The source code dropped from about 75 lines to 5. The C programmers would argue that the executable size would increase slightly, and they may be right, but in my opinion this is a price worth paying given that I've got code which is robust and much much smaller (therefore easier to understand, maintain, test and modify).

Refactoring

Another important concept is encapsulation of complex behaviour in order to refactor code.

A good example of this in our project is the multiple ways our application accesses the database. Over time numerous methods have been developed which exist in various places in the code. As a result there was no consistent method of database access, which can be a significant problem in terms of support and maintenance. In order to solve this a new C++ database classes was written which incorporated all the best features of the various methods and, over time, the various methods are being converted to the new database class. Once all methods have been replaced with invocations of the new class, then any maintenance or upgrades or vendor changes are relatively a lot more simple.

Compilation and checking

Our compiler is GCC. We've got to the point, after a lot of work, where we can have -Werrror enabled meaning that warnings get treated as errors, so compilation fails. This is a good thing with old code, as hidden nasties are often found this way, and new ones are nipped in the bud. Sometimes compilers and static analysers are the ONLY WAY we are going to find these subtle problems. We also play with -Wpendantic, -Weffc++ and -GLIBCXX_DEBUG although the output from these can be huge, so we can't enable them permanently. Having said that -Wpendantic may go in soon. I recently discovered -enable-concept-checks which you need to set when configuring GCC which applies some additional checks for STL usage.

all we had was the result of a lazy programmer's assumption about the nature of the data

Debugging

For debugging we mainly use Totalview and GDB. I'm looking forward to the bi-directional debugger UndoDB supporting threads and shared memory, we will use that when available. For memory analysis we use Valgrind on Linux and Purify and Insure++ on Solaris. We are beginning to look at DTrace and libumem on Solaris 10. With legacy C code, the responsibility for freeing up heap memory is with the application, and is easily overlooked. By wrapping the legacy code in a class, with the destructor responsible for freeing the heap memory, robustness can be improved. In the near future we will also be using a checked STL implementation, like STLport or Dinkumware.

Unit testing

For unit testing we are in an interesting position. The main part of our application is used to process financial information and perform financial calculations. Before every release we run a regression batch, meaning that we take a dump of the live database, and perform the same calculations with the new code on the old data. We then compare the output from the live run with the output from the test run to see if any of the results have 'regressed'. These calculations are extremely complicated and we run hundreds of millions of them and the outputs are compared down to the last penny, so any discrepancies show up very quickly. Although this is not a unit test in the normal sense, it is a very robust test for changed behaviour. In future though we plan to enhance these tests to cover the GUI side of the application and make the tests finer grained. I should mention Michael Feather's book *Working Effectively with Legacy Code*, which seems to focus mainly on unit testing. I haven't read all of this book, but it has a good reputation. ■

Standards Report

Lois Goldthwaite explains the proposed addition of two new character types to C++.

We are all citizens of the world, in these days of ever-increasing globalisation. Two papers in the latest 'mailing' [1] from the C++ standard committee propose to bring a greater helping of globalisation to the C++ standard.

N2018 by Lawrence Crowl [2], "New Character Types in C++", calls for adding two new character types to C++ in addition to the four we already have [3], in order better to support Unicode. The two new types would be called **`char16_t`** and **`char32_t`**.

N2035 by Matt Austern [4], "Minimal Unicode support for the standard library", outlines the impact of adding these two types on the C++ standard library. In a nutshell, it would double the number of instantiations for strings, locales, iostreams, and regular expressions [5].

Internationalisation, and especially the Unicode way of achieving it, are important issues in today's programming world, and growing more important all the time. In a message to the WG21 library committee, Austern says:

> [C++ Unicode support] isn't some obscure C-compatibility feature that will just sit harmlessly in an appendix of the standard and that you can ignore unless you're involved in some kind of specialized bureaucratic task. We're long past the tipping point where Unicode has become more important than ASCII. If we have Unicode in C++ (and I'll be very disappointed if we don't), it will be used prominently in important interfaces. XML is defined in terms of Unicode, for example, so I expect that any future XML library will be built on top of these things. You should take a close look at N2018 and N2035 and form an opinion about them on the assumption that this is something that you will use on a daily basis, that you will use it more often than you use **`wchar_t`** and possibly as often as you use **`char`**.

But wait a minute – Unicode support, doesn't C++ already have it? Isn't this what **`wchar_t`** is for? Well, yes and no. An array of wide characters *might* be a Unicode string literal, but then again it might be encoded according to some other system, such as Big5 Chinese characters. Furthermore, some vendors have defined **`wchar_t`** as a 16 bit data type, whereas others have defined it as 32 bits. For WG21 to announce that standard C++ now requires it to be just one of those sizes, with Unicode encoding only, would break many people's code.

Apart from that, C and C++ have traditionally been agnostic about character sets. Even if you confine the discussion to English text in Roman characters, you cannot take it for granted that all computers will represent that in ASCII characters. Consider IBM mainframes and minicomputers, which use the EBCDIC character codes instead – and in that system the letters of the alphabet are not even in a contiguous sequence! However, it is possible (with a little, well, rather a lot of attention to not building in unwarranted assumptions) to write portable programs which will compile and execute correctly whichever character set is in use.

Standard C++ also has what are called "null-terminated multibyte strings". These are strings encoded under a system which may use one or several bytes to represent a single character in some language like Japanese, using Shift_JIS encoding. Such NTMBSes could also be used to hold a string in the Unicode UTF-8 encoding.

Given that C++ has never before made specific rules about how the contents of a string should be interpreted, is it justified to require that these new character types are specifically limited to Unicode processing only? What about platforms for which ASCII is not the underlying character set – should they be required to support the 'alien' Unicode processing just to claim to be standards compliant?

C and C++ have traditionally been agnostic about character sets

Another question being discussed is whether names like **`utf16_t`** and **`utf32_t`** would be better and clearer about the intent of the new types. Perhaps, but **`char16_t`** and **`char32_t`** were chosen for ISO/IEC TR 19769:2004, "Extensions for the programming language C to support new character data types". As much as possible, the C++ committee tries to make sure that C code will compile under C++ unless there is a very good reason to break compatibility.

In C the two new character types are simply **`typedef`**s for integer types of the right size, but that won't work for C++ because it makes it impossible to distinguish functions overloaded on these types. They would have to be distinct fundamental types in C++, and that is what N2018 proposes.

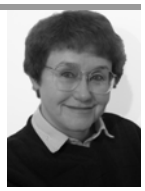
No committee decision has been made on how to answer the above questions, but you can bet that serious discussions will be taking place in the coming year while the C++0x draft is being hammered into shape. If you have an opinion on these issues that you would like to express, please send comments to standards@accu.org and they will be passed on to WG21. ■

Notes and references

1. It is called a 'mailing' because once upon a time, back in the dark ages, these periodic collections of technical proposals would be shipped via snail mail around the world as bundles of paper. Nowadays WG21 has an eco-friendly shiny green paperless process in which only electrons are shifted, but we haven't thought up a new name to reflect the new procedure.
2. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2018.html
3. These are **`char`**, **`signed char`**, **`unsigned char`**, and **`wchar_t`**, but you already knew that, right? It is a historical accident that C++ has three flavours of the simplest data type, **`char`**, two of which are identical – but not the same two on all platforms.
4. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2035.pdf
5. The regular expressions library in TR1 (authored by the UK's John Maddock) will be part of the standard library expected from all compilers in C++0x.

LOIS GOLDTHWAITE

Lois has been a professional programmer for over 20 years. She is convenor of the C++ and Posix standards panels at BSI. One of her hobbies is representing the UK at international standards meetings!
Lois can be contacted at standards@accu.org.uk



Student Code Critique Competition

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members. The title reflects the fact that the code used is normally provided by a student as part of their course work.

This item is part of the Dialogue section of C Vu, which is intended to designate it as an item where reader interaction is particularly important. Readers' comments and criticisms of published entries are always welcome, as are possible samples.

Before we start

Remember that you can get the current problem set in the ACCU website (http://www.accu.org/journals/).This is aimed to people living overseas who get the magazine much later than members in the UK and Europe.

Student Code Critique 40 entries

The student wrote:

> I'm having problems printing out a tree data structure – the code doesn't crash and I don't get any compiler warnings but the tree doesn't seem to get shown properly. It's based on some (working) Java code. This test program shown below only prints the head node and not the children. I've tried both C and C++ but the program behaves the same for both.

Please try to help the student understand *why* their code is broken as much as *where* it is broken.

```
#include <malloc.h>
#include <stdio.h>

struct binaryNode
{
  int value;
  struct binaryNode *left;
  struct binaryNode *right;
};

struct binaryNode createNode( int value )
{
  struct binaryNode *newNode;
  newNode = (struct binaryNode*)malloc(
     sizeof( struct binaryNode ) );
  newNode->value = value;
  newNode->left = 0;
  newNode->right = 0;

 return *newNode;
}

struct binaryNode addChildNode( struct
binaryNode parent, int value )
{
  struct binaryNode tempNode;
  tempNode = createNode( value );

 if ( value < parent.value )
    parent.left = &tempNode;
  else
    parent.right = &tempNode;
  return tempNode;
}

void printNodes( struct
binaryNode head,
                 int indent )
{
  int i;
  if ( head.left != 0 )
    printNodes( *head.left, indent + 1 );
  for ( i = 0; i < indent; i++ )
    printf( " " );
  printf( "%i\n", head.value );
  if ( head.right != 0 )
    printNodes( *head.right, indent + 1 );
}
int main()
{
  struct binaryNode head;
  struct binaryNode node1;
  struct binaryNode node2;
  head = createNode( 3 );
  node1 = addChildNode( head, 1 );
  node2 = addChildNode( node1, 2 );
  printNodes( head, 0 ); // Only prints 'head'
  return 0;
}
```

From Jim Hyslop <jhyslop@dreampossible.ca>

Hmmm... I bet I can see what's coming. "Tree data structure" in C and C++ almost invariably means pointers. Java hides all that icky pointer stuff for you. Probably some basic pointer problems are coming up.

```
struct binaryNode createNode( int value )
{
  struct binaryNode *newNode;
  ...
  return *newNode;
}
```

Just as I suspected. Java would naturally convert the return type into a pointer (yeah, yeah, "Java doesn't have pointers" – yes it does, it just manages their lifetimes for you).

