The magazine of the ACCU

www.accu.org

{cvu}

Volume 18 Issue 5 October 2006 £3

{cvu}

Volume 18 Issue 5
October 2006
ISSN 1354-3164
www.accu.org

Editor
Tim Penhey
cvu@accu.org

Contributors
Ian Bruntlett, Mark Easterbrook, Francis Glassborow, Lois Goldthwaite, Pete Goodliffe, Roger Orr, Tim Penhey

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
Thaddeus Froggley
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

The Winds of Change

In case you have been living in a cave and don't know what's going on, C Vu has a new editor. The last issue was Paul's last and I'd like to thank him for all the hard work and time he invested in ACCU. I guess I'll find out just how much time it all takes in the coming months. When the editorialship was first mentioned to me I thought "Ha ha, yeah, right". I had seen Alan Griffiths (editor of Overload) article hunting in action. It looked quite stressful. However, I found that I couldn't get the idea out of my head. The more I thought about it, the more I liked the idea. I thought that this was something that I could do to really contribute to the ACCU community.

I was out at lunch yesterday with two other ACCU members and one's work colleague. Throughout the course of conversation I asked him if he was an ACCU member. Disturbingly he said "No, I'm not qualified". Somehow he seemed to have got the opinion that ACCU is for the elite only. Perhaps the only ACCU members he knew were at the top of their field? I don't know. However, I did set him straight on that point. I see ACCU as especially good for those who are not yet elite but want to get better. This does highlight a small problem – "How do we get the ACCU message out there?" I'm sure there are many potential members out there who just don't know we exist. By comparison, how many people have seen the BCS adverts on The Register website?

Many, if not most, of us are extremely busy professionals with little time for keeping up with technologies outside our own arena. I see C Vu as a source of information about languages or tools that are outside your normal use. A place to get an overview, or sample usage, that may spread the use of interesting technologies.

I would also like to take a moment to apologise to you, our readers, and to Francis Glassborow. You may have noticed that Francis' Scribbles was missing from the last issue. This was due to an unfortunate editorial oversight and I feel compelled to apologise publicly. So, sorry to you, and sorry Francis. The missing column is in this issue along with an extra addition that Francis has penned.

I'd like to thank all the contributors and reviewers for this, my first edition as editor. Here it is, I hope you like it.

Tim

TIM PENHEY,
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 18.6: 1st November 2006
C Vu 19.1: 1st January 2007

IN OVERLOAD

This month in Overload, you can read: 'Up Against the Barrier' by Simon Sebright, 'Inventing a Mutex' by George Shagov, 'C++ Unit Testing Easier: CUTE' by Peter Sommerlad and 'From CVS to Subversion' by Thomas Guest.

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

More Real Life Experiences of a Software Engineer

Simon Sebright on finding work.

Some time ago, I wrote in C Vu [1] about my experiences in the job market after being made redundant. I thought I'd continue the story where I left off. Not so as to bore you all with my own personal history, but to hopefully strike a common note with those of you who might have been in similar situations. I won't be patronising and say that it'll be alright for you, just wait and see. No, au contraire, your guardian angel may have nodded off, or you might just not be very lucky.

I had applied in vain for several jobs. Every two weeks, I had to go to the job centre in the local town and sign on. After six months, the money runs out, but you can still get national insurance contributions paid in the UK. It was very demeaning. Every time, the consultant, or whatever they are called, would do a search for jobs in my registered fields. Every time, they drew a blank.

I usually came prepared with a printout of all the online applications I had made, and email applications, etc. They never bothered to want to see it, so I became a little lax with it all.

But, one time, a single job vacancy did appear. I nearly fell off my chair when they told me the company – the one I had worked for in my first position as a software engineer several years ago.

Of course, I had no choice but to apply for the job. I said that I had worked there before, and would naturally contact them. Still, they wanted to be sure and rang the company.

I phoned them myself later in the day and spoke to my previous boss. I had emailed him a couple of times after being made redundant, but having no reply, decided not to continue bugging him. I did think it strange, but you never know. It turned out that he had switched to a different email address as he was getting too much spam, but he didn't disable the old account, so I never got a delivery failure. He was surprised to find me out of work and said, yes, they had a vacancy and got very excited about the prospect of having me back again.

Here we go

I became very excited too. Somebody up there wanted me to get back to work, and not too far away in a nice cosy town. Great! I was to head over later in the week for a chat (interview as far as the job centre was concerned). The day came, and I drove over, looking smart but casual. I spent a fair time explaining what I had done in the meantime and trying to impress on them the value of my experience in other companies, as well as joining ACCU and other personal developments. It all went swimmingly well, and I left with a spring in my step, waiting to hear by the end of the week if they would offer me anything concrete, having already mentioned salary requirements.

The end of the week came and went, and I dismissed it as the usual semi-chaotic nature of the boss there. So, I rang the next week to see what conclusion they had arrived at. They had decided that I was too experienced for the position they had in mind. There was already at least one very experienced person there (whom I had in fact helped to mentor). I guess they meant I was simply too expensive.

The phone call had a that negative sinking feeling about it until he mentioned the word "contract" in the sense that they had assumed I was only interested in a permanent position, given I had a house and family, etc. to look after. No, I said, anything will do! So it was decided they would have another think at their end if there was a parcel of work I could do for them.

Yes, there was! I effectively accepted the three month contract in the next phone call, to start ASAP. Phew, some money to bring the overdraft back into the black.

Dilemma

However, the next day I had a phone call from the agent through which I had applied for a contract, at a company we will call a large British company in the aerospace sector. They were going to offer me a three month contract, with possibility of extension. A bit more money too, what to do now? I tend to play things fairly straight, and I try not to let people down. I explained to the agent that I had the opportunity of a contract ASAP, could the other company wait a bit?

No, was the reply, 4 weeks tops to allow for the security clearance process to take place (an extra bonus point for that, too). I rang the local company boss and explained the situation. He sensibly said that I had no real choice but to take the other offer. However, we agreed that I would do a lesser bit of work for him in the 4 weeks, and would work longer hours and weekends to do it.

Somebody up there wanted me to get back to work, and not too far away in a nice cosy town.

Thus, I had gone from no job to two in the space of a week. I had a hectic month, August if I remember. It was very hot, for the UK, and I didn't care that the old car didn't have air conditioning. I got the job done, refactoring some code I'd written for them several years ago to use as a starting point for a conversion utility between a standard format and their own internal file format.

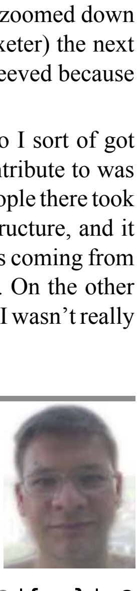
One down, one to go...

And so, after a weekend of wrapping up the little project, I zoomed down the M5 (a British motorway going from Birmingham to Exeter) the next Monday morning full of anticipation. Oh, and I was a bit peeved because I now had to wear a tie.

On the first day, my team leader was not actually there, so I sort of got looked after by another. The software product I was to contribute to was part of a large communication system. There was a lot the people there took for granted about the program and the surrounding infrastructure, and it was a steep learning curve, with most of the eureka moments coming from my own endeavours, rather than any purposeful guidance. On the other hand, working for such a behemoth of a company meant that I wasn't really expected to do much for some time.

SIMON SEBRIGHT

Simon has been programming for 10 years, mainly in multi-tier C++ application development. Recently, he has been designing and developing web/database-based applications using C# and asp.net. He can be contacted at simonsebright@hotmail.com



On Information Poverty

Ian Bruntlett asks: Are you information poor?

An article of mine, "On Killer Apps", appeared in C Vu February 2006 (18.1). If you haven't read it, I recommend you do so – it's only half a page so won't take long to read.

In particular I suggested that in order to produce a killer app then two conditions (which I have now named *Killer App Fundamentals* for convenience) need to be satisfied.

1. A fertile technology base.
2. The economic opportunity to shine.

I'd like to point out that the *Killer App Fundamentals* are based on my personal perspective and, say, aren't universal truths.

The Killer Apps article generated some queries from Kevlin Henney (KH), the technical reviewer:

One question I had, though, was hinted at but unanswered, from your perspective. You mentioned a relationship between success and failure, pointing out that you keep working throughout the failures without noticing them. The killer question is: do you recognise the failures or genuinely remain blind to them? I think this is actually a critical (quite literally) point. What role does that knowledge or blindspot/denial play?

There are known knowns;
there are things we know we know.
We also know there are known unknowns;
that is to say we know there are some things we don't know.
But there are also unknowns;
the ones we don't know we don't know.

Donald Rumsfeld

"…do you recognise the failures or genuinely remain blind to them?"

How does one recognise failure? People with mental health problems are said to possess insight if they realise that things are going wrong. Similarly, I.T. staff working on a project can either possess insight or be in denial when things are going wrong. If one is information poor then you increase the risk of being in denial. Even if, with insight, you do recognise that a project is failing, it is not always obvious what the problem or solution is – success has many parents and failure is an orphan.

Sometimes the degree of success a project has/had can only be assessed after time has passed, when one learns new things, especially of information poverty.

"a relationship between success and failure" - information poverty

Lack of insight caused by being information poor (bad or non-existent text books, lack of knowledge about the ACCU or similar organisations, not having accu-general to get help from more experienced developers) – results in projects failing, despite the best of intentions.

IAN BRUNTLETT

Ian has been involved in broad spectrum of software systems and languages. He works as a volunteer, teaching people with mental health problems how to use and program computers. He can be reached at ianbruntlett@hotmail.com

More Real Life Experiences of a Software Engineer (continued)

Contracting options

I had asked a few friends what they would advise to do about getting paid for the contracts. Most seemed to be seasoned contractors, and had their own companies. As this was potentially only short, I didn't want the hassle of such an arrangement, so opted for an umbrella company for the first month with the old company, and then a managed company scheme. The agent had recommended a couple of companies of accountants he had dealt with. I made my choice on the basis of the perceived competence and politeness of the first people I spoke with from each. So, I ended up with a particular firm who had a scheme where I would have my own company, but they would (for a fee) set it all up and administer it, and pay me.

Switching from the umbrella scheme they had to the individual company scheme didn't go all that smoothly. During one phone call I had, they said that they had just been discussing my case as an example of how to manage the transition from one to the other. In other words, they knew it wasn't working.

After a couple months of late payments, the whole thing got going, I just filled in an online timesheet on the agency's website. This was then authorised by my line manager. I still had to fill out a paper form and fax to the accountant company, who would then invoice the agency, who…

Extensions

Everybody had been saying that I would of course get an extension with such a big company. People remain contractors for years. That didn't help when I was sitting at my desk just before Christmas not knowing for sure if I would be here in the New Year.

I was fortunate enough to be extended, first by another three and then six months. A year or so later, some contractors were not extended.

End game

I was extended, however, and spent the best part of 20 months there before I myself handed in my notice and moved over to Germany, where my wife and children had gone some time before me. It was somewhat ironic. She (herself being German) had wanted to go there to become a teacher (there's more to it than that, all German bureaucracy) to secure our family's income after my redundancy, which started all this. And now, I had to give up a reasonably lucrative position, and go and seek my fortune on the German/Swiss border.

That's for another time. ■

Notes and references

1. "On not being a software engineer", *C Vu*, August 2003, Volume 15 No 4.

I talk about programming failures, the same ideas could be applied, I believe, to design failures

Sometimes I noticed the problems early on but decided to carry on coding anyway – because I knew of no other way to react – I was information poor from a project management point of view.

Immersion vs information poverty

I think I've got a point here: immersion is the other side of the coin marked "information poor". It is possible for an information rich project to fail but that failure should be caused by other problems – lack of a common consensus (analysis paralysis / disruptive politics with the key stakeholders) or poor design. The difference between being immersed in the river instead of being inundated is the difference between swimming and drowning. The trick is knowing how to swim so that you don't drown.

The internet age: information wealth, attention poor vs information poverty

I suggest there's more than one type of information. There is the information that tells you about something (*informative)* (e.g. The Linux Documentation Project) and then there is the stuff that separates the wood from the trees (*directive*) (e.g. the ACCU's book reviews). For years I have relied on the ACCU book reviews to decide which books to buy. While reading earlier versions of this article, Kevlin Henney found a quote which tackled matters from the information wealth point of view:

> *What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention, and a need to allocate that attention efficiently among the over-abundance of information sources that might consume it.*
>
> *Herbert Alexander Simon, economist, Nobel Laureate (1916-2001).*

Other good quality directive sources of information would be *Google News, The Register, Slashdot, The Linux Weekly News* etc.

The *On Killer Apps* article has changed things for me – I now have greater insight in how some things work and why some things didn't quite work. Once I wrote the article I essentially had a revelation – re-evaluating the past, with greater insight. At night I would run downstairs to write extra notes – the article literally made me lose sleep.

While I talk about programming failures, the same ideas could be applied, I believe, to design failures. An information-poor programmer can be handed a textbook. An information-poor designer lacks maturity as well as information – this is not so easy to cure. Once you're reasonably safe with the technology immersion and programming, the other low hanging fruit on the tree are 1) analysis and 2) design. Membership and participation in ACCU have helped me a great deal with Analysis, Design and Implementation of systems. I wrote the Overload September 2000 article on *User Defined Types* because the laws of physics broke down for me in 1999 (first major psychotic episode) and I wanted to document my abilities so that if it happened again I would be able to restore my IT skills from "backup".

Being information poor kept me on a capability plateau, peering over the edge with my hot SuperBasic and 680x0 macro assembly skills asking – is this all there is to software development? With the 1) Linux Kernel books, and 2) the inclusion of GNU C/C++ compiler, bash, sed and awk, Perl, Python with the Linux operating system I now know that there's a lot of things out there.

immersion is the other side of the coin marked "information poor"

For failures, you're experiencing problems in the project. The question becomes: is this a manifestation of the usual IT project problems or are we heading to a project failure? The *On Killer Apps* article raised the question of have you immersed yourself in the technologies required or are you facing a technology gap? It is no good immersing yourself completely in the technology base, systems analysis or design of the project – as well as entering in the river for a length of time, you need to be able to swim. The IT swimming requires taste and skill – unlike the physical activity of swimming.

You and I are essentially immersed in state of the art C or C++, through membership of the ACCU, reading books (recommended by the ACCU) and attending conferences (hosted by the ACCU) – noticing a trend here.