All functions must be modified to accept and return pointers to `binaryNode` objects:

```
struct binaryNode * createNode( int value )
{
...
   return newNode;
}
struct binaryNode * addChildNode(
   struct binaryNode * parent, int value )
```

with, of course, corresponding modifications to the body of the function.

ROGER ORR

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

I'm surprised this program doesn't crash (and a more complicated one probably would). **tempNode** is a local variable, which goes out of scope at the end of the function. The parent's left/right pointer will point into, effectively, garbage. Converting to pointers solves this issue:

```
void printNodes( struct binaryNode head,
                 int indent )
```

I will assume there is a reason the printed out structure has to match the internal structure. Normally, you wouldn't need to worry about that.

OK, so much for the initial problem the student encountered. There are some significant design flaws. First of all, **addChildNode** is error prone. It relies on the user passing in the appropriate parent to **addChildNode**. If **node2** wants a value of 4, then we have to change the line from:

```
  node2 = addChildNode( node1, 2 );
```

to:

```
  node2 = addChildNode( head, 4 );
```

Nasty. This is going to cause no end of subtle, hard-to-find bugs.

In addition, there is no check to see if there is already a child. The tree will get messed up if the user types:

```
  head = createNode( 3 );
  node1 = addChildNode( head, 2 );
  node2 = addChildNode( head, 1 );
```

Instead of thinking about operations on nodes, the programmer should be thinking about operations on the tree. Tree handling should be separated into a distinct file, with an interface (Notes: untested, uncompiled code follows; assume appropriate include guards are present):

```
/* mytree.h */
typedef struct binaryNode
{
  int value;
  binaryNode * left;
  binaryNode * right;
};
binaryNode * createTree( int initialValue );
void addValue( binaryNode * root, int value );
void printTree( binaryNode * root );
void releaseTree( binaryNode * root );
/* end of mytree.h */
```

Note that **printTree** no longer accepts the indent value. This is an internal value that only **printTree** cares about.

Other design issues the student needs to consider: Does each value have to be unique? What about removing elements from the tree? What other operations will need to be performed on the tree, besides printing? Searching, for example? Do we know right now which operations will be performed? If we don't, then is it worth adding a **callback** function, to allow users to define their own operation, something like:

```
typedef int (CALLBACK *)(binaryNode *,
   void * callbackData, int depth );
int performOperation(binaryNode *,
   CALLBACK, void * callbackData );
```

performOperation simply recurses through the tree, and calls the callback function for each node. The **depth** parameter indicates how deeply nested in the tree we are (By the way, student: do we really need to know this?) The return value from the callback is used to determine whether or not to abort the processing:

```
/* mytree.c */
static int doRecursion( binaryNode * node,
   CALLBACK cb, void *callbackData, int depth ) {
  if ( node == NULL )
  return TRUE;
  if ( !doRecursion( node->left, cb,
     callbackData, depth+1 ) )
  return FALSE;
  if ( cb(node, callbackData, depth ) );
    return doRecursion( node->right, cb,
       callbackData, depth+1 );
  return FALSE;
}
void performOperation(binaryNode * node,
   CALLBACK cb, void *callbackData ) {
  doRecursion(node, cb, callbackData, 0 );
}
```

Using **printNode** as an example, we can now write this as:

```
static int printNode(binaryNode * node,
   void * unusedData, int indentCount ) {
  printf("%*.*s%i", indentCount, indentCount,
     " ", node->value); }
/* Note the use of %*.*s to give a variable-width
indent - no more 'for' loop! */
void printTree( binaryNode * root )
{
   performOperation( root, printNode, NULL );
}
int main()
{
  binaryNode * root = createTree( 3 );
  addValue( root, 2 );
  addValue( root, 1 );
  printTree( root );
  releaseTree( root );
}
```

And now, for some comments on style. These are subjective, and my opinion only:

- I would prefer to initialize the pointer **newNode** in the same statement as it is declared. In these small functions, it doesn't make any practical difference, but it's a good habit to immediately initialize variables to a sane value.
- In C I like to declare structs as **typedef struct** so I don't have to type **struct this; struct that;** (you might say I don't want to be **struct dumb;** by all that typing...).
- I prefer **NULL** over the literal 0. The standard **include** files should provide a suitable declaration of **NULL**. In C++, 0 is OK, but I don't remember if 0 is acceptable in C – I suspect it should properly be cast to a void pointer.

From Balog Pal <pasa@lib.hu>

Duh, another "unfair" SCC. To make the student understand the problem he must grok both the java and C++ object model. But to explain that and all its ramifications would take a chapter in a book. Or a couple.

But this piece of code looks so innocently translated from java, and became so utterly broken I can't help but to love it, and try to do something. Though I think Roger shall submit his own solution as punishment – either as an entry or a standalone article. :-)

Let's start with theory. Java has "primitive types" and "reference types". We're interested in the latter. "Objects", that are actual instances of a class or an array are always created on the heap. While the variables you encounter in code will contain a pointer to that object. (Java calls it the "reference value" and the java hypers who never read the langspec (4.3.1) claim there are no pointers in the language, well, actually all you have

around is pointers.) Variables that are members in a class, local variables in a function, function arguments, and function return values are all pointers. When you assign to such variable you assign a pointer. When you pass to a function you pass a copy of the pointer. What you return is a copy of the pointer. A copy of the pointer refers to the same object that sits on the heap. To have an actual copy of the object you must do some special incantations like **`Clone()`** and fight to make it work too. But you never deal with 'dereference' or 'address of'; having no variance the language does all that transparently.

The java heap is garbage-collected, automatically. You just use the pointers in a natural way, if you lose the last pointer to an object (by assigning a new value to the variable storing it), there's no way for the program to ever access it, but the system will eventually reclaim the space it occupies. While as long as you have a pointer anywhere, the object is safely 'living'.

In C++ the world is completely different. Here every object is a direct value. If you have an instance of a struct as a local variable, the struct will exist with its full body in that local stack frame. And cease to exist as you leave that scope. If you pass an object as function argument, or return from the function, it is copied. If you want the object created on the heap, you have to ask it, by using **`new`** (or **`malloc`** for raw memory). You get it allocated, and a pointer to it returned. Then you must eventually issue delete (or free) on that pointer to reclaim the space. If you just lose the pointer, you create a 'memory leak', and risk to waste all the system's memory. You must make sure in the design that allocations and deallocations are very properly paired.

You can also create pointers to existing objects using the address-of (**`&`**) operator. But you must be aware of the lifetimes of the objects. After you delete an object, or it is destroyed by leaving scope, any pointers you have to it are no longer valid. And using their value (even inspecting it, or passing as argument) invokes 'undefined behaviour' you absolutely want to avoid.

Now let's see why the code is wrong as written, and what it actually does:

```
struct binaryNode createNode( int value )
{
  struct binaryNode *newNode;
  newNode = (struct binaryNode*)malloc
    ( sizeof( struct binaryNode ) );
  newNode->value = value;
  newNode->left = 0;
  newNode->right = 0;
  return *newNode;
}
```

Here you allocate raw memory on the heap, store a pointer in **`newNode`** and initialize all its members. Though it's not the preferred C++ way, it's okay for C code, and is correct so far. But the function returns a struct (not a pointer) and you return **`*newNode`**. That means the struct that sits on the heap is copied to a temporary 'return value' structure. Then the function ends, variable **`newNode`** is destroyed, and so you lose any chance to ever call **`free()`** to reclaim the space on the heap. If you want to return a struct, it is better created as a local variable, set up and returned. If you want it on the heap, the pointer shall be returned.

```
struct binaryNode addChildNode(
struct binaryNode parent, int value )
{
  struct binaryNode tempNode;
  tempNode = createNode( value );
  if ( value < parent.value )
    parent.left = &tempNode;
  else
    parent.right = &tempNode;
  return tempNode;
}
```

Here the function will return a copy of a struct, and receive a copy too called **`parent`**. You set up a local node as **`tempNode`**, and set its state using **`createNode`**. Not counting the memory leak, it will work, a state with value and two nulls will be copied to the struct. Then you patch left or right of 'parent' and return a copy of **`tempNode`**. The function ends, all the local variables go out of scope, including that **`parent`** struct, with your carefully applied changes. I say fortunately, as if you managed to issue those changes in a node that is visible, you would face disaster. As **`tempNode`** is just destroyed, yet you stored its address in left or right. This is the explanation of the behaviour you observe, this **`addChildNode`** has no chance to modify the original parent struct you try to pass, so that one stays with null leafs. And you avoid the crash due to problems cancelling each other.

The other functions are broken in a similar way, so I leave the details for brevity, let's instead see a working translation. I made several variants but show only one. It is not the most effective solution, the aim was to make it as close to the original (but no closer), and to make it show how java code maps to C++, along with points where it must differ. And make it safe.

```
#include <stdio.h>
#include <boost/shared_ptr.hpp>
class binaryNode
{
public:
  typedef boost::shared_ptr<binaryNode> tPtr;
  // or std::tr1::
  typedef
     boost::shared_ptr<const binaryNode> tCptr;
private:
  // private constructor: creation is
  // restricted to factories
  explicit binaryNode(int v) :value(v) {}
public:
  // factory
  static tPtr Create( int value )
  {
    return new binaryNode(value);
  }
  tPtr AddChild(int child_value )  // mutator!
  {
    tPtr & p =
       child_value < GetValue() ? left : right;
    p = Create( child_value );
    return p;
  }
  // selectors
  int GetValue() const {return value;}
  tCptr GetLeft() const  {return left;}
  tCptr GetRight() const  {return right;}
private: // node is noncopyable!
  binaryNode(const binaryNode&);   // ni
  binaryNode & operator =(const binaryNode&);
  // ni
private: // state
  int value;
  tPtr left;
  tPtr right;
};
void printNodes( binaryNode::tCptr pNode,
   int indent )
{
  if(!pNode)
    return;
  printNodes( pNode->GetLeft(), indent + 1 );
  for ( int i = 0; i < indent; ++i )
    printf( "*" );
  printf( "%i\n", pNode->GetValue() );
```

```
  printNodes( pNode->GetRight(), indent + 1 );
}
int main()
{
  binaryNode::tPtr pHead
    ( binaryNode::Create(3) );
  {
    binaryNode::tPtr p;
    p = pHead->AddChild( 1 );
    p = p->AddChild( 2 );
    pHead->AddChild( 5 );
  }
  printNodes( pHead, 0 );
  return 0;
}
```

I'm sure in the original Java code **binaryNode** was a class, and transmuted to a struct only as part of creating a C (not C++) code. Now it is a class again. And the data members no longer publicly exposed. The Java best practices ask for immutable classes. I can't see a way to keep this one immutable, but also practical to use – but mutation must be kept at bay and under tight control. Allowing that anyone just poke the guts is out of question.

The plain pointer members got replaced by **shared_ptr**. If we have pointers we must look out for ownership issues, and must manage deallocations. Doing it by hand is cumbersome, error-prone, and would require plenty of code, so we better pick up a "smart" pointer that does most of the work. Among smart pointers **shared_ptr** (from the Boost library, see dox at www.boost.org, or TR1) is a kind that approximates the Java "references" very closely. It can be copied around as function parameter or return value, assigned at will, but still keeps a proper count, how many existing copies still refer to the object we created on the "free store" (C++ name for the heap) with **new**, and when the last is gone, it calls **delete** for us. It is almost as good as the garbage collector, only fails to recover objects referring to each other in a cycle.

To make the code more readable, we introduce **typedef tPtr**, that is inside **binaryNode**, that is the preferred C++ way as opposite to add another name to the outer namespace, like **binaryNodePtr**. And immediately pick its twin **tCptr**, that is a pointer to a **const binaryNode**. Java unfortunately miss the notation and the concept of **const**, and **const**-correctness. Thus relying on programmers to write immutable classes or just refrain from poking stuff without the language's help to introduce and enforce a contract. Going to C++ we better pick up **const** and use wherever possible. It is a great help. Fortunately **shared_ptr** works fine along with **const**, it can copy **tPtr** to **tCptr** while knowing they refer to the same object.

The use of the smart pointers is straightforward in most situations, just do as in the java code, except for member access **.** is changed to **->**.

The next thing in **binaryNode** is the constructor. In C++ we positively want constructors for our structures and classes, and make them initialize the data members. Java does init members to zeros or nulls automatically, C++ doesn't. And having uninitialized objects mean undefined behaviours and nasty debug sessions over weekends hunting phantom bugs that surface depending on what memory garbage got picked up. The assignments that were done in **createNode** migrate to the constructor's initializer list between **:** and **{**. If we still had plain pointers they would be listed too, our smart pointer members have their own constructors that are called, so can be skipped.

The constructor is made explicit, to prevent accidental implicit conversion from **int** to **binaryNode**. And it is made private. Private constructor means no one outside the class can call it. So you can't create an instance of **binaryNode** either with new or as a local variable in a function. Thus we can effectively restrict creation to a small set of controlled functions (factories). Those will know how to create the instance properly, and return a smart pointer – that the user can then safely handle.

The next function is the former **createNode**, it is moved into the class, so gains access to the private constructor, also encapsulates the name. It is certainly static, and I renamed it to **Create** that is the usual name for the basic factory. The users will call it as **binaryNode::Create()** anyway. It creates the instance using new and encapsulates the pointer in a **tPtr**. The member init code was moved to the constructor already.

Our other factory function is **AddChild**, created from **addChildNode**. It's also made a member, as it's more natural than to pass a node as argument, also we thus gain access to members we need. We find out which branch to put the new node, then assign the pointer returned from **Create()** there. If that branch was already occupied it gets nuked (the smart pointer does that). I'm not sure that was the intended behaviour but that's what the original version suggested. If the new node needs to be "dropped down" to an end of a branch, it can be programmed here. A pointer to the newly created node is returned to help a more convenient build of the tree.

Then come the selectors, all **const** functions and carefully return our constified pointer that will not allow mutation of the tree.

The next two lines prevents the compiler to create a **copy** constructor and assignment operator for this class, like simple creation we don't want the user to create copies of a node, also the copy created that way (referring to the same branch nodes as the original) would hardly make sense.

The **printNodes** function I left outside the class, it looks like part of the test framework. It certainly uses **tCptr**, not **tPtr** – no intention to modify the tree. The implementation uses the selectors. I made a simple transformation, but the original structure with **if**-s would work too. You still better look out for the case a passed pointer is null before using it.

main() is similar to the original just with the proper types.

In the class I didn't include any extra stuff not needed by this code, in practice probably some more selectors or specific member functions will be needed to help rearranging the tree.

From Reece Dunn <msclrhd@hotmail.com>

The problem with this code is that the student – having come from a Java background – has made some assumptions that are true in Java, but don't apply to C and C++.

The first assumption is that objects are passed by reference. Take the function:

```
struct binaryNode addChildNode(
    struct binaryNode parent, int value );
```

The parent object is a *copy* of what is passed to it. Therefore, the copy is modified, not the intended parameter passed into the function. Also, the value passed back is a copy of the newly created node. Java passes objects by reference, so in Java this isn't a problem.

The solution here is to make parent and the return value pointers so that the values are updated correctly. For **addChildNode**, this gives:

```
binaryNode * addChildNode( binaryNode * parent,
    int value )
{
  binaryNode * tempNode = createNode( value );
  if( value < tempNode->value )
    parent->left = tempNode;
  else
    parent->right = tempNode;
  return tempNode;
}
```

The other problem is that there is a memory leak, because the **malloc** call in **createNode** does not have a corresponding call to free. In Java, memory is automatically managed by the garbage collector, but in C and C++ it is up to the programmer to explicitly free that memory.

You can use smart pointers to use the Resource Aquisition Is Initialisation (RAII) idiom that means that the programmer does not have to make an explicit call to clean up resources. I will use the Boost version here, but these are also available in C++ TR1. The C++ version looks like this:

```
#include <boost/shared_ptr.hpp>
#include <iostream>
struct binaryNode;
typedef boost::shared_ptr< binaryNode >
binaryNodePtr;

struct binaryNode
{
  int value;
  binaryNodePtr left;
  binaryNodePtr right;
  binaryNode( int value_ ):
    value( value_ )
  {
  }
};
binaryNodePtr addChildNode(
   binaryNodePtr parent, int value )
{
  binaryNodePtr tempNode( new binaryNode(
   value ));
  if( value < tempNode->value )
    parent->left = tempNode;
  else
    parent->right = tempNode;
  return tempNode;
}
void printNodes( binaryNodePtr head,
                 int indent = 0 )
{
  if( head->left )
    printNodes( head->left, indent + 1 );
  for( int i = 0; i < indent; ++i )
    std::cout << " ";
  std::cout << head->value << std::endl;
  if( head->right )
    printNodes( head->right, indent + 1 );
}
int main()
{
  binaryNodePtr head( new binaryNode( 3 ));
  binaryNodePtr node1( addChildNode( head, 1 ));
  binaryNodePtr node2( addChildNode( node1, 2
));
  printNodes( head );
  return 0;
}
```

Note that using C++ means that **createNode** can be (and is in the above example) replaced by defining an appropriate constructor and the new operator.

From Calum Grant <calum.grant@sophos.com>

This problem highlights a fundamental difference between the way C, C++ and Java manage memory. In C and C++ there is a distinction between a value and a pointer. For example

```
struct binaryNode n1;
```

means that **n1** is a value (a **binaryNode**), whilst :

```
struct binaryNode *n2;
```

means that **n2** is a pointer (to a **binaryNode**). When values are assigned to each other, the data is copied – in C++ this is done via a **copy** constructor or an operator **=**, whilst in C this is a simple memory copy. When pointers are assigned to each other, only the pointer is copied, not the value itself.

In Java, there is no distinction between pointers and values, since technically speaking there are only pointers to objects, so there is no need for the *****, **&** and **->** notation.

The value/pointer distinction has significance when you define a function. When a function returns a value, it returns a copy of the value. When an argument to a function is a value, the value is copied into the function, leaving the original unmodified.

Your original program was returning a copy of **newNode** in **createNode()**, not **newNode** itself. Similarly, **addChildNode()** actually modifies a copy of the node passed into it, which means that it has no effect, and explains your output.

To make your program equivalent to Java, you need to change the functions such that they take and return pointers, not values. This is done as follows:

```
struct binaryNode *createNode(int value)
struct binaryNode *addChildNode(struct
  binaryNode *parent, int value)
void printNodes(struct binaryNode *head, int
  indent)
```

You also need to declare **tempNode**, **head**, **node1** and **node2** to be pointers and not values, otherwise they will take copies.

You have a bug in **addChildNode()**. You need to check if there is already a value in **parent->left** or **parent->right**, and call **addChildNode()** recursively.

You might ask why C and C++ bother to make the distinction between pointers and values, when it all works so well in Java. The reason is because C and C++ have manual memory management, which gives you much more control over when and where objects are created and destroyed. On the other hand, Java has automatic memory management (garbage collection) so everything is allocated on the heap and the programmer does not need to worry. This extra flexibility in C and C++ is a common source of errors, but can give better performance.

Since C and C++ have manual memory management, one needs to be mindful of who is responsible for freeing any memory you allocate. In C, you need a function that recursively frees the entire data structure.

C++ often does not need to worry about memory management because it can use containers and smart pointers to manage memory. In our case, we could use **std::auto_ptr** to automatically initialize the pointer to 0, and automatically delete what it is pointing to when it is destroyed. C-style pointers are best avoided in C++ since they are more prone to error.

You could also consider turning **binaryNode** into a class. A class should be designed so that it is very difficult (or impossible!) to use it incorrectly. A class should prevent operations that could invalidate the class. A 'class invariant' is a condition that should always be true, for example that the nodes are ordered correctly. Therefore left and right should be private so that they cannot be modified outside of the class. Another invariant is that the value should always be initialised and not change, so we need a constructor to initialize the value, and value could be **const**.

We must also check that the default copy constructor and assignment operators do the right thing. In this case they don't, so we must disable them by making them private, or implement them to do the right thing.

We don't need to hard-code **int** into **binaryNode**. The class can hold any data type (**T**) by making the class a template. We also need to know how to compare two items, which is usually done via a comparator (**Cmp**), rather than relying on the **<** operator. The default comparator is **std::less**, however we may want to change this.