You can immerse yourself in the C/C++ books held in your local library. However the *quality* of the library's information will not be as high as the ACCU's information.

an information-poor designer lacks maturity as well as information – this is not so easy to cure

One example was when I was using the Psion Archive database 4GL. I could write short applications OK but 1) I had not been introduced to procedural decomposition (weak coupling and high cohesion) and 2) IIRC Archive only had procedures, not functions. The resulting tar pits did not cause commercial damage – they were experiments at home – but it made me realise that I had a gap in my knowledge – in particular that big projects were much more risky than small projects.

Another example was my first job as part of my university's sandwich year. I was offered a job by Grand Metropolitan Brewing's Systems department, based on two weeks' experience at University writing a system with other people. I was a bad choice because I 1) had no idea how to work as a team 2) only had two weeks' dBase 3+ experience at Uni – without access to the manuals. My project leader never really understood why I seemed to spend loads of time pouring through the Ashton Tate manuals. And I didn't realise just how bad I was until decades later.

To conclude

Enough about my experiences. Reread the *On Killer Apps* article and write up your experiences for C Vu. ■

Complacency in the Computer Industry

Julian Smith has something to get off his chest...

I think the computer industry (both closed and open-source) is exhibiting an astounding lack of understanding about the serious problems that plague it. There is a passive acceptance of complexity and duplication that makes working in this industry deeply unsatisfying.

An example: C++. A while ago, I decided to stop attending the UK C++ panel's meetings, because I couldn't see anything fundamentally new being done. Instead, adding fairly simple features (such as initialisers or chained-constructor) to C++ was requiring enormous amounts of time and energy from equally enormously talented and dedicated people, because C++ is such an insanely complicated language. And this despite the best efforts of Stroustrup *et al* to keep C++ as simple as possible. The classic example is the issue of name-lookup. In C++, name lookup is simply ridiculously complicated – I would be very surprised if even a dozen people in the whole world understand it properly (I am certainly not one of them).

To help navigate the complexity that today's programming involves, we turn to books. There are some excellent ones such as Meyers' *Effective C++/STL* that stand out amongst the rubbish that is deposited on the industry, but I wonder whether Meyers and others are actually doing us a disservice. In effect, they make it possible for us to muddle along with current languages and development systems when we should really be throwing them out and doing things properly.

I've thought for a long time that *Effective C++* is basically an admission that C++ is a mess. That C++ is a mess is not a particularly novel idea of course, it's well known and there are good reasons for why it is as it is. Meanwhile, *Effective XYZ* is written to explain how to avoid the pitfalls in the language and new kludges are invented (I'm afraid I include Boost's various meta-programming libraries here). But it's not as though there are any significantly better alternatives around, and it's not just programming languages that are rubbish. More generally, there are hundreds of incompatible build systems around, we're stuck with an antiquated hierarchical file system design, and computers still go wrong far too often. Books spend hundreds of pages talking about the importance of using abstract interface classes and making only leaf-classes concrete, despite this involving so much boiler plate code that your header files double in size. Object orientation with methods owned by classes was proclaimed for years as the solution to all programming problems, despite manifestly not working in many situations; whole books are written explaining how to hack classes with methods so that they don't bite you too often or too badly. The existence of the horrific GNU `autoconf`/`automake` tools allows Unix vendors to get away with putting their header files in different places and offloading the resultant incompatibilities and needless complexities onto developers.

Every so often, I read something about Smalltalk or Lisp systems that had development systems that appear to be everything that we want today, have wanted for years but still don't have, but were around *decades* ago. Things like incremental compilation, modification of running code, reflection etc. were around and understood then. Meanwhile, this sort of thing simply cannot be done in C++ because the grammar is almost impossible to parse in batch fashion, never mind in an incremental way. Java was a massive opportunity, but appears to simply have made a different but roughly equal-sized set of mistakes. On the plus side, languages like Python seem rather well designed and potentially allow one to spend the majority of one's effort solving real problems, but I wonder whether they are really much better than the state of the art 20 years ago?

I think what I'd like to see is some admission from the industry (and that includes us) that the current state of affairs is a massive embarrassment to us all. I'd like it to be written in something very large and brightly-coloured that somehow things have gone badly wrong when writing a database system for a few million tax payers in the UK costs millions of pounds, and doesn't actually work, despite the fact that *all* of the fundamental technical problems of concurrency, transactions etc., were *solved* decades ago.

What should happen when a problem is solved (i.e. understood) in computing, is that the solution should be disseminated, and we should all move on to new and interesting things. Instead, we see people saying "wow" when a Java applet displays an animation. People had the same reaction twenty years ago when BBC micros or Spectrums displayed the same rotating cube. I worked for a company a few of years ago where we were trying to write a system where a web page written in HTML contained some Javascript, which talked to a Java program which talked to some C++ which made a COM call across a network to a server. It was a nightmare involving thousands of lines of hand-written code, but the actual problem being solved was mind-numbingly simple. I fear that thousands of people are engaged in similarly pointless activities, when they could be doing something new and/or useful.

a nightmare involving thousands of lines of hand-written code, but the actual problem being solved was mind-numbingly simple

Consider the rise of GNU/Linux and the BSDs. Many people find that a Linux server is much easier to maintain than a Windows one; things like Samba and Apache work fairly well and have allowed Linux/BSD to be adopted significantly. But have a close look at Linux (or any other Unix). I think it's incredibly crude. After modifying a configuration, you have to manually restart the relevant process. You have to manually set up some sort of backup system, none of which ever do exactly what you want. There is an increasingly complicated package system to avoid the DLL-hell problem. If you want to maintain previous versions of files, you have to set up and use CVS or subversion, which is hardly straightforward. KDE and Gnome look nice and work fairly well, but they take enormous amounts of resources and fundamentally don't do anything new compared to Windows or Apple's equivalents. Being sure that a Unix system is safe against intrusion is distinctly non-trivial.

I can hear people replying with "but you can do that with xyz", "choice is important", "if you don't know what you're doing, you shouldn't be doing it". My answer is that ordinary non-technical people want to do all of the things mentioned in the previous paragraph, but they cannot. I want to do them too, but they are a solved problem, so I should not need to spend time on them – I'd rather spend time trying to work on something new or different, not being the ten-millionth person to read the CVS man page and master its idiosyncrasies. Windows, to its credit, tries to simplify these tasks, but we all know that it doesn't work particularly well either. Francis

JULIAN SMITH

Julian's non-programming interests include violin playing and cycling in the Himalayas (http://op59.net/, jules@op59.net). He recently co-founded Undo Software (http://undo-software.com/) where he works on the undodb bi-directional debugger.

Premature Optimisation

Mark Easterbrook looks at some old code.

There are many coding techniques that can make life difficult for the legacy code maintenance programmer, and near the top of my "dislike" list is premature optimisation. This is where the original programmer is coding for efficiency instead of maintainability, but is concentrating at the code level and manages to achieve neither.

In the real-time and embedded world many programmers come from an assembler background and learn a higher level language on-the-job as they code their first large C project. Trying to apply assembler language optimisations to C usually results in both code bloat and lots of unsafe casting, resulting in a bug-ridden and fragile system.

One of my clients had such a system, originally written mostly by graduates and hardware engineers that, although they were very good engineers, had little experience writing large systems in C. The application was a multi-card embedded system that was mostly message driven both for handling the system payload and for OAM (Operations, Administration and Maintenance) operations. Optimisation started with the design of the message header structure:

```
typedef struct msg_tag {
  union {
    char *ptr;
    char data[4];
  } data;
  byte start;
  byte end;
  . . .
} msg;
```

The . . . represent other fields that are not important for this discussion, making the structure 8 bytes. If the message payload is up to 4 bytes, then it can be held in the header instead of allocating a message buffer. In an embedded system with limited memory and therefore a restricted number of message buffers, it must have seemed worthwhile to trade some simplicity for efficiency.

Encoding

To create and send a small payload:

```
msg mymsg;
mymsg.start=0
mymsg.end=0;
mymsg.data.data[mymsg.end++]=MSG_HEARTBEAT
mymsg.data.data[mymsg.end++]=proc_id;
send_msg(&mymsg,other_card);
```

and to create and send a larger payload:

```
msg mymsg;
mymsg.start=DEFAULT_MSG_OFFSET;
mymsg.end=mymsg.start;
mymsg.data.ptr=get_msg_buffer();
mymsg.data.ptr[mymsg.end++]=MSG_SETTIME;
mymsg.data.ptr[mymsg.end++]=time_now>>24;
mymsg.data.ptr[mymsg.end++]=time_now>>16&0xff;
mymsg.data.ptr[mymsg.end++]=time_now>>8&0xff;
mymsg.data.ptr[mymsg.end++]=time_now&0xff;
send_msg(&mymsg,other_card);
```

MARK EASTERBROOK

Mark is a software developer working with embedded systems, high performance/reliability/availability systems, operating systems and legacy code. He can be contacted at mark@easterbrook.co.uk

Complacency in the Computer Industry (continued)

Glassborow's recent book starts from the premise that anyone should be able to write a computer program. Similarly, anyone should be able to use computers safely and efficiently without having a nightmare getting access to old emails when transferring from one computer to a new one.

anyone should be able to use computers safely and efficiently

The rant is almost over. I shall finish by claiming that what we really need is to start over. Yes, really start over. Forget about the books that talk about how to avoid program crashes by never deriving from a class that doesn't have a virtual destructor, or having containers of pointers. Design a language where the compiler figures out what to do in order to be safe. If you give it a decent grammar, you can spend all the man years that would be spent on a C++ parser, on code analysis that generates faster code than the equivalent C++. Forget about teaching people about the importance of saving regularly under numbered filenames in case something crashes or you make a mistake; instead, design a file system where everything is automatically version controlled and backed up. Hey, you could even get rid of the Save button! Rather than writing yet another build system, try to think more generally and consider what a dependency tree is and how it could be integrated with the file system/backup system; consider what cache does, and think how it could be generalised. Don't write or use yet another widget system, try to abstract things a lot more – consider how widgets in a window are grouped, but so are files in a directory, or emails from a particular person, and try to figure out an abstraction that reflects these similarities.

I have a few ideas about these issues, some of which I'm developing at the moment. It's not easy, and there are many things that I don't know how to do properly, but I think that the important thing is to realise that things could be so much better, otherwise we are stuck in a world where eventually nothing new will ever be thought about, let alone done. ■

Listing 1

```
msg mymsg;
char *data;
uint32 *newtime;

if (get_msg(&mymsg)) {
  if (mymsg.end < 4)
    data = mymsg.data.data;
  else
    data = mymsg.data.ptr;
  switch (data[mymsg.start]) {
  ...
  case MSG_HEARTBEAT:
    newtime=(uint32*)(&data[mymsg.start+1]);
    set_time(*newtime);
    break;
  ...
  default:
    DEBUG("Invalid message: %d\n",
data[mymsg.start]);
   }
  if (mymsg.end > 3)
    ret_msg_buffer(mymsg.data.ptr);
}
```

Decoding

Once the message has arrived at its destination, it needs to be decoded, as shown in Listing 1.

As it was believed that function calls in C have a high overhead, all the "special cases" to deal with short and long payload types were in-line code and appeared in almost every source file. A lot was cut-and-paste, the rest was coded by hand to introduce mutations such as **`<=3`** instead of **`<4`**.

Analysis

With the development of a new system, some poor design decisions only start to look really bad after so much code has been written it is too expensive to go back and change them. At the point I got involved, the system had been in the field for several years and each new feature added had to test for this message header optimisation.

thousands of lines of inline special case code, quite a few bugs, and a maintenance nightmare, all so that just one message type doesn't need to have a message buffer allocated

After creating a regression test framework, it seemed a good investment for the future to remove the small message special case, or at least factor it out into a few common functions (or macros, if function call overhead was still considered a problem). The few original coders who were still around argued that removing it would slow the system down, so some analysis of the messages was in order:

- Of the many hundred possible messages [1], only three were 4 bytes or less.
- Of the three possible short messages, two were used for card-to-card and only one for process-to-process on the same card.

All card-to-card messages needed to be encapsulated, which increased the size, and therefore had to use a message buffer. The card-to-card communications module had to test each message and copy small messages into a message buffer.

So all that work, thousands of lines of inline special case code, quite a few bugs, and a maintenance nightmare, all so that just one message type doesn't need to have a message buffer allocated for it.

Listing 2

```
send_msg(msg *inmsg, byte card)
{
msg outmsg;
uint23 *inmsg_quick_copy=(int32*)&inmsg;
uint23 *outmsg_quick_copy=(int32*)&outmsg;

outmsg_quick_copy[0]=inmsg_quick_copy[0];
outmsg_quick_copy[1]=inmsg_quick_copy[1];
  . . .
```

Further optimisation

The message header structure was always allocated locally, usually on the stack, and therefore had to be passed to functions by value. Rather than let the compiler generate the most efficient copy, the programmers knew better so passed a pointer and performed the copy manually, as shown in Listing 2.

The target architecture was 68000 based, so non-word align accesses were safe, albeit slow. Although I never had time to measure it, I would have been very surprised if the **`int32*`** cast was faster than a **`memcpy()`** or structure copy, both of which would be easier to read and could be resilient to changes to the message structure.

Message handling revisited

The message code handling in this application has so many problems apart from the premature optimisation that it is worth picking apart a bit more:

1. The code assumes **`get_msg_buffer()`** does not fail. The whole driving force behind the previous discussion is a limited supply of message buffers so that failure is a real possibility. A failure returns **`NULL`**, which will immediately cause a bus error – on the surface this is a good thing because this is a dual redundant system and if something goes wrong it is better to failover to the hot-standby than try and recover from a crippled state. In practice, the system only ran out in two scenarios:
    - Memory leak. On a system with limited memory, this would be detected fairly early in development so is not a production problem.
    - High message load. Unfortunately a fail-over needed some buffering of messages, so it couldn't work reliably at maximum load. If the active card failed in this way, the hot-standby would shortly follow.

   This was a design error in not considering graceful degradation under high load and relying on the dual-redundancy to handle every failure condition.
2. The encode and decode are not symmetrical. This was endemic throughout the system with byte-by-byte, cast element-by-element, and cast whole structure being freely mixed throughout the system. Whenever a message structure was changed, every instance of encode and decode needed to be changed manually to match. Missing one would produce a difficult to detect bug.
3. The flag to indicate if the message was a small message or not was implicit in the end value: 0-3 small, 4-255 large, and the end value was the item that changed the most. As the flag meant the difference between interpreting the union as a pointer or data, the code was very fragile. Many messages were close to an end value of 255 so every increase in size need to be checked for overflow.
4. There was no sanity checking of messages, so an inbound message consisting of a byte stream was cast to a structure or sequence of built-in types in the hope that it was always correctly formatted.