Here is a C++-style implementation of **BinaryNode**. Compared to the C implementation, the C++ implementation is safe, generic, easier to use, and impossible to abuse.

```
#include <string>
#include <iostream>
#include <functor>
#include <memory>

template<typename T,
   typename Cmp = std::less<T> >
class BinaryNode
{
  const T value;
  std::auto_ptr<BinaryNode> left, right;
  BinaryNode &operator=(const BinaryNode&);
  BinaryNode(const BinaryNode&);
public:
  BinaryNode(const T &v) : value(v) { }
  void insert(const T &v)
  {
    if(Cmp()(v, value))
    {
      if(left.get())
        left->insert(v);
      else
        left.reset(new BinaryNode(v));
    }
    else
    {
      if(right.get())
        right->insert(v);
      else
        right.reset(new BinaryNode(v));
    }
  }
    void display(int indent=0) const
  {
    if(left.get())
      left->display(indent+1);
    std::cout
      << std::string(indent, ' ')
      << value
      << std::endl;
    if(right.get())
      right->display(indent+1);
  }
};
int main()
{
  BinaryNode<int> node(3);
  node.insert(1);
  node.insert(2);
  node.display();
  return 0;
}
```

From Nevin Liber <nevin@eviloverlord.com>

First off, let me say that I found the code to be pretty good. Functions and variables are named well, and functions are short and each one tends to have a single purpose.

Misunderstanding

The code is broken due to a fundamental misunderstanding about objects in C. C is a pass-by-value language. Objects and their lifetimes are managed explicitly. If one wishes to build a recursive data structure (such as a tree), then the nodes are allocated (via `malloc`) in the heap, and the rest of the program manipulates pointers to those objects. Unlike Java, C does not have built in garbage collection, so one must explicitly free those objects when they are no longer needed.

Specifics

```
#include <assert.h>
#include <malloc.h>
#include <stdio.h>
```

I will be using `assert` to signal programming errors detected at run time. I'll expand on that when I discuss `addChildNode`.

```
typedef struct binaryNode binaryNode;
struct binaryNode
{
  int value;
  binaryNode *left;
  binaryNode *right;
};
```

The `typedef` is a notational convienence: it allows me to replace `struct binaryNode` with `binaryNode` everywhere else that I use it. The code was not incorrect before; I just find that there is less clutter this way.

```
binaryNode *createNode( int value )
{
  binaryNode *newNode = (binaryNode*)malloc(
    sizeof( *newNode ) );
  if ( newNode ) {
    newNode->value = value;
    newNode->left = 0;
    newNode->right = 0;
  }
  return newNode;
}
```

`createNode` now returns a `binaryNode *` instead of a `binaryNode`. Note: while it is unlikely, `malloc` can fail. I added the `if` check for this possibility, and only set its members on success. However, since this routine now returns a pointer, the callers of it may also have to check for failure (which is indicated by a NULL pointer).

```
binaryNode *addChildNode( binaryNode *parent,
   int value )
{
  assert( parent );
  binaryNode *tempNode = createNode( value );
  if ( value < parent->value ) {
    assert( !parent->left );
    parent->left = tempNode;
  } else {
    assert( !parent->right );
    parent->right = tempNode;
  }
  return tempNode;
}
```

Made the changes to use pointers everywhere.

But now that everything is pointer based, someone could pass NULL in for parent. Should we allocate the node in that case? I don't know. And since I don't know what to do, I consider it a programming error to pass in NULL, which I indicate by using `assert`.

There is a similar issue with the `parent->left` or `parent->right` that gets replaced. If it is non-NULL, then those point to part of a tree. What should I do? Recursively destroy those nodes? Not allocate our new node? Without knowing the intentions of the programmer, I won't guess. Instead, it is another programming error to do so, and I'll again indicate that by using `assert`.

```
void printNodes( binaryNode const *head,
   int indent )
```

```
{
  if ( head ) {
    int i;
    printNodes( head->left, indent + 1 );
    for ( i = 0; i != indent; ++i )
      printf( " " );
    printf( "%d\n", head->value );
    printNodes( head->right, indent + 1 );
  }
}
```

Once again, use pointers. Note the use of **const**: since **printNodes** does not change the data structure that **head** points to, the **const** documents that in a compiler enforceable way.

Bug corrected: the correct **printf** specifier for **int** is **%d**, not **%i**.

[Ed: the C standard allows either.]

Since I have to check if head is non-NULL anyway, I simplified the code by getting rid of the checks on **head->left** and **head->right**, as the recursive call to **printNodes** will catch that anyway. When implementing recursive algorithms, I find that having one terminating condition is far more understandable than multiple ones scattered throughout the function. The termination condition for this function is when head is NULL.

```
void destroyNodes( binaryNode * head )
{
  if ( head ) {
      destroyNodes( head->left );
      destroyNodes( head->right );
      free( head );
  }
}
```

This is a new function I added. As mentioned before, in C one has to clean up the heap objects when finished with them. This recursive function has the same basic structure as **printNodes**, and the termination condition is when **head** is **NULL**.

```
int main()
{
  binaryNode *head = createNode( 3 );
  binaryNode *node1 = addChildNode( head, 1 );
  binaryNode *node2 = addChildNode( node1, 2);
  printNodes( head, 0 );
  destroyNodes( head );
  return 0;
}
```

Again, fixed everything up to use pointers.

I leave it as an exercise for the reader if the calls to **createNode** or **addChildNode** returned NULL. The reason I am leaving it as an exercise for the reader is that what should be done may depend on behavioural changes made to **addChildNode** (to eliminate the asserts).

I added the call to **destroyNodes**, since we need to clean up the heap objects when we are done with them. Caveat: once called, **head**, **node1** and **node2** can no longer be dereferenced, since they point to memory that is no longer a **binaryNode**.

From Seyed H. HAERI (Hossein) <shhaeri@math.sharif.edu>

The evil in this SCC isn't that hidden. The student has come from Java, and is not still used to preliminaries of (manual) memory management. What has caused the problem reported is not this one however. Despite that, the latter problem also stems from coming from Java. The student is wrongly assuming that whilst calling by value, you get references to the same object. That's why, as one can see, all the arguments to the functions are copied (wrongly) supposing that they will be references to the same place in memory. This said, not willing to change the C structure of the code, changing all (but one) of the occurrences of **binaryNode** to **binaryNode***, and a little number of syntactic fixations would turn the program into something not having the mentioned problem. (I hope the editor doesn't expect me to add the full above fixed code.)

As told above, the evil isn't this one however. The student leaves a dreadful memory leak in his program. Any trained eyes looking to this code will soon understand that it allocates memories whilst it doesn't give them back; it uses **malloc**, yet there aren't any **dealloc**s! Trying to add free wouldn't mitigate anything because again this is not the leak. The leak is quite nasty: in the **createNode** function, the student allocates memory, copies its contents and returns. This allocated memory will be dangling thereafter because no pointer possesses it anymore. And this is evidently a direct result of coming from Java. Applying the above change to the program, one would then have to return the ownership of the allocated memory to the caller. This will pass the caller the duty of cleaning it up, and, adding appropriate number of **dealloc**s will force the devil to run away. This doesn't end the story however.

The code is furthermore in C, as opposed to in C++. This means that, given that the code is pretty trivial, there shouldn't be much surprise in that it behaves the same in C/C++. (I know that this is not a rule, and can well break…) The things that should be noted whilst porting it into C++ are:

1. Use ADTs for both nodes and data structure! Classes, of course, are the C++ gadgets for this. This will open the door of a world of goodies to you. For example, in the current version, one can easily change the pointers to children of the nodes without being checked ever. This is whilst those pointers are in fact internals of the nodes, and should well be forbidden for the strangers. Had you encapsulated this data, you wouldn't allow strangers this easily change the internals of your code.
2. Use **new**/**delete** instead of **malloc**/**dealloc**. This will prevent you from the dangers of unwillingly dealing with raw memory. For your level at least, this is not a good piece of practice.
3. Don't declare variables until their point of use. Try adhering to this unless there remains no other way for you.
4. Use **std::cout** instead of **printf**. This will save you from the bother of using the appropriate label.
5. Replace all **if(p != 0)**'s with simply **if(p)**. (I guess this is correct also in C. But, I'm not sure…). [SCC: it's OK in C too]
6. Remove all the redundant uses of the keyword **struct**. For example, in C++ you need not to write

   ```
   struct binaryNode* createNode(int)
   ```

   you could simply write

   ```
   binaryNode* createNode(int)
   ```
7. Except for the above function in which you have to return the allocated memory, use references instead of pointers. This will make coding easier. (You don't need to worry about the usage of **operator *** and **operator ->**. Your familiar **operator .** will do.)

Next, I come to the coding recommendations. First, I would like to note the student about the point that if the student is about to implement a BST – as it is a common assignment for the level – but the student is doing that wrongly. I'm not sure whether this is the intent, but whatever that is, the unstructured/semi-structured addition of elements is an alarm bell. This may lead to some sort of anarchy in the data structure. (I am not clear what the data structure is at least.)

Second, I give a plus to the student for appropriate naming of variables and functions. However, the types of arguments, as mentioned above, aren't appropriate. I don't see any bad coding practice except that he/she doesn't comment anything – which is a big sin! (Yes, this happens whilst the code is definitely trivial.) **printNodes** should get its argument constantly because it doesn't change the tree it gets. So, another minus! If the code is not an assignment which is supposed only to work with **int**s, then all the material should be templatised to prevent hard-coding. All **operator --**'s should be turned to prefix. And, finally the code hasn't got an appropriate interaction with the user.

From Lars Hartmann <lars@hartmannix.dk>

The main issue here is calling conventions.

There are at least three ways to pass arguments to a function:

1. Call by value (CBV)
2. Call by name (CBN)
3. Call by reference (CBR)

Each of these can be supported by any given language, but they might not be. Understanding the differences between them, and knowing if they are supported or not by a language will aid a lot when faced with a porting task.

The calling conventions above are characterized as follows:

- CBV: When using this calling convention we copy values when passing them on to functions. Values might be a fixed set of types given by the programming language, or the language might allow the extension of the types that can act as values.
- CBN: This is used when we give an argument a name, and let the receiving function look up the value associated with the name when it is needed. This also makes it possible to change the value associated with the name (In C/C++ this is done by passing a pointer).
- CBR: This is a mix of the above. The reference has a name like in CBN, and the name is used to access or modify the associated value. The name is used in place of a value, like the value is used in CBV.

What we need to consider here are: Which constructs are available in Java, C and C++.

	Java	C	C++
CBR	x		x
CBV		x	x
CBN		x	x

So when we need to take a Java implementation and port it to C or C++ we need to decide how we want to pass the information along to functions. This decision might not seem very important right now, but remember the copying that takes place when using CBV, this makes it hard to update data structures without having to recreate them all the time. It is possible to do this, but using either CBN or CBR (depending on which language you port to) seems a lot easier.

The distinction between CBR and CBN in C++ is very subtle, and one could argue that the Java CBR concept looks a lot more like C++ CBN that C++ CBR.

If all this talk with calling conventions are confusing this might be a good time to look for more information.

Having said this, its time to look at the code.

```
struct binaryNode
{
  int value;
  struct binaryNode *left;
  struct binaryNode *right;
};
```

This struct is quite all right, the student has decided to use CBN (that is, pointers). Good choice no copying problems to take care of.

Next the function to create a node:

```
struct binaryNode createNode( int value )
{
  struct binaryNode *newNode;
  newNode = (struct binaryNode*)malloc(
     sizeof( struct binaryNode ) );
  newNode->value = value;
  newNode->left = 0;
  newNode->right = 0;
  return *newNode
}
```

At first glance this will look very all right to a Java person. The problem is that the returned value of the function is a value and not a pointer to a value. When we return a 'bare' type like this, we will get a copy of the object returned, which most likely was not the intention.

Keeping the the decision to use CBN it should have looked like:

```
struct binaryNode* createNode( int value )
{
  struct binaryNode *newNode;
  newNode = (struct binaryNode*)malloc(
     sizeof( struct binaryNode ) );
  newNode->value = value;
  newNode->left = 0;
  newNode->right = 0;
  return newNode;
}
```

The **addChildNode** function uses CBV calling convention, this means that the arguments to this function are first copied before changes are made to them, and when the function returns, these temporary objects are destroyed. The function actually returns a valid node object, but the parent object that stored the pointer to this object will point at the memory of the tempNode, not the returned node, therefore the pointer stored in the parent is not valid. Instead we should use CBN here as well:

```
struct binaryNode* addChildNode(
   struct binaryNode* parent, int value )
{
  struct binaryNode* tempNode;
  tempNode = createNode( value );

  if( value < parent->value )
    parent->left = tempNode;
  else
    parent->right = tempNode;
  return tempNode;
}
```

The final function **printNodes** is again using CBV with the same array of problems as described above. Changing it to use CBN solves the problem:

```
void printNodes( struct binaryNode* head,
   int indent )
{
  int i;

  if( head->left != 0 )
    printNodes( head->left, indent + 1 );
    for( i = 0; i < indent; i++ )
    printf( " " );
    printf( "%i\n", head->value );

  if( head->right != 0 )
    printNodes( head->right, indent + 1 );
}
```

All that is left now is to adapt **main** to use these new functions:

```
int main()
{
  struct binaryNode* head;
  struct binaryNode* node1;
  struct binaryNode* node2;
  head = createNode( 3 );
  node1 = addChildNode( head, 1 );
```

```
  node2 = addChildNode( node1, 2 );
  printNodes( head, 0 );
  return 0;
}
```

This neatly prints three lines of indented numbers, one for each node.

Commentary

I'd like first to pick up on Balog's complaint "to make the student understand the problem he must grok both the java and C++ object model". This is true; but I don't think it is unrealistic. I haven't worked on a single-language project for years; the last project I worked on used many languages including: C++, Java, PL/SQL, PHP, Bash, Perl, HTML and R. My experience may perhaps be atypical – what is your own experience?

One of the things I find interesting about multi-language projects is that each programmer knows different languages to different levels; and a programmer moving from a familiar language to a new language can produce bugs someone more used to the language would never think of.

So it is with this code. An experienced C programmer would never produce it but it looks plausible to a Java/C# programmer because it is in line with their mental model of 'how things work'. The task of the critique is to explain not just what is wrong but *why* – and this does involve some knowledge of the C memory model (and some knowledge of the Java model will help too).

However, there is a *design* issue with the code too (as some entrants pointed out). It appears most clearly in **addChildNode** where the child node is added based on a comparison of the new value with that of the parent node; but there is no validation that the chosen branch is not already in use. Safe use of this function requires co-operation between the caller and the implementation.

To my mind this is perhaps more serious than the details of syntax. Even if the code was 'fixed' to compile and execute successfully in C the result would be very brittle to the exact values and parents passed in to successive calls to **addChildNode**.

I would be trying to find out more from the student about the task their code is actually trying to solve; my suspicion is that they are trying to produce a binary-sorted tree and need a recursive search for adding a child node in the correct place.

As is common with the SCC sadly the student is unavailable for questioning when you produce your critique, but I would be happy to see a shortlist of questions you'd like to ask if they were to hand!

The Winner of SCC 40

The editor's choice is:

Lars Hartmann

Please email francis@robinton.demon.co.uk to arrange for your prize.

Student Code Critique 41

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

The student wrote:

> I'm having trouble with deleting things from a collection.
>
> The code used to work, it printed out:
>
> ```
> contents: 123
> deleted: 2
> contents: 13
> ```
>
> Then I upgraded my compiler and it complained about "delete iterator" (see '`now won't compile!`' in the code).
>
> Someone explained that iterator was only 'like' a pointer so I should use **&*** on it to get the pointer back. But can someone explain why I need **&*** – does this do *anything* at all?
>
> Anyway, I did try this, and got it to compile with the newer compiler but the program sometimes crashed. I've tried compiling with full warnings but I don't get any – is this compiler broken?

Please point out both good *and* bad things the student is doing.

```
#include <vector>
#include <iostream>
#include <string>
using namespace std;
class VectorTest
{
  vector<string>::iterator iterator;
  vector<string> vector;
public:
  void addToVector( string string )
  {
    vector.push_back( string );
  }
  void printVector( ostream & ostream )
  {
    ostream << "contents: ";
    for ( iterator = vector.begin();
      iterator != vector.end(); iterator++ )
    {
      ostream << *iterator;
    }
    ostream << endl;
  }
  void deleteFromVector( string string )
  {
    for ( iterator = vector.begin();
      iterator != vector.end(); iterator++ )
    {
      if ( *iterator == string )
      {
        delete iterator; // now won't compile!
        cout << "deleted: " << string << endl;
      }
    }
  }
};
int main()
{
  VectorTest test;
  test.addToVector( string("1") );
  test.addToVector( string("2") );
  test.addToVector( string("3") );
  test.printVector( cout );
  test.deleteFromVector( "2" );
  test.printVector( cout );
  return 0;
}
```

Prizes provided by Blackwells Bookshops & Addison-Wesley

Bookcase

The latest roundup of book reviews.

C Vu has had, for a very long time, held the reputation and standard for top-notch reviews of programming and technical books. We have a great relationship with many publishers, some of whom publish quotes from the reviews on their literature (after getting permission of course!).

We are independent of any publishing company and as such have been known to slate books from one publisher and, in the same breath, praise another book from them. C Vu and its reviewers are respected for their impartiality and independent knowledge. However, looking at the number of reviews for this edition, it looks like the proverbial ink wells are running dry. We need more reviews!

Remember, if you submit a book review you are contributing to the greater knowledge of the membership. Books are expensive and the last thing anyone wants it to spend upwards of 30 pounds on a book which is an utter turkey! That said, if you decide to review a book, the worst that will happen is you lose a fiver – and if the book has the "Not Recommended" rating, your next book is free. What can be fairer than that.

As always, the ACCU must thank the Computer Bookshop, Blackwells and a range of other publishers for providing us with the review books.

Games Programming

Beginning Maths and Physics for Game Programmers

by Wendy Stahler (with Dustin Clingman and Kaveh Kahrizi), New Riders, ISBN 0-7357-1390-1, 454pp

Reviewer : Paul F. Johnson

Computer and Video Games courses at Universities and Colleges of FE are growing in popularity all of the time, yet most of them completely omit the maths and physics involved with any good game engine or system. This book hopes to set the omission right.

And set it right it does. Plenty of examples, good clear discussion and everything set out logically and best yet, it assumes very little. The maths is as user friendly as you will get and the physics uses "real" examples on how projectiles work, cars collide and other such events you would expect to see in any game.

Stahler doesn't shy away from the more complex areas of physics, such as 3D, but approaches them in a step-wise and easy to follow way. It is refreshing to see this in any beginners book, let alone one on such a complex subject. Wendy really does need a pat on the back for her efforts in this respect.

The book uses OpenGL to demonstrate the theory and if you know how to remove the Win32 material from the source, the value escalates tremendously. However, if you don't, the value isn't deminished (the theory still holds) just it may not convey the theory as well. It is hard to say as different people have different abilities to visualise how an aspect may look without physically seeing it. Unfortunately, from a pragmatic point of view, by not having the Win32 parts in either an external file or conditionally compiled, drags down the book.

Hopefully, Wendy will do something about this in the next iteration of the book.

Recommended.

C++ for Game Programmers

by Noel Llopis, Charles River Media, ISBN 1-58450-227-4, 404pp

Reviewer: Paul F. Johnson

If this book had been called C++ – an intermediate guide, this review would have been somewhat different. It isn't and as such, it's not a very pleasant review.

The problem is that the book has very little to interest a game programmer. The C++ contained in the book is pretty much your run of the mill, generic material. The code fragments are just that – fragments and in my opinion don't really re-enforce the points the author is trying to make.

Sure, there are bits on AI, games physics, communication and large objects, but these are consigned to the back of the book – less than 1/4 of the book covers these aspects. The rest, containers, smart pointer, linked lists – standard fodder.

Secondary to this is the book is unsure who it is aimed at. The book claims "intermediate" on the back – however, it's not. The STL is far better covered in Josuttis (the STL takes the majority of the book). It's no good for beginners as it's way to brief. It's no use to advanced bods as they will already know the material covered and it's not high enough for intermediates who should already know about linked lists et al.

If you're after a straight C++ book, there are far worse on the market – there are also far better.

Not recommended.

Games Physics

by David H Eberly, Elsevier, ISBN 1-55860-740-4, 744pp

Reviewer: Paul F. Johnson

This book is not for those with a weak constitution. If your maths is not up to muster, then don't go near the book – you plain won't be able to understand it. I didn't on the first read and I teach maths! However, if you are a games programmer and want to ensure your games realism is as good as real life, then you must have this book. It is fantastic!