5. Messages did not contain any versioning information despite the requirement for live upgrades. The only way to change a message format was to create a new message type and support both the old and new messages depending on the version of software in the partner card.

There is also some credit due:

1. The message was built starting at an offset (**`DEFAULT_MSG_OFFSET`**) so that if it needed to be encapsulated or needed a routing header added, it could grow from the beginning without having to shuffle-copy it within the buffer.
2. The **`begin`** and **`end`** formed a half-open-range with all the advantages it brings such as **`begin==end`** means empty, **`end-begin`** is size, etc.
3. Debug versions of the message buffer allocation/deallocation and the message send and receive functions used unused parts of the buffers to record statistical data and markers so that a snap-shot or crash memory dump of the message buffers provided valuable data about how the system was performing and gave early warning about buffer overruns.

Lessons learnt

There are a number of lessons that can be learnt from this code.

1. Don’t optimise (yet)

The original authors were not familiar with the first and second rules of optimisation (1st: Don’t optimise, 2nd: Don’t optimise yet). Their attempts at writing more efficient code were at odds with the tools they were using:

Modern compilers have very sophisticated optimisers. These are more likely to work given small simple sequences of code rather than “clever” tricks with multiple paths and duplicate code.

Most CPUs now have pipelines, out-of-order execution and both instruction and data caches. These work best with small code and minimal branches, neither of which result from programmer code-level optimisation.

2. Measure, don’t guess

Empirical evidence has shown many times that it is difficult to guess in advance the lines of code that account for the largest amount of execution time, and this case was no different. Making sure that optimisations are applied only after careful measurement has a number of advantages:

- Initial effort can be directed towards making the code readable, maintainable, and correct, instead of wasting time failing to make it fast.
- Deferring optimisation until later means that if there are no performance issues, no effort needs to be expended on it.
- Instrumentation and measurement is a lot less interesting than coding fast and clever code, and therefore is a deterrent to premature optimisation.

3. Choose one encode/decode scheme and keep to it.

Message decoding and encoding can easily become very messy and unmaintainable if not well designed:

- The encode and decode should be symmetrical, preferably using identical data definition methods.
- The closer they are in the code, the less likely they are to diverge in method and style.

Provide a method of checking a received byte stream message before “casting” it to local data structures. Often a simple length check is sufficient.

4. Consider message versioning

If one side of a messaging path can be upgraded without the other, then message versioning needs to be designed in from the start:

- Make data fields large enough for likely future use.
- Ensure that mismatch versioning can work, either by negotiating a common version, or by making later versions of messages backwards compatible.
- Consider including a version number in each message.
- Text based and TLV (Tag-Length-Value) message encodings are much easier to upgrade than plain binary structures.
- Don’t use byte sized messages codes unless you are absolutely sure you won’t need more than 256.

Conclusion

Premature optimisation is a problem that many programmers suffer from, yet can easily be avoided with a bit of fore-thought and help from some more experienced developers. ■

Notes

[1] Although there were only 255 possible message codes, **`0xff`** was reserved for expansion, allowing another 255 message codes to be allocated using the next byte.

Effective Version Control #2

Pete Goodliffe continues to describe good version control practices.

This is the second instalment of this series investigating version control. Last time we laid the groundwork, establishing what version control is and why it is an indispensable part of the development process. We saw the basic mechanics of VCS usage and learnt the important version control mantra: simplicity is a virtue.

In this part we'll delve more deeply into the version control quagmire. We'll look at how to manage version controlled repositories, and how to work correctly with source code that is held under source control.

Managing the repository

We start by looking at issues surrounding your version control server and the repositories held within it. We'll consider repository contents and file structure, and look at how we access repositories from development machines.

4. Create the right repositories

When do you create a repository? If you're starting from scratch then this isn't a difficult question. But even at this stage there's an important choice: what do you call the repository? Do you name it after:

- the new project's cryptic codename,
- the released product name (which might not yet be known),
- the code framework's name (if there is such a thing),
- the company name, or
- your favourite colour?

In different situations each of these answers could be right. You can only answer this question with a pragmatic eye on the future. Are future projects likely to build upon these code files (that is: is the next project is an evolution of this one), and so will it be held in the same repository? Will future projects be entirely separate work, or might they share some parts of the current code? Will future work on the same project need to go in this repository, or in a different one? Your future intentions give clues to the right choice of repository name, and obviously determine which repository your subsequent projects end up in.

When you eventually grow more than one repository, keep them on the same server if possible. It's the simplicity rule again: it will be more obvious where to find them.

See items 6 and 7 for more on repository creation policy.

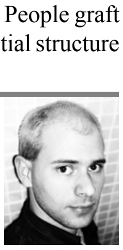
5. Structure your repository thoughtfully

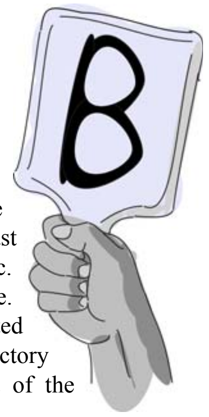
It's important to get your repository structure right: you'll be working with this beast for a long time. Of course, a good VCS allows you to easily move things about [1] but that's no excuse to not (try to) get it right first time. Aim for perfection – it can't hurt! You'll save reorganisation time later on.

A good structure sets the standard for how easy the repository is to work with. Projects inevitably grow over time, and there is a tendency for the repository to become large, unwieldy and hard to understand. People graft blobs of code into random places without thinking. A good initial structure will prevent the worst effects of code entropy.

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





Consider these structural issues:

- Partition well. Arrange the code into logical directories. Don't be afraid of creating structure to manage things – it doesn't make using the source tree any harder. Have at least one directory per component, per platform, etc. Do, however, resist the urge to over-structure. You know you've done this when nested directories start to hide the code. Good directory structure simplifies the check-out of a part of the project.
- Establish filename conventions for all files. Define a consistent capitalisation, and mandate that file names reflect their contents. Create conventions for where source code files go, where common include files go, and where test harnesses, shared libraries, applications and other assets go.
- Keep it all under constant review and management. Check that people haven't made a mess (they'll get up to all sorts of mischief when you're not looking). Tidy up and reorganise as necessary. A little constant care and attention will prevent a big headache later on.

All codebases, like gardens, need active maintenance, with caretakers to carefully tend and weed the borders. Plant the right repositories, and look after them well.

6. Manage third-party code

It's not unusual for your software to rely on code from other people that is shipped in source form, whether it's some open source code, or stuff written by subcontractors. This code needs treating differently from your normal code, or you could accidentally tie yourself up in knots. You must track the external upgrade releases, and integrate them with any code modifications that you've make locally.

There are countless bad ways to do this, and an established best practice: using vendor branches. We'll look at branches later on (in items 16-19, in the next issue of C Vu), but here's vendor branches in a nutshell:

- Create a branch for the third party source package. Name it after the package (with no version number)
- Import the first release into that branch.
- Tag the release (include the version number in the tag name).
- Use that branch in your build tree.
- If you need to make a modification to the vendor's code (perhaps to fix a bug you've found) then create a new development branch for the package, and commit your changes there. Now use this branch in your build tree.
- When the vendor sends you a new release, import it onto the vendor branch, and tag the new release (with the new version number in the tag name).
- Merge the vendor branch on to the corresponding development branch – this will merge their changes with yours. (If they've fixed the same bug you may have some conflicts to resolve).

Think about what third party code you put into vendor branches. If a package is justifiably part of the compilation environment and you won't ever modify it then it makes sense to leave it out of your build tree and make it an installation prerequisite.

7. Share code

Some parts of your code will be libraries or common components shared across a number of projects. Put these in well-defined places that clearly show they are shared (see item 5):

- Chose a directory name in the repository that highlights shared code (common is frequently used to denote 'common' code).
- Or put it in an entirely different repository that is specifically for shared code.

There are pros and cons to each approach. A different repository makes project check-out harder, whereas a 'common' directory full of shared components that aren't used in every project makes your repository larger. Chose whichever makes most sense for you.

Be careful with code that might one day be shared but isn't yet. Don't put it in a shared area until you know that it will be used on more than one project. (It's that simplicity rule again). Don't design code for multiple clients if there never will be one than one – you'll be wasting effort.

8. Store the right things

So you have a shiny new VCS set up and ready to go. You have a repository structure. What are you going to put in it? There are two answers:

Answer one is store everything. Store all project artefacts. Everything in your build tree that's required to create your software must be checked in. Starting with an appropriately configured build machine, with the correct OS and compilation environment (build tools, standard libraries, etc, plus sufficient disk space) [2], one simple check-out operation should get you a good buildable source tree. That means your repository must include:

- all source files,
- all build files (makefiles, IDE setup),
- all configuration files,
- all graphics/data files, and
- any third party files

required to create the final software.

one silly change or careless file operation could easily destroy all your hard work

Answer two is store as little as possible. You do have to store a lot of stuff. But don't put include any unnecessary cruft. It will confuse, bloat, and get in the way. Keep the file structure as simple as you can. Specifically:

- Don't store IDE configuration files or cache files (i.e. precompiled header files or dynamic code information, ctags files, user preference settings files, etc).
- Don't store generated artefacts – you needn't check-in object files, library files, or application binaries if they are a result of the build process. You needn't even check in automatically generated source files. Binary files from other companies (e.g. driver files, or third party dlls) might justifiably live in your repository, though.

Sometimes you'll see that generated files do get checked in: if they are particularly hard to generate or take a long time to create. This decision must be made very carefully – don't pollute your repository with unnecessary rubbish.

Should you version control the software release images? Some shops put all their releases into a repository. This is usually a separate 'release' repository; the binaries do not really belong beside the source files. However, consider archiving these in a simple static directory structure elsewhere. Version control does not buy you much when recording less dynamic file structures for posterity.

9. Treat the repository with respect

There is a clear separation between the code repository and your checked-out working copy. What you do to your working copy doesn't affect the repository in any way. Do what you want with it, treat it as a digital punch-bag. It's fine; you won't hurt anyone else. But be careful how you commit the changes you make (see item 13) and remember that you can't handle the repository as you can a working copy.

The repository is the centre of the universe for your source code; everything revolves around it. One silly change or careless file operation could easily destroy all your hard work. Never furtle around in a repository by hand without wearing the right protective clothing (including reinforced underpants). A repository's physical contents on disk are designed to be opaque and are not for human access. These things are notoriously brittle to tampering; one little edit could easily screw things up. Or, worse still, could break something very subtle that will only bite you years later. Hard. Some VCSs are even dependant in the repository's physical location – you can't move them around easily.

Mere mortals should have their access to the VCS server restricted to insulate them from their own evil urges. No one ever intends to break the repository. But every now and again, a quick rummage in its innards seems like a perfectly good idea: safer and easier than a more complex client command invocation. But the consequences could be dire…

Several times I've seen a VCS administrator resort to restoring a backed-up copy of a repository because a gung-ho developer tried to be clever and remove the evidence of their careless check-in by modifying the repository by hand. It's not safe and it doesn't work – especially when people could be running the engine (accessing version control) whilst you've popped the hood. Work gets lost. People get inconvenienced. You look stupid. Don't do it.

10. Administrate your repository well

Repositories obviously don't spring up out of the blue, and they don't work by magic. Someone makes them when the need arises (see item 4), and keeps them well oiled to ensure smooth operation. In item 9 we saw that developers shouldn't tinker with the inner workings of the VCS. Enforce this by separating the world into two classes, the developer proletariat (who clearly do all the important work, but are physically prevented from access to the VCS's inner sanctum) and the administrator bourgeoisie (who set the rules and police the state).

The administrator's jobs are:

- Perform privileged version control admin (i.e. create/remove/maintain repositories).
- Set up backups of every repository. Ensure these backups run smoothly. Test them from time to time. [3]
- Maintain access privileges so the right people can see/edit the right bits of code.
- Define secure access routes to the repository. Enable secure off-site VCS use too, if necessary. (Take care to lock down unnecessary routes into the repository – more open doors mean more security risks. If you want to keep your proprietary code under wraps then security issues are very important.)

This isn't just an exercise in bureaucracy. It's important to have a set of well-defined VCS administrators. That way people know who to run to when they have a problem. That way, when something does go pop, it's clear whose responsibility it is to sort out the mess.

Have stand-ins (deputy administrators) to avoid the danger of losing all your admins to a flu bug on a particularly bad hair day for your VCS.

11. Have a migration plan

You might have a lot of code that you're trying to migrate into a new VCS. Don't do it without thought: this code was important and how you get it into a new repository will determine how much use you'll get out of the amassed version control history.

You have two migration options: to retain the old version control history, or to take a baseline of the project and import that into the new VCS. There's no right answer: this depends on how easy migration is, and how much access to the legacy code you'll need in the future (don't forget that it will always be available in the old VCS if you ever need to perform some archaeology). If the new VCS doesn't provide repository conversion tools then a full history import requires a lot of manual legwork. This may be prohibitive.

There is a good middle-road strategy: to import the oldest version of code you will ever need access to (choose this point carefully). Make this the first commit to your new repository. Now import a copy of the very latest version of your code over this, and begin new work from here. If you ever need to perform bugfixing on an older version of the software (a version that isn't in the new VCS), then you can create a bugfix branch from the historical first import, import that release's source tree (out of the old VCS) into the branch, and then work on the bugfix down there.

This isn't supremely elegant, but it works well enough.

Working with code

These items describe how to work with version controlled code: both the way you should write it and the way you should handle it.

12. Manage your working copies

Your day-to-day VCS interaction takes place through the working copy – your checked out copy of the repository's files. To use your VCS well, you must use the working copy well.

A working copy is a personal thing. Only one person can use each working copy: don't share them. Don't export them over a network filesystem – this can confuse VCS operations. Even if it's mechanically OK to share a working copy, and two people working in the same directory won't terminally confuse your version control system, sharing can cause all sorts of confusion. If another person updates the tree without you realising, the world has changed and you don't know about it. The effects are far worse if they perform an update whilst you're doing a build. That's a sure-fire recipe for disaster.

Similarly, don't fiddle with someone else's working copy without letting them know. Perhaps Fred went home in a rush and left the repository in a broken state, so you need to check-in a file he missed, or unlock something he reserved. Without the discipline and manners to tell Fred what you've done, you will confuse him in the morning, and might even cause him to make another VCS mistake later.

Update your working copy frequently so that you are always working with the repository's state-of-the-art. Working days (or even hours) behind bleeding edge can lead to problems. If the APIs or code that you depend on change then your checked-in code might break the build. Of course, this depends on project maturity and the stability of the module APIs (you should always aim for stable APIs).