For the uninitiated, games physics governs how a bullet arcs, how the speed drops over time and the likes of terminal velocity when it comes into contact with a non-air body. And that's at a simple level.

The maths in the book is enough to scare most, but thankfully, the explanations are clear and consise with all of the terms used defined. There are also plenty of code examples, so the cold maths become simpler for the programmer to

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Computer Manuals** (0121 706 6000)
  www.computer-manuals.co.uk
- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

understand (and use) as it is in a form they understand. The source is also on the cover mount CD.

The book steers clear of any platform issues (everything will compile on gcc, VC++, Intel C++ and nVidias cg compiler). It is purely for the physics.

Highly recommended

Game Programming in C++: Start to Finish

by Erik Yuzwa, Charles River Media, ISBN 1-58450-432-3, 378pp

Reviewer: Paul F. Johnson

I had high hopes for this book. Okay, 370-ish pages from start to finish is as insane as learning C++ in 21 days or Java in a weekend, but I put that to one side. This book is what I'd love to have on my shelf – covered OpenGL and SDL, two open source libraries, both widely used, both very capable. Then things went wrong; I opened it.

22 pages before anything of substance – it covered Inno, using CVS and setting up various bits on the machine. I can ignore that, it doesn't concern me, but it does take the page count down to about 350. Okay, still enough to give an over view of SDL...

The first code example fails to compile! Totally fails. Why? Look at the code and the description

```
// specify that you want to use
// objects defined in the std
// namespace
using namespace std;
int main(int argc, char
*argv[])
{

// define a string object. can
// also be defined as a
// std::string
  string text_string;

// instead of strcpy we can use
// the = operator
  text_string = "Hello world";
// instead of strcat we can use
// the += operator
  text_string += " I am a
     std::string";
// when needing a pointer to the
// character string buffer,
// always use the .c_str()
// method
  cerr << text_string.c_str()
     << endl;
  return 0;
}
```

It looks innocent enough, but it is assumed that the reader has little or no C knowledge at all (the book is for beginner to intermediate). What is this **strcpy**, **strcat** and what on earth is a **method** and **operator**?

Also, why doesn't it compile. That's right. The **#include <string>** is needed. This sort of mistake of not having the **#include**s or bothering to really explain what is going really doesn't help.

I can gloss over those. I have a bigger beef.

Remember I said it was for beginners – it launches very quickly into using templates. Me, I didn't hit templates for yonks while learning, yet the author seems to consider them fair game for beginners! If they had been written with care and with a lot of explanation, I could let it ride. This isn't. In fact, some parts are just wrong.

Up to around page 68, there isn't very much in there that would be considered programming, though the design chapter is quite good. Page 69 is where the book really stops being of use – it goes platform specific. A massive opportunity missed. From this point on, the source is very much Windows orientated and it becomes more and more less use to anyone not using Windows.

There is a section on dlls and interaction with COM. Why? The beginner wants to get something running and wants to link to the SDL or OpenGL libraries, not bother with their own – they want quick, simple and pretty.

Now, gentle reader, how many people think that they can master graphics programming maths? You know, 3D models, vectors, matrices – that sort of thing. Quite a few, yes? How many think that it can be covered in enough detail in 20 pages? Not a hope. The author should have used the pages for something more productive and had a line in about using something like "Beginning Math and Physics for Game Programming" – an entire book which uses the SDL and OpenGL, but covers the maths in a great deal more detail – and accuracy.

I could go on, but you get the picture. It's not a very good book. Given that one of the finest books on OpenGL (OpenGL Programming Guide, Vsn 2) runs to almost double the page count, covers everything in far higher detail and yet costs a shade under a tenner more (according to Amazon), it speaks volumes on what this book hopes to achieve.

Me, if the other half bought it as a present, I'd think she was having an affair and it was her way of telling me so.

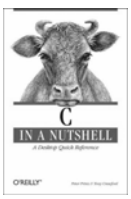
Not recommended.

C, C++ and C#

C in a nutshell

by Peter Prinz & Tony Crawford, O'Reilly, ISBN 0-596-00697-7

Reviewer : Christer Lofving



It does not happen very often, but it happens. You run into a technical book which takes the breath out of you.

Last time it happened to me was when I reviewed "Java I/O" by Elliotte Rusty Harold in the late 90's. Except for being fun to read at that time, it has also since then come to extensive use for me in a lot of real-world Java projects.

I think this aspect is that will raise an already good computer book to an extraordinary level; it's usefulness AFTER reading it! Also, on the book market today you can find a trillion titles on themes like "Java quick start" and "Teach yourself C#". (Even C++ has its good share here).

But there is a considerable shortage of books on C programming. In the bookshelf at home, my freshest pure C book has a printing year of 1994!

And a search on Amazon for "C programming" gives similar results.

The K&R classic "C programming language" as top hit, followed by a couple of titles from the 90's. So there is really a gap to fill. Good news are this title fills it indeed.

OK, almost all the "nutshell" books are of high quality, but "C in a nutshell" is outstanding even in that good company.

It has its 600 pages divided up and marked with "thumb marks" per section. A feature which gives it almost the same feeling as a theologian study Bible.

Part I describes the basics, and by that I really mean the basics. A beginner can handy use it from start as a step-by-step tutorial. Then he/she can work on with literals, expressions, functions and to real advanced topics like pointers. For the experienced user this "feature catalogue" of course is of great value as well.

The text is informative and intricacies are explained in clean English. And best of all; everything is illustrated with small, but fully functional code samples. This is an invaluable feature for any programming tutorial or reference book, because some lines of code (in form of a well-chosen example) often says more than many words.

This style is followed throughout. The most valuable part for myself was the complete, updated reference to the C standard library which makes up the core of the book.

I now feel up-to-date on the many enhancements added in the 1999 standard. The third and ending part "Basing tools" contains tutorials about using gcc, make and gdb. That makes it a complete, all-around C programming book.

Useful for the beginner as well as for the most experienced user to be found on earth. I am sure it will accompany me for many years to come.

C Primer Plus 5th ed

by Stephen Prata, Sams. ISBN 0-672-32696-5

Reviewer: Paul Thomas

Recommended.

Ah, C! This brings back memories. I had to learn C "on

the job" and what a confusing beast it was. It was years before I found time to learn the details and I was probably dangerous until then! It's always better to take the time to learn something like this properly and I could have done worse than work through this book. Though I expect it would have taken some time. The K&R "bible" has run out of pages before this book has covered the `if` statement.

I didn't do the exercises but I did give it a full read – a good 750 pages or so before the reference section. I learnt a few things too; mostly about C99 which the book focusses on. It does have the good grace to point out where compiler support is lacking and there is a good grounding in other practical matters. Not enough emphasis on non-portable behaviour for my liking, with bad advice on bitfields. But the teaching style is nice – introduce a new program, then follow it up with a lengthy discussion. Not just the basic language details either, but common use idioms.

I would leave it at that and recommend it for anyone committed to follow a large tutorial. But I have a few worries.

About halfway through, I came across this gem:

> So in serious programming, you should use `fgets()` rather than `gets()`, but this book takes a more relaxed approach.

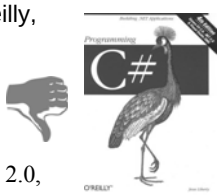
Relaxed approach? Your stock is falling Dr. Prata. Still, that's academics for you: wagging a finger at us for shunning garbage collection and formal verification; then writing FORTRAN libraries.

Another thing I look for in a basic C book is `goto`. Sure enough, it's not recommended and there is even a good section on how to replace it with structured code. But no satisfactory explanation for why it should be shunned. So students that learn from this book will likely rebel when they see how easy it is to use and will badger C/C++ lists to accept that it's a good thing. Probably quoting Knuth.

So, while other books might be better, it's a good tutorial for C that takes a complete novice into fairly advanced territory.

Programming C# (4th edition)

by Jesse Liberty, O'Reilly, ISBN: 0-596-00699-3. 644pp



Reviewer: Alan Lenton

This is the 4th edition of this book, and covers C# 2.0, .NET 2.0, and Visual Studio 2005. I originally bought the book late last year when I was trying to persuade .NET Windows Form to do anything other than produce programs which were database clients (I failed, incidentally). Having discovered that a week long visit to the dentist was probably less painful than using C++ to program .NET, I decided to learn C#, and bought this book to do so.

I found the book frustrating.

Yes, all the syntax is there, so are simple examples, and it's impressively heavy, but there is something missing. It took me quite a while to figure out what it was that was missing. If you want it in eastern mystic terms, the Tao of C# was missing. By that I mean it didn't really explain how to think in C# terms – how to build your program in a C# compatible way so that you weren't distorting C# usage to make it fit into a concept more appropriate to another language.

The failure is all the more severe because the book advertises itself as being for experienced professionals wanting to get up to speed in C#. These are exactly the sort of people who would need the conceptual information far more than an eight page explanation of how to do loops in C#.

The book would probably be OK for someone starting out to learn C# as their first language, although even then I'm not sure. And, come to think of it, I don't recall ever hearing of anyone who didn't learn C# as a second or subsequent language!

Not recommended, although, to be fair, I was unable to find a C# book that I would recommend.

C++ for Engineers and Scientists 2nd ed

by Gary Bronson, Thomson, ISBN :0-534-99380-X

Reviewer: Paul Thomas

Recommended with Reservations

Quite an interesting one this, it has a strange mix of the old and the new. Some parts of the language that are usually left to the advanced sections are introduced quite early as an integral part of the language. Function templates, for instance, are introduced at the same time as regular functions. Strings and streams are definitely not treated as add-on features. So the usual downfall of C++ introductory texts is avoided.

The presentation is impressive. The material is moved forward in a way that's bound to keep you interested. The Engineers and Scientists appear in the title because that is the focus of all the examples and exercises. It certainly makes a nice change from bank transactions and if this is your area then I imagine it will hold your interest much more.

There are a few technical inaccuracies that let the book down and this combined with the material that's missing gave me the impression that the author wasn't as fully conversant with C++ as you might hope. As an example, the section on cast operators was misleading and plain wrong in places. But a few minor problems like that shouldn't detract from such a nice tutorial text.