13. Make considered modifications

The act of developing code is a series of modifications to the source tree. This is the building block of VCS. However obvious it seems, make the right changes and consider how they affect the repository. Never ever check in code that will break the source tree – this will cause every developer using the repository untold grief; no one else can test their own work because they can't build any software. Breaking the tree is a criminal offence in many development shops, with consequences ranging from public humiliation to corporal punishment.

The simple process is: make a change, test it builds, test it works, check it in. In that order. Always. When you're dashing out the door to catch a bus it can be very tempting to rush a check-in of code that 'should work'. Take it from me: it rarely does.

There is one situation in which you can safely ignore this advice: when you're working on a personal branch (see item 16, in the next installment) and can be assured that no one else is relying on it to work.

Layout changes are frustrating in a version controlled source tree. A layout change hides other modifications – when you browse the version history important changes will be lost in a sea of whitespace alterations. For this reason, avoid making gratuitous changes unless absolutely necessary; and then never do it alongside 'real' code modifications.

keywords look cute, but they are evil and should be avoided

14. Have a good check-in strategy

The check-in commits your carefully made modifications to the repository. With the modified files you supply a textual description of what has changed – the log message. To ensure that the information in the repository is maximally useful, always provide good log messages. A good message includes: a brief summary and the detailed reasons for the change. It also describes the level of testing performed (e.g. does the code build, does it work properly, and does it pass the unit tests). If you're fixing a fault, include the bug number for cross reference. The description must be clear with enough background to make sense. Don't describe each individual change, or which files were changed – the VCS will record this automatically.

The best check-in strategy is little and often. Make small steps and commit them to the repository before making the next change. Don't accumulate a week's worth of work before you check in. You'll suffer many problems:

- It's harder to track the changes made in the code
- The rest of the code repository could have changed massively between your updates. You new work might not be valid any more.

Before you check in any code, first test it against the latest version of the repository – this will ensure that your new code won't break the build. Other components it depends on might have changed between your working copy updates, causing your new work to be erroneous.

15. Put the right things in your files

A lot of this is good development practice. Ensure that it's easy to find things in the repository, and make sure that file names match their contents. Never make a `KitchenSink.c`.

Many VCSs provide a facility to helpfully modify the source file content when it's committed, using keyword expansion. For example: include the text **`$Id$`** in a code comment somewhere, and on check-out this keyword will be updated with information including the author's name, and the date of last modification. Keywords look cute, but they are evil and should be avoided; they cause all sorts of problems, including odd file comparisons (often the keyword shows as a change between revisions, when it is not really an important change).

Even worse than **`$Id$`** is **`$Log$`** – a keyword that expands to the entire revision history of that file. There is never a good excuse for using this keyword, and typing it should be punishable by death. Don't do it. The log information gets in the way of the code, and is a duplication of information. Why should the file contain the exact information the repository records?

Some development shops like to maintain a text file containing the revision history of the repository. With every check-in you must edit this history. It is an odd practice – duplicating the VCS! Don't maintain by hand what the tool automatically does for you. If you can't get the information out of your VCS in a form you like, write a script to do it for you (see item 22, also in the next installment).

Next time...

Next issue will see the final instalment of this series. We'll look at good practices for branching and labelling your code, how to merge code safely, and how to use version control in your product release procedure. Until then, keep it simple... ■

Notes

[1] Most VCSs do, with CVS as the notable exception.
[2] This configuration must also be controlled – not necessarily version controlled, but documented clearly and unambiguously. Why not place that document under version control, though?
[3] It's *essential* to test the backups, otherwise when you need to restore a repository you could be sadly disappointed.

Threads and Shareable Libraries

Roger Orr tackles posix threads on Windows.

I came across a problem moving some code from Unix to Windows and thought some of the issues raised might be of general interest.

The Unix programs used a shared library to provide some optional functionality and this had been working fine on Linux/Solaris. The shared library performed some asynchronous functions, like logging, and used pthreads to provide the threading model. I wanted to extend the target architectures supported to include the Windows platform.

There were three possible approaches that I considered.

1. Use cygwin [1] to give a 'Linux API emulation layer' or 'Linux-like environment' on Windows. Alan Griffiths (and others) object to Cygwin describing itself as an 'emulation'; be that as it may, Cygwin does provide a very familiar environment for programmers coming to Windows from Unix.
2. Use pthreads-win32 [2] to provide the threading functions I needed on Windows (this library is LGPL-licensed).
3. Port the codebase to use Windows APIs.

The first approach was the easiest, as the make files and compiler settings were compatible with Linux, but it required target machines to have cygwin installed. I wasn't entirely sure I wanted this restriction (apparently you can get away with copying a single DLL but I've not been able to verify that this is officially supported).

The second approach seemed better: pthreads-win32 is a fairly mature library which seems to have a good record and it can be linked with the program or shipped as a single DLL to the target machines.

The last approach was not favourable as there was quite an overhead in producing (and maintaining) a separate code base for two different platforms. Obviously I could abstract away some of these differences – but this would effectively be trying to write the code for one of the first two approaches on my own!

Problem discovery

The initial work went very easily. I downloaded pthreads-win32 from Redhat; then copied `pthread.h`, `sched.h` and `semaphore.h` into my `include` directory and `pthreadVC2.lib` into my `library` directory. There was a little bit of work to set up a Visual Studio compatible build environment but the code (including the pthreads code) just compiled and ran without any problems – at least at first.

However, problems occurred with a program that dynamically loaded the shareable library. The program worked fine until it unloaded the library, at which point it just hung. **`Ctrl-c`** and **`Ctrl-break`** didn't stop it either; I had to use task manager to kill the process. One of the times I was killing the program this way I noticed that the column for 'Threads' (selectable using View->Select Columns->Thread count) was showing four threads, not the two I was expecting. Further investigation revealed that the thread count went up by one each time I pressed **`Ctrl-c`** or **`Ctrl-break`**. What was going on?

It looked like some sort of multi-threaded deadlock was occurring and my initial thought was that I had found a bug in pthreads-win32. So I decided to try out cygwin, hoping that this environment would not have the same problem. I was fairly quickly able to compile my shareable library and test program using gcc on cygwin. I discovered though that the behaviour of the program was exactly the same – even down to creating extra threads when I typed **`Ctrl-c`** – so it looked like something more complicated was going on than just a bug in pthreads-win32.

I tried to attach the Visual Studio 2003 debugger to the hung process, but wasn't able to get it to connect successfully. It simply didn't debug – and if I tried to break into the process I got a dialog box saying :

```
Unable to break into the process 'testwindows.exe'.
Please wait until the debuger has finished loading,
and try again.
```

I tried again, this time starting `testWindows.exe` under the debugger rather than attaching when the program surfaced. When it hung I tried to break into the program and got a message box saying:

```
The process appears to be deadlocked (or is not
running any user-mode code). All threads have been
stopped.
```

I was however able to view the list of threads – one thread was inside **`thread_join()`** and the other was inside **`__endthreadex`**.

I abandoned the Visual Studio debugger as it wasn't helping me resolve the issue and reached for a different tool – WinDbg. This is one of the Microsoft Debugging Tools [3] and gives a lower level, but sometimes more informative, view of your process than the Visual Studio toolset. This time I hit pay dirt. When the test program hung I attached WinDbg and got the following couple of lines embedded in the output from the debugger:

```
Break-in sent, waiting 30 seconds...
WARNING: Break-in timed out, suspending.
```

This is usually caused by another thread holding the loader lock.

I executed the WinDbg command **`!locks`** and it displayed the following:

```
CritSec ntdll!LdrpLoaderLock+0 at 77FC2340
LockCount          2
RecursionCount     1
OwningThread       f38
EntryCount         2
ContentionCount    2
*** Locked

Scanned 102 critical sections
```

At this point the debugging session had provided enough information for me to understand what was the cause of the problem.

The Windows API uses a per-process lock internally (as the output from WinDbg shows this lock is defined inside `ntdll.dll`, the lowest level user-space library in the system). This critical section is locked while loading and unloading shareable libraries and this same lock is also used during thread exit. (There is more information about this lock on MSDN [4].)

My program was calling **`FreeLibrary`** and the implementation of this call locks the loader lock and then calls the DLL entry point to notify the DLL that it is being unloaded. The C++ runtime hooks into this call in order to run destructors for any static objects including the destructor for the 'theWorker'. This destructor signalled the asynchronous thread to exit (by setting **`timeToEnd`** to **`true`** and waking the thread up) and then joined against the thread. However the thread exit also needs the internal loader lock (since thread detach is notified to shared libraries using the same mechanism as when unloading libraries).

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

A classic deadlock has occurred: the main program thread has the loader lock and is waiting for the worker thread to complete; the worker thread needs the loader lock to complete its exit.

Whose fault is this hang?

Is this then a fault in both the pthreads emulations on Windows? This is hard to answer – there is no standard binding for Posix on C++, just on C. The 2003 C++ standard does not mention threads and so has nothing to say about defined behaviour in my example, and I looked in vain at the POSIX standards [5] for mention of destructors or statements of validity of **pthread_join** inside calls to **dlclose**.

It seems that the designers of Windows wished to ensure that the thread 'main' function was only active in one thread at a time. This does have some advantages – for example it means easy avoidance of common race conditions initialising and terminating shared libraries. The downside is the possibility of deadlock.

Since the loader lock is an internal implementation decision of the Win32 API it is hard to see how the pthread-win32 library or cygwin could avoid the deadlock since the use of the loader lock is outside direct application control.

It also means that option (3) – re-implementing the library using Win32 calls – would not help either since the problem was with the lock used by the Win32 API itself.

Simplify, simplify

I wanted to have a smaller piece of code to work on while I investigated the problem further. I managed to refine the code down to a small example which demonstrated the same hang on unload. The sample program worked perfectly on Unix using **dlopen**, **dlsym** and **dlclose** but not on Windows using Cygwin, nor when using the equivalent Win32 functions **LoadLibrary**, **GetProcAddress** and **FreeLibrary**.

The sharable library has only one entry point: **callLibrary()**. This makes sure the **Worker** instance is initialised and then calls a method on this instance. The worker makes use of a helper thread to perform its function; in this example the worker thread simply waits for **timeToEnd** to be set – the original code used some internal queueing to pass work items to the helper thread.

The driving program is also very simple. It loads the library, looks up the address of the entry point **(callLibrary)** and invokes it. Then it unloads the library and exits.

The code is shown in Figure 1 (Loadable.cpp) and Figure 2 (testWindows.cpp).

Listing 1

```
// Loadable.cpp
#include <iostream>
#include <memory>
#include <pthread.h>

class Worker
{
public:
   Worker();
   ~Worker();
   void dosomething();
private:
   void start();
   void stop();
   void run();
   static void *threadStart( void * );
   pthread_t thread;
   pthread_mutex_t mutex;
   pthread_cond_t cond;
   bool timeToEnd;
   bool started;
};
std::auto_ptr<Worker> theWorker;
```

Listing 1 (cont'd)

```
Worker::Worker()
: timeToEnd( false )
, started( false )
{
   std::cout << "Worker ctor" << std::endl;
   pthread_mutex_init( &mutex, 0 );
   pthread_cond_init( &cond, 0 );
}
Worker::~Worker()
{
   std::cout << "Worker dtor" << std::endl;
   stop();
   pthread_cond_destroy( &cond );
   pthread_mutex_destroy( &mutex );
}
void Worker::dosomething()
{
   start();
   std::cout << "Queue something" << std::endl;
}
void Worker::start()
{
   pthread_mutex_lock( &mutex );
   if ( ! started )
   {
      pthread_create( &thread, 0, threadStart,
       this );
      started = true;
   }
   pthread_mutex_unlock( &mutex );
}
void Worker::stop()
{
   if ( ! started )
      return;
   pthread_mutex_lock( &mutex );
   timeToEnd = true;
   pthread_mutex_unlock( &mutex );
   pthread_cond_signal( &cond );
   pthread_join( thread, 0 );
   started = false;
}
void Worker::run()
{
   pthread_mutex_lock( &mutex );
   while ( ! timeToEnd )
   {
      pthread_cond_wait( &cond, &mutex );
   }
   pthread_mutex_unlock( &mutex );
}
void *Worker::threadStart( void * arg )
{
   Worker *self = static_cast<Worker *>(arg);
   self->run();
   std::cout << "Thread exiting" << std::endl;
   return 0;
}
extern "C"
{
   void
#if defined (_MSC_VER)
   __declspec( dllexport )
#endif
   callLibrary()
   {
      if ( ! theWorker.get() )
      {
         theWorker.reset( new Worker );
      }
      theWorker->dosomething();
   }
}
```

Listing 2

```
// testWindows.cpp
#include <windows.h>
#include <iostream>
#include <string>
int main( int argc, char ** argv )
{
   std::cout << "About to load library"
    << std::endl;
   std::string library( argc == 1 ? "Loadable"
    : argv[1] );
   HMODULE handle = LoadLibrary( library.c_str() );
   if ( ! handle )
   {
      std::cout << "Failed to load '" << library
       << "': " << GetLastError() << std::endl;
      return 1;
   }
   std::cout << "Loaded library '" << library
    << "'" << std::endl;
   PROC pCall = GetProcAddress(
    handle, "callLibrary" );
   if ( pCall )
   {
      typedef void (*pfn)();
      pfn pfnCall = (pfn)pCall;
      std::cout << "About to call library"
       << std::endl;
      pfnCall();
   }
   // Fake some work before we close the library...
   Sleep( 5 * 1000 );
   PROC pClose = GetProcAddress(
    handle, "closeLibrary" );
   if ( pClose )
   {
      typedef void (*pfn)();
      pfn pfnClose = (pfn)pClose;
      std::cout << "About to close library"
       << std::endl;
      pfnClose();
   }
   std::cout << "About to unload library"
    << std::endl;
   if ( FreeLibrary( handle ) == 0 )
   {
      std::cout << "Failed to unload: "
       << GetLastError() << std::endl;
      return 1;
   }
   std::cout << "Unloaded library" << std::endl;
  // Fake some more work before we eventually exit...
   Sleep( 5 * 1000 );
   std::cout << "About to exit" << std::endl;
   return 0;
}
```

When I ran this program on Windows here is the output I saw:

```
About to load library
Loaded library 'Loadable'
About to call library
Worker ctor
Queue something
About to unload library
Worker dtor
Thread exiting
```

and the program was hung.

I still had my hang – how could I prevent it? The fundamental problem was trying to stop a thread while unloading a shared library.

At first I thought about taking out the **`pthread_join()`** – mark the thread as stopping and then return from the destructor. Unfortunately this produces a nasty race condition – we are calling the destructor while handling the unload of the DLL, after we return from **`FreeLibrary`** the DLL's code and data has gone from memory and so if the second thread is still running it will get an access violation as there is no longer any code for it to execute!

Eventually I decided that the best solution was to add a **`closeLibrary()`** call to the shared library that would stop the thread and to ensure that client programs using dynamic loading called this function before unloading the DLL. This still left a problem if the client code neglected to call this function of course, but at least a diagnostic message could be produced warning that the library had been unloaded without closing.

A slightly safer approach is for the library to call **`LoadLibrary`** on itself when it creates the thread and **`FreeLibrary`** when **`closeLibrary()`** completes. This ensures that if **`main`** tries to unload the library without calling **`closeLibrary()`** the library will stay in memory and avoid the free-running thread causing a crash.

Microsoft published an MSJ article some years ago that solved the problem by splitting the single DLL into two; the second DLL contained the actual worker threads each of which called **`LoadLibrary`** on the second DLL itself to keep it in memory while they executed. These worker threads called **`FreeLibraryAndExitThread`** when they completed so the second DLL was only unloaded asynchronously by Windows when all the threads completed. This would have been rather more of a change to the sharable library than I had time for (and also the use of **`FreeLibraryAndExitThread`** would make the resultant code rather more Windows-specific than I wanted).

What have I learned?

I think there were three main things I learned from this experience.

Firstly, the interaction of threads, sharable libraries and C++ destructors is a bit of a minefield. Although my program worked happily on Unix this might just be luck – I don't think there are any guarantees in the standards that the code should work. Actually, this article could be viewed as another example of the singleton anti-pattern since the single instance of the worker thread object is the root cause of the problem.

Secondly, when debugging having multiple tools in your toolkit is a good thing. In this example the Visual Studio debugger was unable to provide much assistance with this particular problem; but WinDbg produced helpful information as soon as I used it. In fact, I find that WindDbg produces additional information so often that sometimes I attach it non-invasively to a process already under the Visual Studio debugger. This allows both debuggers to be used to investigate the state of the stopped program.

And finally, this exercise shows the truth of the phrase "abstractions are leaky" [5]. I was trying to use pthreads-win32 and cygwin to provide Unix-like threading a non-Unix operating system. In this particular case an implementation detail of the underlying operating system leaked into the application and caused a failure. It is hard to prepare for this in advance, since it is difficult to predict where the abstraction is going to break down, and it also indicates that the better your knowledge of the underlying platform-specific implementation the more likely you are to be able to recognise and solve such problems. ■

References

[1] www.cygwin.com
[2] sourceware.org/pthreads-win32
[3] freely available from http://www.microsoft.com/whdc/
[4] http://msdn2.microsoft.com/en-us/library/ms172219.aspx (example)
[5] http://www.opengroup.org/onlinepubs/009695399/toc.htm
[6] www.joelonsoftware.com/articles/LeakyAbstractions.html

Acknowledgements

Thanks are due to Alan Griffiths, Phil Bass and others who reviewed a draft of this article and suggested improvements.

Header Checker

Tim Penhey demonstrates the ease of Python.

This series of articles is targeted at those readers who don't know too much about Python [1], but are curious. Curious about what Python can do, whether or not it is as easy to read as the Python enthusiasts expound, and is it worth putting effort into learning.

I have been wanting to write an article about Python [1] for a while but as is often the case you think "what could I possibly write about?" I wasn't really interested in writing a syntax guide, there are many of those around already [2]. The solution to the quandary was to ask a muse (Paul Grenyer).

[13:37] tim: I'm wanting to write a python article for cvu

[13:37] tim: in the comming edition

[13:37] Paul Grenyer: ok

[13:37] tim: but not sure what to cover

[13:37] tim: ideas?

<boring – non relevant bit snipped>

[13:38] Paul Grenyer: Well, the last script I was thinking of writing....

[13:39] Paul Grenyer: was to go through source files and checking to see if the GNU license was at the top and if it wasn't, adding it.

I thought that this could be interesting, mildly useful, and cover a number of things without being too complex. So here we go.

The first thing to do is to decide exactly what the script has to do.

- Given a base directory to crawl
- Iterate over the files, and check files with extensions that match a predefined set
- Check to see if the provided header file is at the start of each file

Next, we need some files to crawl over. Given that it was Paul I was talking to, he suggested Aeryn [3], an excellent C++ unit testing framework he wrote.

The interactive interpreter

Python is an interpreted language. This means that there is no explicit compile step to turn source code into something that is executed. Python also provides an interactive interpreter (often just referred to as 'the interpreter'). Once the interpreter is started, it presents the user with a prompt at which you can type code directly. Python statements are executed as they are entered; this is a great place to test out code without even committing ideas to a source file.

On Linux machines, Python is normally in the default path, and starting the interpreter is done by calling the Python executable from the shell prompt, as shown in Figure 1.

If you are in Windows with a default python installation, then the location of the Python executable isn't in your PATH. See Figure 2.

To exit the Python interpreter, use `Ctrl-D` on Linux, or `Ctrl-Z` on Windows. Even though the development of this script was done on both Linux and Windows, all the path names in the examples are given as Linux paths for consistency.

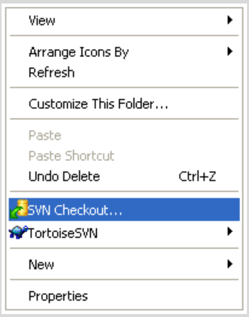
Getting Aeryn

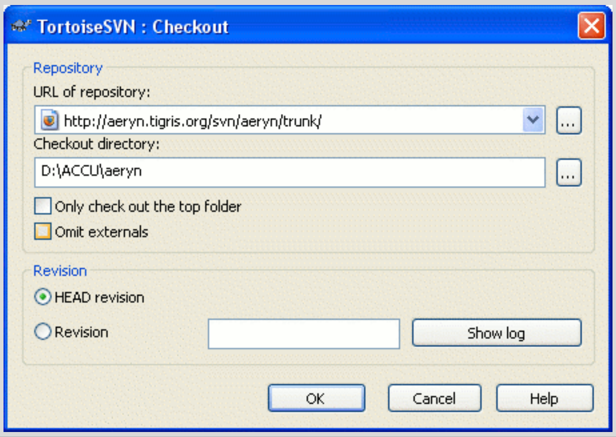
On Linux:

```
$ cd ~/sandbox
$ svn co http://aeryn.tigris.org/svn/aeryn/trunk
    aeryn
```

On Windows

1. Install TortiseSVN [4]
2. In Explorer, right click "SVN Checkout..."
3. Specify the URL of the repository and the Checkout directory




TortoiseSVN : Checkout

Repository
URL of repository: http://aeryn.tigris.org/svn/aeryn/trunk/
Checkout directory: D:\ACCU\aeryn
☐ Only check out the top folder
☐ Omit externals

Revision
● HEAD revision
○ Revision

```
tim@spike:~$ python
Python 2.4.3 (#2, Apr 27 2006, 14:43:58)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
tim@spike:~$
```

Figure 1

```
C:\Documents and Settings\Tim>c:\Python24\python
Python 2.4.3 (#69, Mar 29 2006, 17:35:34) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z

C:\Documents and Settings\Tim>
```

Figure 2

TIM PENHEY

Tim believes in choosing the right tool for the job. After years of hard core C++ hacking he's found that some things are just easier in Python. He can be reached at tim@penhey.net

Which files?

The first piece of code we need is something that will crawl a directory recursively and easily.

The first place to look for code that you suspect to exist is the Python Library Reference [5]. Anything that is operating system specific, such as traversing a directory is in the **os** module. A quick look through the documentation leads us to the Files and Directories section [6] and there is an excellent example of how to use the **os.walk** function there.

Listing 1

```
>>> import os ①
>>> for root, dirs, files in \
... os.walk('/home/tim/accu/aeryn'): ②
...   print root, dirs, files ③
...
[result snipped - way too much stuff printed out]
```

Notes on Listing 1

① Python comes with an extensive list of standard modules – **os** contains operating system specific commands. The **import** statement finds the module (by traversing the python path), initialises it, and defines one or more names at local scope. In this case the local variable **os** is initialised to be a module instance.

② The **walk** function provides iteration over the directories. The result of each iteration is three parameters – the directory name, a list of directories in that directory, and a list of files in that directory

③ The **print** statement takes an arbitrary number of parameters, and by default will print the string representation of the parameters. Also by default the print statement adds a carriage return.

A quick look at the results from this shows that the subversion directories are also being crawled here – and we don't want that. So let's add a quick check to remove them. (Listing 2)

Listing 2

```
>>> for root, dirs, files in \
... os.walk('/home/tim/accu/aeryn'):
...   if '.svn' not in root:   ①
...     print root, dirs, files
...
/home/tim/accu/aeryn ['.svn', 'corelib', 'examples',
'include', 'make', 'src', 'testrunner',
'testrunner2', 'tests', 'www'] ['aeryn2.sln',
'Doxyfile', 'lgpl_aeryn.txt', 'license.txt',
'Makefile', 'SConstruct', 'VERSION']
/home/tim/accu/aeryn/corelib ['.svn']
['corelib.vcproj', 'Makefile']
    <<-- snipped results -->>
/home/tim/accu/aeryn/examples/customreport1 ['.svn']
['main.cpp', 'customreport1.vcproj']
/home/tim/accu/aeryn/examples/mockfiletests ['.svn']
['main.cpp', 'mockfiletests.vcproj']
```

Note on Listing 2

① **'str' in var** returns **True** if the string **'str'** is found in the string **var**, **a not in b** is the same as **not a in b**, but easier to read

This is a bit messy, though. A re-read of the documentation shows that the list of directories that is returned from the **walk** function is checked for the next iteration. Removing an entry from the directory list tells the function not to traverse it. This is a much tidier way of avoiding the subversion directories. (Listing 3)

Listing 3

```
>>> for root, dirs, files in os.walk('/home/tim/accu/aeryn'):
...     if '.svn' in dirs: dirs.remove('.svn')
...     print root, dirs, files
...
/home/tim/accu/aeryn ['corelib', 'include', 'www', 'src', 'testrunner2', 'tests',
'testrunner', 'make', 'examples'] ['aeryn2.sln', 'Doxyfile', 'VERSION',
'lgpl_aeryn.txt', 'license.txt', 'SConstruct', 'Makefile']
    <<-- snipped results -->>
/home/tim/accu/aeryn/examples/customreport1 [] ['main.cpp', 'customreport1.vcproj']
/home/tim/accu/aeryn/examples/mockfiletests [] ['main.cpp', 'mockfiletests.vcproj']
>>>
```

Once the complexity of the code being written goes over a couple of lines, I end up writing a script to contain the code. A script can be imported into the interpreter as a module. For example, Listing 4 is the file `checker.py`.

Listing 4

```
import os  ①

def files(basedir, extensions): ②
  result = []  ③
  for root, dirs, files in os.walk(basedir):
    if '.svn' in dirs:
      dirs.remove('.svn')
    for file in files:
      for ext in extensions:
        if file.endswith(ext):
          result.append(os.path.join(root, file)) ④
  return result  ⑤
```

Notes on Listing 4

① every script has to stand alone, so a script must **import** all modules it uses

② the **def** command is used to create a function

③ no need to declare variables, just assign to them. In this case result is set to be an empty list

④ **os.path.join** joins together two or more path elements with the appropriate path separator for the platform

⑤ return the list of files – if the **return** statement is omitted, then **None** is returned

And the script is imported into the interpreter as shown in Listing 5.

Listing 5

```
>>> import checker
>>> result = checker.files('/home/tim/accu/aeryn',
... ('.cpp','.hpp'))
>>> print result[:5] ①
['/home/tim/accu/aeryn/include/aeryn/
platform_report_output.hpp', '/home/tim/accu/aeryn/
include/aeryn/test_name_not_found.hpp', '/home/tim/
accu/aeryn/include/aeryn/use_name.hpp', '/home/tim/
accu/aeryn/include/aeryn/namespace.hpp', '/home/tim/
accu/aeryn/include/aeryn/xcode_report.hpp']
>>>
```

Note on Listing 5

① **[a:b]** is the slice operator, where **a** and **b** are optional. **result[:5]** says return a list that has the elements from **result** starting at the start and up to but not including element at index **5** (the sixth element – indices start from zero).

Working with command line options

Now that there is a way to get the names of all the files that we are interested in, we need to do something with them. In this case, look at the start of the files to see if there is a matching header. A simple way to define the header that we are looking for is to put the header in a file, and pass that name of the file to the script.

Command line parsing is a problem that all but the most trivial scripts need to handle. Luckily Python has an outstanding module for parsing command line arguments – **optparse** [7] .

Listing 6

```
from optparse import OptionParser  ①

def parse(args):  ②
  parser = OptionParser()
  parser.add_option('-d', '--dir', default='.',
        help='The base directory to start from')③
  parser.add_option('-e', '--ensure-header',
        dest='header', metavar='FILE',  ④
        help='Ensure that the header in FILE is'\
        ' at the start of all the files')
  parser.add_option('-r', '--remove-header',
        dest='remove', metavar='FILE',
        help='Remove the specified header first'\
        ' if it is there')
  parser.add_option('-x', '--ext', action='append',⑤
        help='Look at files with this extension')
  parser.add_option('-t', '--test',
        action='store_true', ⑥
        default=False,
        help="Test run, doesn't actually change
        the files") ⑦
  return parser.parse_args(args)  ⑧
```

Notes on Listing 6

① Instead of importing an entire module and prefixing the use of all functions, you can also **import** individual functions, classes or variables from modules using the **from** statement. When imported this way, the imported entity does not have to be prefixed with the module name.

② Command line arguments are available as a list. An advantage of having a stand alone function for parsing the arguments is that it can be tested in isolation using the interactive interpreter.

③ The first parameter is the short option name, and the second is the long option name. If a **default** is not specified **None** is used. The **help** parameter is printed out if the **-h** or **--help** option is set. The variable name that is used to store the option is, by default, the same as the long option name with the prefix **'--'** removed, so in this case **'dir'**.

④ Here the variable name to store the option is being overridden to a shorter name using the **'dest'** parameter. The **metavar** parameter is used only when printing the help.

Start of help text without **metavar**:

```
  -e HEADER, --ensure-header=HEADER
```

Start of help text with **metavar**:

```
  -e FILE, --ensure-header=FILE
```

⑤ The **append** action allows the option to be specified multiple times. The variable containing the option is returned as a list.