I'd recommend it to anyone needing to cross over from the harder sciences, but on the understanding that it is not really suitable as a reference and should be read in conjunction with something a bit more meaty.

C++ and Object-Oriented Numeric Computing

by Daoqi Yang, Springer, ISBN: 0-387-98990-0. 440pp

reviewed by Francis Glassborow

I have to be honest and tell you that I get irritated by this kind of book. The title starts the problem for me. You would expect (or at least I would) a book that starts with the assumption that the reader is familiar with programming and C++ in particular. You would be mistaken.

This is an introductory text on C++ aimed at scientists and engineers. That would be OK if the book tackled programming in C++ in a way that is appropriate to what I call the incidental programmer – someone who needs to program as part of their work or profession but who is a professional in some other discipline. Such people need a good clear introduction on how to express the problems of their discipline in C++. They need the full power of the high-level abstractions provided by the Standard Library (and should be told about both Boost and other specialist quality libraries).

What these people do not need is a tedious introduction to low level C++ with a lot of emphasis on its C ancestry.

Let me give you an example of what sticks in my gullet. The author wants to spend some time on operator overloading so he writes a chapter on implementing classes for complex numbers, vectors (mathematical ones) and matrices. The first of these is already fully designed and implemented in the Standard Library complete with specialisations for all the floating point types and facilities for mixing the different specialisations. What we find in the book is a partial implementation of a complex number type with doubles used for the real and imaginary parts. There is no discussion of the reasons for the design. For example, the author correctly implements operator **+** in terms of operator **+=** but give no explanation.

Later, in chapter 7 (on templates), he gives the briefest of introductions to the Standard `complex<>` types.

Perhaps I should not go on, because I clearly have no sympathy for the author's approach. From my perspective the material is too low-level, too skimpy, lacks explanation of either how the implementation works or how the reader might use C++ in his/her discipline.

This book is clearly ill-suited to the ordinary novice and, in my opinion, does not tackle the use of C++ by mathematicians, scientists or engineers. There is a place for a good clear introduction to numerical programming using C++ but this book is not it

Python

wxPython in Action

by Rappin, Noel & Robin Dunn, Manning, ISBN: 1-932394-62-1, 552pp

Reviewer: Ivan Uemlianin

Recommended

This is a likeable book, recommended without reservation to anyone getting started in python GUI programming. n.b.: the book says virtually nothing of use on the relationship between wxPython and wxWidgets.

The book's three parts introduce in turn wxPython and GUI programming in general, the basic wxPython widgets, and then more complex wxPython stuctures. Each part has six chapters. At the next level down the book is organised as a FAQ: each section or sub-section title is phrased as a direct question: 'What terminology do I need to understand events?', 'How do I create a pop-up menu?', etc.

The authors are serious about GUI programming and good design, and about bringing these to a new audience. The writing is clear and steady if a little earnest, as if an effort has been made to avoid intimidating budding GUI pythonistas.

The FAQ format makes it very easy to use as a reference but, apart from the first section, it gets too tedious to read as a book. This effectively works as a gentle nudge encouraging the reader to try a few things out and come back when they need help.

Although the book has an introductory feel, and a lot of time is spent on foundational issues, it still covers a reasonable amount of ground, up to things like using tree controls, html displays, printing and drag-and-drop.

The book disappointed me (majorly) twice.

First, there are virtually no complete applications (I counted one, a basic sketchpad). There are many working scripts, but these invariably demonstrate a particular widget or behaviour. Compare this with Python and Tkinter Programming by J.E. Grayson (2000), which offers a wide range of often quite spohisticated GUIs, including AppShell.py – a kind of GUI template which the reader can 'fill in' and customise as required.

The authors seem completely unaware of this earlier book on Python GUI programming; their publishers certainly can't have been. Rappin & Dunn (among many others) claim that wxPython improves significantly on Tkinter but, in the absense of any GUIs approaching the sophistication of those in Grayson (2000), they do not demonstrate their case.

Second disappointment: the book contains no discussion at all of SWIG or wxWidgets. wxPython is essentially a Python wrapper around wxWidgets, and it uses SWIG extensively to provide this wrapper. How simple is it to port a wxPython GUI to wxWidgets? Should this be a factor in choosing python GUI toolkits? Practical issues like these remain unaddressed. Although slightly tangential, an illustration of how SWIG is used to generate wxPython from wxWidgets would have made a valuable appendix.

However, wxPython's use of SWIG and wxWidgets is mentioned only in passing, and the book proceeds as if it were of no practical interest.

I found these two ommissions extremely annoying, but they are missed opportunities rather than serious errors. Notwithstanding my disappointments I found the book very useable. I recommend it as dead tree format documentation for wxPython, or for python programmers who want to get started on GUIs.

The book can not be recommended for people who want to learn about the relationship between wxPython and wxWidgets.

Java

The Java Programming Language 4th ed

by Ken Arnold et al., Addison Wesley, ISBN : 0-321-34980-6

Reviewer: Paul Thomas

Highly Recommended

Now this is more like it! A proper language text with dense, value for money, trustworthy content. None of your namby-pamby tutorial here, lots of caffeine is needed to read it. Java is often dismissed (mainly by C++ programmers) as a language written for children frightened by the sharp edges of a real language. They should have a skim through this and see how much is actually involved.

The book is actually billed as a tutorial, but those parts are very light. If you need a gentle worked pace, then you won't get far with this one. It starts with a quick tour to give you a feel for the language, then methodically moves through just about every aspect of syntax, primitives and core library classes. As the material moves toward the (huge) class libraries, it becomes more reference-like with lists of selected class methods. At just about every point, the reference material is mixed with rationale, advice on common usage and examples to place it in context.

This is a no-nonsense description of the language that can help you get up to speed very quickly – particularly if you are familiar with similar languages. It also has enough depth that you can be confident not to have missed something. It won't help if you are new to programming. Make no mistake, this is programming for grown-ups and my copy is likely to become worn through heavy use before long.

Pocket PC

Pocket PC Network Programming

by Steve Makofsky, Addison-Wesley, ISBN : 0-321-13352-8. 656 pp

Reviewer: Frances Buontempo

This book covers network programming Pocket PC 2002/ 2003 and Pocket PC 2002/2003 Phone Edition devices, though some of the APIs may be available on a Windows CE 3 or 4 device and much of it applies to PCs as well. The contents include Winsock, WinInet, Internet Protocol Helper API's (IPHelper), Network Redirector, Serial and Infrared connections , Remote Access Services, Connection Manager, Pocket PC 2002 Phone Edition, Desktop Synchronization, and the Pocket Outlook Object Model, Email (MAPI). The final briefly chapter covers the compact framework: VB.Net and C# for Pocket PC. However, since it is aimed at earlier versions of Pocket PC, it does not cover Bluetooth.

This book pulls together a wealth of items, some of which are covered in disparate places across the MSDN making them hard to find without knowing what to look for. It adds extras such as warnings and error cases to be aware of, alternative ways of doing things and giving a brief overview of networking concepts such as the TCP/IP stack etc.

On the plus side, it gives a thorough coverage of networking ranging from Desktop Synchronization to more detailed items such as using HTTP programmatically, communication with other devices via IrDA, telephony and the RAPI, and with networks. It includes lots of code samples and usually demonstrates various ways of doing things, though not always clarifying pros and cons of each method where alternatives exist.

The code samples are mostly in C. Their standard is sometimes poor: magic numbers all over the place, including the same constant repeated over three times in one code 39 line code sample...neither an enum nor a `#define` in sight. Furthermore, it often says to set certain parameters to API calls to NULL with no explanation. More background or depth would be good from time to time. However, it is enough to show which functions should be used in which order to do what.

This book is complete, thorough and relevant to more than just Windows Mobile devices, but has some nasty code samples and a lack of advice about when to use what. In addition throw away statements are made without explanation from time to time such as page 307: Each [SMS] message can be up to 160 alphanumeric characters long (140 bytes). Excuse me? Oh... 7 bit chars...right. There are several throwaway comments like this that could do with some

explanation. Finally, some of the code samples are troubling for an entirely different reason. In particular, `InternetSetOption` is proferred as the way to set timeouts when using HTTP. According to the MSDN this has a known bug: `InternetSetOption` with timeout values doesn't work.Calling InternetSetOption (or MFC CHttpFile::SetOption) with INTERNET_OPTION_SEND_TIMEOUT or INTERNET_OPTION_CONNECT_TIMEOUT does not set the specified timeout values. (See Q176420 support.microsoft.com/support/kb/articles/Q176/4/20.ASP)

Recommended with serious reservations. If you buy it, it would provide a good starting point, but take everything it says with a pinch of salt.

Programming

The Art of Computer Programming V4 Fascile 4

by Donald Knuth, Addison-Wesley,
ISBN: 0-321-33570-8. 120 pp

Reviewer: Francis Glassborow

This is another part of volume four of *The Art of Computer Programming*. This latest fascicle (by the way, at the time of writing I am not aware of the publication of fascile 1 of this volume. The fascile 1 you may see in your local bookstore/shop is for volume 1) has bwcovers the generation of all trees. Closer examination revealed another aspect of this curious way of dragging text out of a perfectionist author (The author has been working half a lifetime on this volume and still has volume 5 to come). You cannot simply put the fasciles together and get the finished volume because each fascile has a coherent extract from the projected work but not consecutive. This fascile refers back to earlier fasciles.

If you know Donald Knuth's work, you do not need me to tell you of its authority and readability. The author is without doubt one of the leading figures of the computer world and has been for over 30 years. The earlier volumes have stood the test of time, though they have been revised and updated.

If you do not know The Art of Computer Programming this is not the place to start, get the first three volumes and study them carefully. You will then be ready to dip into the fourth volume either via the drafts on the web or by getting the fasciles as they become available. It is probably unwise to wait for the completed volume as this writer has grown old waiting and has had his hopes raised several times over the last decade.

General Interest

What the Dormouse Said

by John Markoff, Penguin,
ISBN: 0-14-303676-9, 310pp

Reviewer: Alan Lenton

I've been waiting for this book to come out in paperback ever since I read an interview with John Markoff in the ACM's online magazine, Ubiquity! (http://www.acm.org/ubiquity/interviews/v6i29_markoff.html)

The perceived wisdom about the genesis of the Internet is that while hippies and lefties were out on the streets protesting about the Vietnam War, a small cadre of 'all American' engineers was busy laying the foundations of personal computing and ARPNet, the Internet's precursor.