⑥ The **store_true** action specifies that there is no associated input expected for this option.

⑦ Notice that the **help** string here uses double quotes not single quotes. Python strings can be defined using either single or double quotes, although normal usage is to use single. Here I am using double quotes as it avoids having to escape the single quote in **doesn't**. Alternatively it could have been written as:

```
help='Test run, doesn\'t actually change the files'
```

⑧ The **parse_args** function returns a tuple of **(options, args)** where the **options** object contains the parameter values, and the **args** parameter is a list containing the arguments that did not match any of the options.

The **options** object that is returned has members defined according to the arguments that were parsed, shown in Listing 7.

Notes on Listing 7

① If a module has been imported and then changed, the updated code can be loaded by using the **reload** command.

Listing 7

```
>>> reload(checker)  ①
<module 'checker' from 'checker.py'>
>>> args = ['-d', '/home/tim/accu/aeryn',
... '--ensure-header=gnu.txt',
... '-x.cpp', '-x', '.hpp', '-t']  ②
>>> options, args = checker.parse(args)
>>> options.dir
'/home/tim/accu/aeryn'
>>> options.header
'gnu.txt'
>>> options.ext
['.cpp', '.hpp']
>>> options.test
True
>>>
```

② This gives a way of testing the equivalent of passing **'-d /home/tim/accu/aeryn –ensure-header=gnu.txt -x.cpp -x .hpp -t'** on the command line.

Looking for headers

In order to create the file `gnu.txt`, I cut the top off one of the files. To load the contents of the file into a variable you can do this:

```
>>> header = open(options.header).read()
```

This is a little sloppy though as it relies on the garbage collector to close the file handle for the associated file object. Python 2.5 (which is currently in beta) is adding a **with** statement which is similar to the **using** statement in C#. Until then, the clean way is like this (Listing 8):

Listing 8

```
>>> f = open(options.header)  ①
>>> header = f.read()   ②
>>> f.close()  ③
```

Notes on Listing 8

① The **open** command returns a file object, and by default opens a file read only.

② The **read** method returns the entire contents of the file as a string.

③ Close the file.

Since this functionality is going to be needed in a few places, put it in a function (Listing 9):

Listing 9

```
def readfile(filename):
  '''readfile(filename):
  returns the contents of the file 'filename' '''  ①
  f = open(filename)
  contents = f.read()
  f.close()
  return contents
```

Note on Listing 9

① This is called a documentation string or **docstring** and is used to document functions, classes or modules.

If the first statement is a string literal, it is bound to the attribute **__doc__** (and **func_doc**).

Strings that span multiple lines can be specified by using three single or double quotes – called triple quoted strings.

```
>>> reload(checker)
<module 'checker' from 'checker.py'>
>>> print checker.readfile.__doc__
readfile(filename):
    returns the contents of the file 'filename'
```

Next we need to write the function that will actually check the headers of the source files. Python has many convenient string handling functions, a few of which will be used here.

Listing 10

```
def process(filename, ensure_header, remove_header, options):
    '''process(filename, ensure_header, remove_header, options):
       filename: a string
       ensure_header: the header to be added if missing
       remove_header: the header to be removed if found
       options: options object from command line parsing
    '''
    contents = original = readfile(filename)
    actions = []
    if remove_header and contents.startswith(remove_header):  ①
        actions.append('removed header')
        contents = contents[len(remove_header):]   ②
    if ensure_header and not contents.startswith(ensure_header):
        actions.append('added header')
        contents = ensure_header + contents
    if contents != original:
        print filename, ' and '.join(actions) ③
        if not options.test:
            f = open(filename, 'w')  ④
            f.write(contents)
            f.close()
```

Notes on Listing 10

① The following entities evaluate to **False**: **None**, empty string, or empty container (**list**, **tuple**, **set**, **dict**). Python uses short circuit boolean evaluation, so in this case if **remove_header** is **None** the interpreter never tries to evaluate the **startswith** method.

② The **len** function is the standard way to get the length of different types. Here we are returning the substring of the contents from the end of the header to the end of the string.

③ String literals in source are treated as string objects. The **join** method takes something that can be iterated over as a parameter, and creates a string by appending the contents of itself between each item in the parameter.

④ Open the file in **write** mode.

Created a simple file with a few lines of code called `test.txt`. Back to the interpreter to test the process function.

```
>>> reload(checker)
<module 'checker' from 'checker.py'>
>>> checker.process('test.txt', header, None, options)
adding header to test.txt
```

Checked the file – no change. Hmm... hang on a sec, was **test** set to **True** or **False**?

```
>>> options.test
True
>>> options.test = False
>>> checker.process('test.txt', header, None, options)
adding header to test.txt
>>> checker.process('test.txt', header, None, options)
>>> checker.process('test.txt', None, header, options)
removing header from test.txt
>>> checker.process('test.txt', None, header, options)
>>>
```

Bringing it together

Now it is behaving as expected. The last few bits that are needed to tie the functions together in a script are shown in Listing 11.

Notes on Listing 11

① While not absolutely necessary, I like to have **main** as a specified function so it can be called in the interactive interpreter.

② The logical operators **and** and **or** do not return boolean values, but return the last expression needed to calculate the return value.

```
>>> 'hello' or 42
'hello'
>>> 'hello' and 42
42
```

③ The module level variable **__name__** is set when the script is executed or imported. When the script is imported as a module **__name__** is set to the name of the script (in this case **'checker'**). When executed from the command line **__name__** is set to **'__main__'**.

④ Import statements can appear anywhere, and since the only place the **sys** module is needed is when the script is run as a script, we can load it there. The command line arguments are found in the list member **argv**.

Listing 11

```
def main(args):  ①
  (options, args) = parse(args)
  if not options.header and not options.remove:
    print 'Nothing to do, neither --ensure-header'\
          ' nor --remove-header set'
    return
  if len(options.ext) < 1:
    print 'Nothing to do, no extensions specified'
    return
  ensure_header = options.header and readfile(options.header) ②
  remove_header = options.remove and readfile(options.remove)
  filenames = files(options.dir, options.ext)
  for filename in filenames:
    process(filename, ensure_header, remove_header, options)

if __name__ == '__main__': ③
  import sys  ④
  main(sys.argv)
```

And there you have it. The next thing to do is to see if it actually works. In order to get some form of meaningful results over the Aeryn codebase, I decided to update the copyright date in the GNU licence header. I copied the header from a file and named it `gnu2005.txt`, then edited the file to have year 2006 and saved it as `gnu2006.txt`. Executing the script in test mode with the following parameters gave some surprising results:

```
./checker.py -d ~/accu/aeryn/src -x .cpp -x .hpp -r
gnu2005.txt -e gnu2006.txt -t
```

Code rarely works the first time

There were several files where it wasn't removing the header, but it was adding one. Looking at these files it became apparent that that the matching algorithm was less than entirely sufficient. The headers matched in all places except one space. Given that this then meant that the header was not matched, and it would have ended up with two headers is not ideal. There must be a better way. One solution is to match against strings that have all the whitespace stripped. However once you have found that it matches, you then need to somehow work out the substring of the file that contains the header so it can be removed.

Stripping whitespace from a string is relatively simple.

```
''.join(content.split())  ①
```

① The **split** method creates a list of words from a string broken on whitespace. In this case the string that is the delimiter between each of the words is the empty string, so we end up with a string stripped of all whitespace.

Finding the appropriate position of the non-stripped string for cutting is not much more difficult. In order to deal with stripped strings, there needed to be some modification to the functions process, and **main** as in Listing 12.

Notes on Listing 12

① The **xrange** function generates the values from zero up to but not including the parameter value, so **xrange(5)** will generate the values **0**, **1**, **2**, **3**, **4**.

② Python does not have either postfix or prefix increment (or decrement) operators. It does however have **increment** and **assign**. In fact it pretty much has any operator with assignment supported.

Executing the updated script over the Aeryn source no longer gave any surprises. The full Python script and gnu headers used for these tests can be found on my website [8].

I sincerely hope that this article has enlightened you to some of the power and simplicity of Python. ■

Listing 12

```
def strip_header(contents, header):
  i = 0
  for x in xrange(len(header)):  ①
    while contents[i] != header[x]:
      i += 1  ②
    i += 1
  return contents[i:]

def process(filename, ensure_header, ensure_stripped,
            remove_stripped, options):
  '''process(filename, ensure_header, remove_header, options):
     filename: a string
     ensure_header: the header to be added if missing
     ensure_stripped: the ensure_header with no whitespace
     remove_stripped: the header to be removed if found
        with the whitespace removed
     options: options object from command line parsing
  '''
  contents = original = readfile(filename)
  contents_stripped = ''.join(contents.split())
  actions = []
  if remove_stripped and \
     contents_stripped.startswith(remove_stripped):
      actions.append('removed header')
      contents = strip_header(contents, remove_stripped)
      contents_stripped = contents_stripped[len(remove_stripped):]
  if ensure_stripped and not contents.startswith(ensure_header):
    if contents_stripped.startswith(ensure_stripped):
      contents = strip_header(contents, ensure_stripped)
      actions.append('updated header')
    else:
      actions.append('added header')
      contents = ensure_header + contents
  if contents != original:
    print filename, ' and '.join(actions)
    if not options.test:
      f = open(filename, 'w')
      f.write(contents)
      f.close()

def main(args):
  (options, args) = parse(args)
  if not options.header and not options.remove:
    print 'Nothing to do, neither --ensure-header'\
          ' nor --remove-header set'
    return
  if len(options.ext) < 1:
    print 'Nothing to do, no extensions specified'
    return
  ensure_header = options.header and readfile(options.header)
  ensure_stripped = ensure_header and ''.join(ensure_header.split())
  remove_header = options.remove and readfile(options.remove)
  remove_stripped = remove_header and ''.join(remove_header.split())
  filenames = files(options.dir, options.ext)
  for filename in filenames:
    process(filename, ensure_header, ensure_stripped,
            remove_stripped, options)
```

References

1. http://python.org
2. http://wiki.python.org/moin/BeginnersGuide
3. http://www.aeryn.co.uk
4. http://tortoisesvn.tigris.org
5. http://docs.python.org/lib/lib.html
6. http://docs.python.org/lib/os-file-dir.html
7. http://docs.python.org/lib/module-optparse.html
8. http://scorefirst.com/articles.html

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 41 entries

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;
}
```

From Paul Smith <paullocal@pscs.co.uk>

As you were told, **`iterator`** is only like a pointer. **`iterator`**s usually have an overloaded unary **`*`** operator which gives you the object that the **`iterator`** is pointing to. So, **`*iterator`** will give you the object that **`iterator`** is pointing at and then **`&*iterator`** will give you the a pointer to the object that **`iterator`** is pointing at. You can then **`delete`** this (as your compiler lets you do).

However, while this will delete the appropriate string, it will not update the vector of strings to indicate that the string in question no longer exists. So, at some point your program will still try and reference the deleted string (in the last **`test.printVector`** if not earlier) – causing a probable crash.

It would be unusual for a compiler to be able to detect the problem with what you have done, because it is syntactically correct C++. (A good LINT program might have warned about it though.)

In order to update the vector properly, you should use the **`erase`** method of the vector class – i.e.: **`vector.erase(iterator);`**.

This will delete the object pointed to by **`iterator`** as well as updating **`vector`** so that it doesn't refer to the deleted string any more. The problem with this, in your current situation is that **`iterator`** will still point to the deleted string, so when you increment that in the loop in **`deleteFromVector`** it might skip one more element – taking it past the end of **`vector`** (almost definitely causing a crash, since **`iterator != vector.end()`** will always be true) or at least missing out an element.

ROGER ORR

Roger 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

The **deleteFromVector** loop needs to be re-written. I would use:

```
iterator = vector.begin();
while (iterator != vector.end())
{
  if ( *iterator == string )
  {
    iterator = vector.erase(iterator);
    // vector.erase returns an iterator that
    // points at the item after the one erased
    cout << "deleted: " << string << endl;
  }
  else
  {
    ++iterator;
  }
}
```

Alternatively, if you KNOW that all item items in **vector** are unique, you could possibly use a **break** – e.g.

```
for ( iterator = vector.begin();
      iterator != vector.end(); ++iterator)
{
  if ( *iterator == string )
  {
    vector.erase(iterator);
    cout << "deleted: " << string << endl;
    break;
  }
}
```

Other comments on the code:

- I notice you used **iterator++** in your loops. In general, especially with complex objects like iterators, using **++iterator** is more efficient, since it doesn't need a temporary of the object being incremented to be kept to be returned by the operator.

  The **postfix ++** operator (**iterator++**) returns the value of iterator BEFORE the increment operation, so the compiler has to:

  - make a copy of **iterator**
  - increment **iterator**
  - return the previously made copy of **iterator**

  The **prefix ++** operator (**++iterator**) returns the value of **iterator** AFTER the increment operation, so the compiler has to:

  - increment **iterator**
  - return the new value of **iterator**

  So you can see the 'make a copy of **iterator**' step is missing. Depending on the implementation of **iterator** this may be a relatively time consuming step that you can miss out as you don't need the return value of the increment operator.
- many people don't like the use of **using namespace std;** in your code, and prefer you to explicitly use the namespace everywhere it's needed – i.e. **std::vector<std::string>**. This makes it clearer which namespace is being used, rather than having to 'remember' that you're using the **std** namespace.
- I wouldn't have **iterator** as a member of **VectorTest**. The value of that variable is not needed to be kept between methods, so having it as a member of **VectorTest** makes its purpose less clear, people may think it is used between methods, rather than just locally, as well as using up heap memory unnecessarily.
- To make things a bit clearer, I would probably create a new type of vector of strings, e.g.

  ```
  typedef std::vector<std::string> stringvector;
  ```

  Then instead of having to type the verbose **std:vector<std:string>:iterator** every time you need an iterator, you can use **stringvector:iterator**.
- **printVector** could be a **const** method. You'd have to use **stringvector:const_iterator** instead of **stringvector:iterator** in the loop, but that would be a good idea anyway.
- **addToVector** and **deleteFromVector** should take **const** references as parameters, rather than value parameters. Using value parameters as you have done means that a copy will need to be made of the parameter for passing to the method. Since the methods don't change the parameters at all, they could be **const** references, to avoid this copy being made: **addToVector(const std::string &string)**.
- Having a parameter called **string** (e.g. in the **addToVector**) method is probably not a good idea, since that is also a type name.
- When you call **addToVector** you explicitly create a string in the parameter list, whereas with **deleteFromVector** you don't. There's no problem with either way, but you should try to be as consistent as possible. As the explicit string creation isn't necessary, I'd leave it out, it makes the code easier to read.
- I notice that you have implemented all the methods inside the **class** definition. This can make it a bit less clear what the **class** interface is. I would just put method declarations inside the **class** definition, and put the implementation outside the **class** definition.

  For instance:

  ```
  class VectorTest
  {
  public:
    void addToVector(
      const std:string &stringToAdd );
    ..........
  };
      void VectorTest::addToVector(
    const std:string &stringToAdd)
  {
    vector.push_back( stringToAdd );
  }
  ```
- it's good that you are using iterators to iterate through the vector in **printVector**. Some people would use the 'C' style **for (int i = 0; i < vector.length(); i++)** construct.

(PS – as to how it worked before you upgraded your compiler – I have no idea, I'm surprised at that!)

From Simon Sebright <simon.sebright@ubs.com>

Done on the last day, I thought I would make this one as succinct as possible, whilst still being useful. I often find that the SCC answers are lengthy and go into a lot of detail, often geared up for the audience reading C Vu, as opposed to the hypothetical student. From the code often presented, these students need more than anal amounts of detail; they need some higher-level insights, some things to think about, ways to approach problems. Let's try that and see what happens.

Right, basically your code only worked by fluke with your previous compiler, because you are mixing idioms. On the one hand, you use a **vector**, good, that's the container of choice for most circumstances. Vectors, and other STL containers manage their own memory. You pass them things by value, they worry about taking copies and where those copies live, and when they get destroyed.

You, however, have tried to get involved in the management of the vector's contents by calling the **delete** operator. This is bad, regardless of whether it compiles or not. The **delete** operator is the partner of the **new** operator, and you should only use them together. **new**, **delete**, **new**, **delete**, keep saying it. If you don't, all bets are off as to how the system will behave. This is a good example of the principle of Symmetry in software design: **open**, **close**, **start**, **finish**, **begin**, **end**, **new**, **delete**.

It appears as what you are trying to do is remove a particular string from the **vector**. As previously explained, a **vector** manages its own scoped objects, therefore you have to use the vector's functions to achieve your aim. Go away and read the documentation on **vector**, together with Scott Meyers' *Effective STL*. I'll give you a heads up that removing something from a vector is a two-stage process. See how you get on.

Right, you had a specific question. Why do you need to use **&*** to get an address of the object in the **vector** from a given iterator? First, don't forget, that you shouldn't generally be doing this for access to objects, particularly not as above to delete them. However, it is a valid question and sometimes useful. This is the case when wanting to access the vector's contents as a C-style array. Perhaps to pass this to an old-school api function, or if using a **vector** of **char**s for a string, then you get back the pointer to a **char** in the **array**, which is a C-style string.

Well, let's see. What type is the **iterator**? It's a **vector<string>::iterator**. Note that it is not a **string***. Have you read the documentation on **vector**? Well, you must do so. In it, you will find that **iterator** is a type available within **vector**, and that this type will support various operations for accessing the contents of the **vector**, and moving through those contents. So, operators ***** and **->** provide access to the content. Operators **++**, **--**, **+**, **-**, etc. provide the iteration. Nowhere does it in the documentation say what type the **iterator** is, only that it shall support these operations.

Different **stl** implementations are allowed to do what they want, as long as the code you write compiles and behaves (should you follow the rules). It just so happens, that the type **string*** would also support all these operators, as they are things you generally do with pointers. Your first compiler's implementation probably had a **typedef** such that **vector<T>::iterator** was **T***. However, your second compiler more than likely had a full-blown **class**, supporting the necessary options. Now, when you call operator **delete** on the **iterator**, you have a compiler problem because **delete** is expecting a pointer, you have given it an instance of the **iterator** class. Not allowed.

Should you really need the address of the object, you have to first get the object as a reference, and then take its address. So, **operator *** on the **iterator** gets you the object. You know this, because you have used it to output the values to the output stream. To get the address of an object, you use **operator &** on an instance. Combining the two, we have **&*iter**. If you put explicit parentheses in to show you what the compiler is doing to order the evaluation of these operators, you have **&(*iter)**. Of course, in the case where **iterator** is implemented as a pointer, these two things cancel each other out, but to write portable code, you must respect the requirements of **iterator** as documented, not what you find you can get away with.

Some by the way points:

- It is not a good idea to name variables the same as a class name. This is confusing, i.e. your **vector** and **iterator** members of this class **VectorTest**, and string parameters.
- The **iterator** member of **VectorTest** should not be a member. It is only used within the scope of functions, and its value is not used across those functions. Make it a local variable in functions where needed.
- It is idiomatic to pass non-built-in types by **const** reference. You have passed in your **string** objects by value.
- When calling a function taking a string, or as it should be a **const string&**, then you may use arguments as raw strings, because the right constructor of **string** will be invoked for you. You explicitly created **string** objects, e.g. **string("1")**. This is unnecessary here (I see you have done this when calling **deleteFromVector()**). Beware, though, if the constructor were marked explicit, you would need to do it this way. This is the case when the **class** writer does not want mistaken conversions to occur. For example, a **file class** with a constructor taking a **string** for the path would have that constructor as explicit, to prevent you from accidentally transforming strings into files.
- The data members of **test** are only **private** because of the default access level of classes is **private**. Better to explicitly mark them **private**. And, put them at the end. A **class** should exist for its users, in this case the public functions are of interest to your **main()** function. Make them most easily visible in the **class** definition.

Some praise:

- In contrast to much student code, this shows a good style in the breaking up of functionality, even small chunks, into reusable functions. Many students would be tempted to write the whole thing in **main()**. Keep writing small functions like this.
- As above, **vector** is the right default container to use, and using **iterator**s is the normal way to access it.

From Michal Rotkiewicz <michal_hr@yahoo.pl>

My first remark concerns sections in the **VectorTest** class: it's recommended to put the public section at the beginning of the class. This way class is more 'readable' – you can easily see the interface.

The **addToVector** function is fine but it may be improved in terms of performance. When you pass argument to a function by value it's copied by **copy** constructor. In general it's true but when we are considering temporary values it's a little bit more complicated.

Example:

```
void foo(myClass arg); ①
myClass object;
foo(object); ②
```

In this example **object** is copied with **copy** construct during ②. But if we have: **foo(myClass());** then **myClass()** might be constructed in the space used to hold the function argument ①. In this case **copy** constructor is not called. But it's implementation dependent. My compiler (gcc 4.1.1) did it whereas other compilers may create temporary object (using **myClass copy** constructor) and then initialize function argument. See C++ standard 12.2.2 example.

If we are passing arguments by reference **(void foo(myClass &arg))** the arguments are not copied and I recommend to use it in this case. But if you try to write:

```
void addToVector(string &string) ...
```

the compiler will complain that you can't initialize reference with temporary object. It would be legal if you wrote something like this:

```
VectorTest test;
string s1("1");
test.addToVector(s1);
```

In case of **test.addToVector(string("1"));** temporary object **string("1")** is created. As temporary objects are **const** you have to use **const** reference. Finally **addToVector** function should look like: **void addToVector(const string &string)**...

This way you are sure that argument is not copied. Instead of **printVector** function I suggest to overload **operator<<**. Having it you can use **cout** with **VectorTest** objects like with built-in types: **cout<<test;**

So let's write such operator:

```
friend ostream& operator<<(ostream &output,
                           const VectorTest &ref)
{ .... return output; }
```

I will be back soon to the body of this operator. At this moment I'd like to focus on one aspect: why did I define it as a **friend** function ?

The **operator<<** is called binary as it needs two arguments. One on its left side and second on its right side. We can define binary operator as a nonstatic member function taking one argument or nonmember function having two arguments. If the binary operator is defined inside the class **myClass**, its leftmost operand is of a type **myClass**. In terms of

VectorTest class it may look like: **ostream &operator<<(ostream &output)...** But in this we would have to write:

```
VectorTest test; ... test<<cout;
```

This is very ugly and we would like to use **cout** like with built-in types. Therefore we can't define **operator<<** as a member of **VectorTest** class. The second approach is global function:

```
ostream &operator<<(ostream &output,
                    const VectorTest &ref) ...
```

This way we can write:

```
VectorTest test;
....
cout<<test;
```

Inside the **operator<<** we have to use private member **vector**. To be able to do it, **VectorTest** class must declare friendship:

```
friend ostream &operator<<(ostream &output,
                           const VectorTest &ref);
```

But there is another subtle issue. The friend function doesn't have access to the **typedef**s and **enum**s defined in the class. In our case we don't have them so there is no issue but it's worth remembering how to cope with it. Function can use **typedef**s and **enum**s from the class if it is in the lexical scope of the class. Of course every function that is defined inside the class meets this requirement. So in our case we have to place **operator<<** inside the class but not as a member function. How to do it ? Simply by placing the **operator<<** inside class preceding it with word **friend**. Keyword **friend** prevents from treating the function as a member of the class.

```
class VectorTest
{ ....
  friend ostream &operator<<(ostream &output,
    const VectorTest &ref) { .... }
};
```

Let's deal with **operator<<** body. Instead of using **for** loop for printing **vector** element I suggest using **copy** algorithm with **ostream** iterator:

```
friend ostream &operator<<(ostream &output,
  const VectorTest &ref)
{
  output<<" contents:";
  copy(ref.vector.begin(), ref.vector.end(),
    ostream_iterator<string>(output," "));
  output<<"\n";
  return output;
}
```

Because **operator<<** returns reference to **ostream** object we can use it like this:

```
VectorTest test1, test2;
....
cout<<test1<<test2;
```

Finally we have **deleteFromVector** function. First I recommend to change the argument to **const string &string**. (See explanation to **addToVector** function.) To know what's going on at the line **delete iterator** let me say a few words about iterators.

Bjarne Stroustrup says in *The C++ Programming Language*: An iterator is an abstraction of the notion of a pointer to an element of a sequence.

To get an element that iterator points to we need to dereference it. Example:

```
vector<string> vec;
vec.push_back("one");
vec.push_back("three");
vector<string>::iterator it = vec.begin();
cout<<"First element = "<<*it<<endl;
it++;
cout<<"Second element = "<<*it<<endl;
```

Meanwhile **delete iterator** says: delete dynamically allocated iterator. There is no word about element that iterator points to. Example:

```
vector<string>::iterator *iterator =
   new vector<string>::iterator();
 ....
delete iterator;
```

So let's try another approach: **delete *iterator;** This construction says: delete dynamically allocated object that iterator points to. Example:

```
vector<string*> vec;
vec.push_back(new string("one"));
...
vector<string*>::iterator iterator = vec.begin();
delete *iterator;
// calls string("one")
// destructor and deallocates memory.
```

But above code doesn't remove element from vector; So after **delete *iterator** vec size is still 1 and pointer **vec[0]** points to the deallocated memory. To remove vector element we have to use **erase** function from **vector** class. This function takes one argument: **iterator** pointing to the element that must be erased. Once element is erased function returns **iterator** to the next element. So the code may look like:

```
for (iterator=vector.begin();
iterator!=vector.end();
   iterator++)
{
  if (*iterator == string)
  {
    iterator=vector.erase(iterator);
    cout<<" deleted: "<<string<<endl;
  }
}
```

If you write only **vector.erase(iterator)** then you have undefined behaviour as **erase** invalidates all iterators in the range **[iterator, vector.end()];** Therefore it's important to have valid **iterator** after **erase** call. That's the reason why **erase** function returns the next valid iterator. We may apply the changes I wrote to the function **deleteFromVector** and leave it but I recommended to use algorithms instead of writing loop. Vector class has function **erase(iterator1, iterator2)** that erases all elements from the range [iterator1,iterator2) and returns **iterator** to the next element.

Unfortunately this function is not sufficient to remove all elements with given value as they might not be in the continuous space exactly between two iterators. To do it we may use remove algorithm which pseudo declaration is as follow: **iterator remove(iterator1, iterator2, val);** Despite the fact that this algorithm is named **remove** nothing is removed in fact! We may describe the behaviour of this function as follow: Move all elements from the range [iterator1,iterator2) having the value 'val' to the end of collection and return iterator to the first of the elements moved to the end. Example:

```
vector<int> vec;
vec.push_back(1);
vec.push_back(5);
vec.push_back(6);
vec.push_back(5);
vec.push_back(4); //vec has elements 1 5 6 5 4
```

If we call:

```
vector<int>::iterator it = remove(vec.begin(),
  vec.end(), 5);
```

then the **vec** looks like: **1 6 4 5' 5''** (**'** – is for notational purposes) and it points to the **5';** To get rid of the elements we have to call **erase**: **vec.erase(it,vec.end());** We may write it at one line as well:

```
  vector.erase(remove(vector.begin(),
    vector.end()),string));
```

(as we don't iterate we are not interested this time in the **erase** return value). Comparing to original code we won't see 'deleted: ' sentence each time element is deleted but I think it's minor issue ;). If you really want to have such output read about **for_each** and function objects :)

Finally the code looks like:

```
#include <vector>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

class VectorTest
{
public:
  void addToVector (const string &string)
  {
    vector.push_back(string);
  }
  void deleteFromVector(const string &string)
  {
    vector.erase( remove(
      vector.begin(),vector.end(),string),
      vector.end());
  }
  friend ostream &operator<<(ostream &output,
    const VectorTest &ref)
  {
    output<<" contents: ";
    copy(ref.vector.begin(), ref.vector.end(),
      ostream_iterator<string>(output," "));
    output<<"\n";
    return output;
  }
private:
  vector<string>::iterator iterator;
  vector<string> vector;
};

int main()
{
  VectorTest test;
  test.addToVector(string("1"));
  test.addToVector(string("2"));
  test.addToVector(string("3"));
  cout<<test;
  test.deleteFromVector("2");
  cout<<test;
}
```

From Ric Parkin <ric.parkin@ntlworld.com>

When I first saw this code, I scanned it, immediately saw the simple mistake the student had made, and moved on. First appearances can often be deceptive, and a more in depth look revealed a surprisingly rich set of problems, style, design, use of the standard library, and many other things.

So first, to the stated problem: the author is trying to remove a **string** from a **vector** by calling **delete** on the **iterator** 'pointing to' the string to remove. The author themselves almost had the problem solved when they describe an iterator as only 'like' a pointer, so should use **&*** to get the pointer back.

What pointer? There are no pointers to be seen in the code, and no call to **new** to match with the **delete** rings a major alarm bell: **new** and **delete** should always be matched up, and in clear code this is often enforced by class design that wraps up the allocation/deallocation, or makes the mismatch **very** clear, eg by quickly wrapping in an **auto_ptr**.

Also, saying iterators are just 'like' a pointer isn't quite accurate, and the wrong way around – an iterator is an abstraction of ***some*** of the things you can do with a pointer. So a pointer can be used as an iterator, but an iterator cannot always be used like a pointer.

This then explains the change in compiler stopping their code from compiling: previously the **vector** iterator was implemented as a pointer to the contained item so calling **delete** just happened to compile, and now it's implemented as a class that represents the item 'pointed at' so it no longer will. This is good as the original code called **delete** on a pointer to the middle of the **vector**'s owned memory, which invokes undefined behaviour. The big mystery for me is how the original code ever got the reported output!

So, fix the problem: how to remove a value from a container? Vector has a member function **erase**, that takes an iterator to an item and removes it from the container, so replacing

```
  delete iterator;
```

with

```
  vector.erase(iterator);
```

will do the trick.

So from a trivial problem immediately into more interesting ones: could you use **const_iterator**? After all, you're not modifiying the string anywhere, just the container. The answer is a Pollard-like 'No but Yes but No'. No, because the existing standard says that **erase** takes an **iterator**, not a **const_iterator**. But Yes, because this has been recognised as being too restrictive and is likely to change in the future (see DR 180). But No after all, as that change hasn't been agreed yet, and if it makes it into a future revision not all libraries will get that change immediately, and portable code could be better off not relying on it.

Looking a little further out, the surrounding loop looks for a value to remove. There a few minor stylistic things about this loop that I would do differently: I'd use pre-increment instead of post, save the value of **end()** in a variable to avoid calling repeatedly, and use local variables instead of the class data member iterator.