I always thought this had to be wrong. You only have to look at the ethos of the early ARPNet/Internet to see how hippy and power-to-the-people-ish it was. Even today, after decades of commercial activity, the battle still rages. Technically, the architecture of Internet remains fundamentally that of peer-to-peer, even though the majority of major applications are client/server oriented. As for the open/closed source program dichotomy – the battle is, if anything, fiercer than ever.

This didn't all happen by accident. As John Markoff's remarkable book shows, the personal computer and networking revolution was a product of US society on the West Coast in the late 60s.

The truth is that the architectural ideas and much of the technical work on things we take for granted about personal computing and the Internet came from people who were part of the San Francisco scene, they took drugs – including acid – and in many cases were real hippies.

To give just one example of the depth of the links, one of the camera operators at the first ever public demonstration of cyber space on December 9th, 1968 was Stewart Brand, soon to become famous as the creator of the Whole Earth Catalog!

The book covers the period from the start of the sixties through to the infamous Bill Gates letter denouncing members of the Homebrew Computer Club for 'stealing' Gates's version of Basic for the seminal MITS Altair personal computer.

I really recommend this book for those who would like to find out the whole story of how the technologies that came together to make the network enabled personal computer came int existence. It may well be that personal computing and the Internet turn out to be one of the most enduring legacies of the 60s hippy movement!

Oh - and just what was it that the dormouse said?

> 'When logic and proportion
> Have fallen sloppy dead
> And the White Knight is talking backwards
> And the Red Queen's "Off with her head!"
> Remember what the dormouse said:
> Feed your head!
> Feed your head!
> Feed your head!'
> (From the song "White Rabbit" by Jefferson Airplane, 1966)

Skype: The Definitive Guide

by Harry Max & Taylor Ray, Que, ISBN: 0-321-40940-X. 263 pp

Note by Francis Glassborow

This is one of those books that you do not really need but some people are happier with a printed book rather than relying on the web. I am not sure that is the right way to go because Skype is still developing and so any book is out of date before it gets to the shops.

Nonetheless if you want to use Skype you might find having this book to read when you have a moment (such as when travelling to work) might open your eyes to the potential Skype has beyond being just useful for Internet telephony.

One of the things I like about Skype is its support for such things as SMS messages (text messages to mobile phones). I can quickly send a text message from my laptop when it is connected to the Internet.

If you are not already familiar with Skype, it is well worth trying it out and its basic features from computer to computer are free. Even the computer to phone connections are relatively inexpensive.

If you are a Skype user, you might browse this book in your local bookshop to see if it meets your needs, but there is nothing here that you will not be able to find for yourself by reading the information provided by its website.

Crabby Office Lady

by Annik Stahl, Microsoft Press,
ISBN: 978-0-7356-2272-2. 178 pp

Note by Francis Glassborow

Let me make this one simple (after all it is not about programming); load up your chosen browser and Google 'Crabby Office Lady'. The book is based on Annik Stahl's Internet columns.

Yes, it is fun and despite the publisher (Microsoft Press) shows an appropriate lack of reverence for his employer. ■

ACCU Information

Membership news and committee reports

View From the Chair

Jez Higgins
chair@accu.org

Somebody asked me recently what it felt like to have my hands on the levers of ACCU power. I laughed. There are no levers, not even a small ones. ACCU is not a ship, at whose wheel I stand, guiding the organisation smoothly and safely through whatever waters this particular failing metaphor might take us. That's entirely as it should be. We're a voluntary organisation, and everybody is here because they want to be. The Chair doesn't dish out orders and crack the whip. I'm not good at that kind of thing, anyway. The nearest I've come in the past to any kind of "command position" was as a patrol leader when I was a scout (1st Tacolneston, now sadly defunct). Organising a group of 14 year olds relies very little on bossing around and very much on encouragement, suggestion, nudging and negotiation. I was pretty good at it, actually. Being Chair is turning out to be kind of similar, but with more beards.

First in line for an activity badge is Ewan, who took a combined conference committee and ACCU committee meeting to a look at a potential new venue for next year's conference. The conference venue is a subject that evokes strong emotion, and is something everyone on the conference committee takes extremely seriously. We were given an extensive tour of the facilities, which were really quite impressive, and really rather too much lunch, which was also rather good. We checked out the car park too. That's right, car park. I have to be slightly coy about the specifics at the moment, but hopefully Ewan will be able to say more next time.

Badges too for Allan, Tim and Tony, who continue to work away on the website. Much of the work isn't yet immediately visible, but going live as I type this are the ACCU blogs - technical weblogs by ACCU members. There's much more too, I suggest you speed over to Allan's report for the details.

Finally, a badge and many thanks to Paul, who is standing down as C Vu editor to spend more time with his newly enlarged family and newly enlarged job. He's done sterling work over the last two and half years. On behalf of the whole of ACCU I'd like to thank Paul for his commitment and effort. On a personal note, I want him to know that it is very much appreciated, even if I am routinely late with this report. Taking over the C Vu reins is Tim Penhey, who I'm sure is going to do a fine job. Editor – now that is a job where you get to crack the whip ...

Secretary's Report

Alan Bellingham
secretary@accu.org

The May committee meeting took place, unusually, at a hotel. However, there was good reason for this: the hotel is a candidate venue for the next AGM and, parenthetically, the next conference.

(Having managed to be at the last two AGMs without actually attending the conferences they were attached to, I may be a little one sided. One was missed due to an injured back, the other due to a business 'crisis', and I hope neither reason is repeated.)

The Oxford Hotel is a compromise between the constricted central Oxford location of the Randolph and the out-at-the-edge nature of the Holiday Inn at Peartree. It's a sister hotel of the Hinckley Island (at which I will be attending a convention in late August), and the Paramount Hotel group aspires to a casual stylishness which I find more attractive than faded grandeur. The food was better than the Randolph, it's much more car-friendly, and if the conference committe go with it, I for one would be happy.

But on to the meeting.

Most of the usual bunch were at the meeting, but we also had in attendance Giovanni Aspronni, there for the first time since his election, as well as Tim Penhey.

The minutes were checked and approved, and officers' reports were received.

The first major issue we discussed was whether the handbook should still be published on paper, or whether it should more sensibly appear on the website. The conclusion was that although the latter option is attractive, we shouldn't just go ahead and do it without consultation from the members. So, for this year, it will still be printed, but we may well move to go online next year. We will need to guard against address skimming and the like.

Following this, we considered the future direction of these journals. With the arrival of web publishing, with a lower cost of delivery, is the continued existence of paper magazines still necessary? There was, as you would expect, considerable discussion here. With the arrival of the new website, can we go web-only? If so, should we trickle articles out as they arrived, or should we stick to a known publication schedule? In the end, we decided that we would start by publishing Overload on the website at the same time as the physical copies were posted. This will mean that at least far-flung members will have a chance to read articles at the same time as UK members, and online discussions could be less disjoint.

Another issue, again on publishing (we seemed to have a strong theme for this meeting) was that with our C Vu editor retiring, we need a replacement. Overload has been running for a while with a team structure, and this seems a good opportunity to do the same for C Vu. As a result, we have asked Tim Penhey to recruit such a team for C Vu.

And then, changing the subject, we finished with two related issues. The first was the result of a request from the WG21 ISO panel that the UK should host the Spring 2007 meetings. This has been done before – it's convenient for the meetings to be just before our conference, so that speakers can arrive for the panel meeting and then do the conference as well. However, hosting such a meeting takes money, and without money it won't happen. Therefore, we need sponsors. If you know someone, whether inside Sun or Microsoft or Google, or perhaps a smaller organisation, that might be interested, please contact Lois. No amount too small!

Finally, we talked about support for local meetings. We've long been impressed by the regular meetings that the Silicon Valley branch has been holding, and though the annual conference is very impressive, smaller local meetings haven't taken off in the same way. We reckoned that seed funds for local groups such as Nottingham or similar could well help get them going, and agreed that, given a suitable case, we would provide them.

Website Report

Allan Kelly
allan@allankelly.net

Just over two weeks ago we launched the new ACCU website! I expect most of you have already visited the site, and on the whole comments are positive, but just in case you haven't, check out the new www.accu.org.

The site was finally launched on 14 February, it is now the start of March and we've had just short of 130,000 page visits. Incredible I know, some of them will be bots but still, that is a lot of visitors.

I have to say a big thank you to those most closely involved: Tony Barrett-Powell the ACCU web-editor and Tim Pushman of Gnomedia who have done most of the work. Jez Higgins has been a great help in the past few months, additional thanks for various supporting work go to Alan Lenton, Ian Bruntlett and Paul Johnson.

We also have a new book database system. This has been developed by Parthenon Computing and is linked to bookshops and carries adverts. Parthenon will be paid from the revenue generated with any extra revenue be split between the ACCU and Parthenon. So, if you are buying a book please buy it through the site.

The new site isn't the end of the story. We still have work to do. The whole point of redeveloping the site and installing a new CMS system was to allow us to keep the site up to date and us it as a new journal media.

Still, there is pressing work to do, we need to move the US ACCU site over, mailing lists, mail archives and journals have yet to be moved from the old server. And there is more.

Once again, thanks to all those who have helped.

Since the conference there has been a lot of talk on the committee list about creating local groups – inspired by Reg Charney. One of the things we've started thinking about is creating an online calendar to track all these events.

Hopefully by the time I write the next report we'll have some new stuff to talk about.

I just wanted to say you were fantastic!

And do you know what? So was I.

Yep, it had to happen. My time in this position is fast drawing to a close and like the proverbial timelord, you won't be seeing me again, well not with this silly ol' face.

I've throughly enjoyed editting C Vu. I remember my first issue was one that I took over in mid production run – it was a very strange experience, but not an unenjoyable one.

The magazine has seen quite a number of changes in personnel as well as the new website and the redesign. All of them have made C Vu a stronger and more professional publication. The stall has been set, and my heart felt wishes go to our next editor. Good luck Tim. Don't let us down!

Going soon. It's time to say goodbye... Might regenerate. I don't know. Feels different this time...

Learn to write better code
Take steps to improve your skills
Release your talents

ACCU
PROFESSIONALISM IN PROGRAMMING
JOIN : IN