```
for ( iterator i = vector.begin(),
      e = vector.end(); i!=e ; ++i )
  if ( *i == string ) {
    //...
    break;
  }
```

But even better would be to not code for a loop at all: this is a simple find algorithm, and the standard provides one in the header 'algorithm'.

```
  iterator = find( vector.begin(), vector.end(),
    string );
  if ( iterator != vector.end() )
  {
    vector.erase( iterator );
    cout << "delete: " << string << endl;
  }
```

Note that you have to check it was actually found before calling **erase**. In the Standard Sequence Requirements in Table 67, it says **erase(position)** 'erases the element pointed to' by the iterator, which implies it ***should*** point at an element, although the range version could well take an empty range.

This raises the question of what is this function's error handling strategy. For example, if I ask it to remove a value that isn't there, or what if there are ***multiple*** values there? The current behaviour is to do nothing, and only remove and report one value, respectively.

If we'd like to remove all values that match, then the **remove** algorithm is ideal for it. This is a slightly weirdly named algorithm that generates lots of questions as it doesn't actually do any removing, instead it shuffles all values that are going to stay to the start of the range, and tells you the start iterator of the remaining values. You can then use this as the start of a range to erase the values left over using the overload of **erase** that takes a range:

```
erase( remove( vector.begin(), vector.end(),
  string ), vector.end() );
```

There is another loop in **printVector** that can be replaced with an algorithm, **copy**, using an output iterator to send the values to a stream:

```
copy( vector.begin(), vector.end(),
  ostream_iterator<std::string>(ostream) );
```

So to more minor stylistic issues.

Overall, the student is trying to do some good things:

- using the standard library for vectors and strings rather then reinvent the wheel
- clearly named methods so you know what the code is trying to do
- some testing to clarify their understanding and make sure the code does what they think it does
- passing in a **ostream** as a parameter to **printVector**, so can send the output anywhere.
- **int main()** is nice to see: a certain popular IDE's code generator often produces **void main()**, which isn't allowed by the standard.
- even though the final **return 0** is redundent as it'll be added in automatically, I think it is better to always be in the habit of returning a value. This is so that you don't forget to return 0 in some other function, which would trigger undefined behaviour, and is clearer to readers that you really do want to return 0, and didn't just forget.

But some things aren't so good:

- all the methods have names that include the word 'Vector', despite them being on a class called **TestVector**. This is redundant and leads to overly-verbose code with lots of repetition. **add**, **print** and **remove** should be sufficient.
- the variables are named identically to their type which is rather confusing, and leads to odd declarations such **string string;**.
- the **iterator** data member isn't really part of the class state – every time it is used it is reinitialised so its old value is never needed. It should be replaced by a local variable where needed
- once that is gone, the only data member is a vector, so **VectorTest** is in fact a simple wrapper around a vector of strings that provides some long-named methods. Another hint is that the name **VectorTest** isn't particularly revealing, which suggests to me that it isn't really modelling anything in particular, and shouldn't be a class at all. I would prefer it to be written instead as free functions that took a vector of strings as a parameter.
- **using namespace std** isn't bad per-se in a source file, but would be a bad idea in a header. Alternatives would be to fully qualify names, or pull in just the names used, or put the using in each function if it's too hard to read with full names.
- a **typedef** for the vector of strings would be a good idea. This also simplifies the names of nested types like iterators, i.e.

  ```
  typedef
     std::vector< std::string > StringVector;
  StringVector v;
  StringVector::iterator i;
  ```
- an **operator<<** would be nice for this class, even if it only called **printVector**
- **printVector** should be a **const** method
- while **printVector** gets given a stream for output, **deleteFromVector** has **cout** hardcoded. If you want the same one used throughout, perhaps it ought to be passed into the constructor.
- there is no **copy** constructor, **operator=**, or **destructor**. This means the compiler will generate them, and if the iterator member is still there, the first two will do the wrong thing. Instead, declare them private but non implemented. The destructor is fine.
- only **iostream** is in the list of headers, but **endl** etc are used. **iostream** is only required to pull in the names for **cout** etc, but isn't required to pull in anything else. This is only compiling because on this particular library enough other headers are pulled in, but another library might not, so should also include **ostream** to actually use the stream and **endl**.
- **endl** actually does two things: output a newline, and flush the stream. In most cases, **cout << '\n'** is all that is wanted or needed and will be a lot faster if that is a bottleneck.
- strings are being passed by value. Many string implementations are reference counted, or cheap to copy, but convention is to pass anything non-trivial by **const** reference unless there's a good reason why not.
- in **main**, **addToVector** is called by explicitly making strings from string literals, whereas **deleteFromVector** relies on an implicit conversion.
- **deleteFromVector** and **addToVector** could take a **const char*** and avoid some of the overhead of creating and copying a string. This depends on whether the client code is expected to be using modern C++ which could assume that string is universally used.
- incrementing iterator is using post-increment. As the previous value isn't needed, it ought not to be asked for, so pre-increment should be preferred. This can be argued on opimisation issues, but I prefer clarity – say what you ***mean***.
- many people write the public parts of a class first, and the private last (the rationale is that people want to use a class more often than they maintain it, so that part should be first)
- it isn't clear what **deleteFromVector** should do if the value isn't there or if multiple values are there.
- there aren't any comments. Fortunately, the code is generally clear enough that not many are actually needed, but some notes on any contracts and behaviours would be useful, especially around **deleteFromVector.**

One of the interesting things about this code is that it is on first sight pretty good with one mistake, and yet there are all sort of simple things that feel oddly wrong when delving in deeper. I blanch to think about the amount of my own code written under even slight pressure that would have a similar effect on me if I looked at it in depth again.

Commentary

After a slow start – most ACCU members were presumably enjoying a well-earned summer break – more entries arrived just before the deadline; between them the entries cover almost all the points I'd thought of (and some I hadn't) about the code provided.

I felt that some of the changes suggested might be slightly too advanced for the student, who was struggling with understanding object ownership. It can be hard to know how much to cover when trying to help someone with a problem; especially in the SCC when the 'student' is absent!

As most entries pointed out, the fact that the code works at all is a bit of a fluke (it is down to the low level implementation details of memory management and string class layout for the compiler involved). In my experience most programmers when faced with code that sometimes works tend to assume the code is fine and blame the environment in which the program fails. ("It only crashes in release builds – must be an optimiser bug" or "It works on Windows, must be a gcc bug").

The advice in this particular case that the call to **`delete`** is not balanced by a call to **`new`** is good – it helps to identify this bug but also gives the programmer a rule of thumb that may help them in the future.

I agree with the entrants who wanted an **`operator<<`** for the class, but do have a preference to make this a non-**`friend`** inline function calling the (existing) **`print[Vector]`** method. I like this pattern as, in a class hierarchy, the **`print`** method can be made virtual and hence **`operator<<`** automatically behaves polymorphically.

The use of comments is a debate on its own (as recent discussion in C Vu and elsewhere has shown!) I thought the code was clear without comments as the method names were self-documenting, my only minor problem was with the **`deleteFromVector`** method where it is unclear what behaviour to expect if duplicate elements are located.

The winner of SCC 41

The editor's choice is Ric Parkin.

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

Student Code Critique 42

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

Despite appearances, this code is not from a student but extracted from an existing code base. There is a string class that sometimes gives unexpected failures. Here's a simple test program that asserts with one compiler – but only in debug – and works with another. Please explain the assert failure and critique the code.

```
#include "MyStr.h"
#include <assert.h>
int main()
{
   MyStr s;
   assert( s == "" );
   s = "10";
   assert( s == "10" );
}
```

The header file for **`MyStr`** is:

```
 #include <string.h>

class MyStr
{
public:
  MyStr(char const * str = 0);
  MyStr(const MyStr& str);
  const MyStr& operator=(const MyStr& str);
  ~MyStr() { delete [] myData; }
  bool operator==(const MyStr& d) const;
  friend bool operator==(const MyStr& lhs,
    char const * rhs);
  friend bool operator==(char const * lhs,
    const MyStr& rhs) { return rhs == lhs; }
  void alloc(unsigned N) const
  {
    MyStr *nc = const_cast<MyStr*>(this);
    char * od = nc->myData;
    nc->myData = new char[N];
    nc->mySize = N;
    copyStr(myData, od);
    delete [] od;
  }
  void copyStr(char* d, char const* s) const
  {
    if(s == NULL)
       d = NULL;
    else
       strcpy(d,s);
  }

  operator char *() const { return myData; }
  // more methods unused in test not shown
private:
  char * myData;
  unsigned  mySize;
};

inline MyStr::MyStr(char const * str)
: myData(NULL), mySize(0)
{
  if(str == NULL) return;
  unsigned newlen = strlen(str) + 1;
  alloc(newlen);
  copyStr(myData, str);
}
inline MyStr::MyStr(const MyStr& str)
: myData(NULL), mySize(0)
{
  if(str.myData == NULL) return;
  unsigned newlen = strlen(str.myData) + 1;
  alloc(newlen);
  copyStr(myData, str.myData);
}

inline const MyStr& MyStr::operator=(
  const MyStr& str)
{
  if(str.myData == NULL)
  { myData = NULL; mySize = 0; }
  else
  {
    unsigned newlen = strlen(str.myData) + 1;
    if(newlen > mySize)
      alloc(newlen);
    copyStr(myData, str.myData);
  }
  return *this;
}

inline bool MyStr::operator==(const MyStr& d)
  const
{
  if(myData == d.myData) return true;
  if(myData == NULL || d.myData == NULL)
    return false;
  return !strcmp(myData, d.myData);
}
 inline bool operator==(const MyStr& lhs,
 char const * rhs)
{
  if(lhs.myData == NULL && rhs == "")
    return true;
  else if(lhs.myData == NULL) return false;
  else return
!strcmp(lhs.myData, rhs);
}
```

Prizes provided by Blackwells Bookshops & Addison-Wesley

Francis Scribbles 27

Francis Glassborow brings us the final installment of his regular column.

Time wasting

I received a reminder to renew my subscription to the *New Scientist* this morning. Among the methods available was via the Internet. That seemed the quickest and easiest way. It was not. Their data capture software denied that there was a subscriber with that account number either named 'Glassborow' or with 'OX4 1PA' as a postcode (validation was either by last name or postcode). After trying all the options (with/without spaces in postcode, with/without use of upper case) I finally gave up and renewed by phone.

Next I received an email from the BSI training program that attempted to tell me how to successfully take the course on BS0 (BSI's standard for standardisation and now required study and examination for all participants in standards work above the level of a panel). Despite all their advice, my machine absolutely refuses to let me see even the first word of the course. Yes, it is some form of machine/OS related problem because I can start the course on my wife's machine.

Even without this problem, there are several serious irritants with this 'distance learning' programme/program. The start (on my wife's machine) list speakers as a hardware requirement and then adds the comment in red that there is no audio element. Interestingly, when I disable my Bluetooth headset connection and reset my machine to use its speakers and soundcard, I no longer get an initialisation failure but get no other progress.

I should mention that this whole package requires the use of Internet Explorer with pop-ups enabled. I can see no sign that it will work other than with some version of Windows (though that includes some workstations as well). I can see no support for Linux.

This is not the first time that BSI's IT usage has seemed deficient. I would care less if I were paid to work on Standards. I am not, nor are any of the other people who do the real work, so you can imagine my resentment at my time being wasted by people who are paid to do their work.

We are now well into the twenty-first century and major organisations should be able to use modern technology a great deal better than the evidence suggests they are doing. And, yes, this does relate to programming as the whole of this sorry affair relates to competence (or lack thereof) in writing the software that enables people like me to get on with our lives instead of having to try to solve problems created by bad software.

If BSI likes to reply to the above criticism, I am sure the C Vu editor would be happy to publish.

Significant dates

Last month I reached my 100th birthday. Yes, I know that is a stretch but this is a magazine for computer programmers, so octal is reasonable (broad grin). That event reminded me of other significant ones.

The first volume of C Vu spanned most of two years but there were only five issues. The second volume was a quarterly and I took over as editor for the second issue. From volume 3 onwards C Vu has been a bi-monthly.

If you do the arithmetic carefully you will realise that issue 18.1 was the 100th (denary this time) issue. Over those 100 issues, there have been well over 200 contributors and many millions of words. Granted that the quality has varied but most, if not all, our writers have added new skills while helping readers increase their understanding.

Looking back at the early days, I notice that we published much more for the less experienced in those days. We also published much more by less experienced writers (of course they rapidly became experienced and many went on to write elsewhere). The stable readership is one cause of this. The average length of membership is approaching 10 years, which is a remarkable achievement for any voluntary organisation. This is a good indicator that ACCU is achieving something that many members value.

I wonder if it is time that we ran a beginners' corner on a regular basis. I know that the Internet has provided an excellent alternative to the printed word but I think we should be very clear that less experienced programmers are very welcome amongst us.

A competition

With this in mind, I am offering a prize for the best 'essay' describing the fundamental elements of programming. I will be the sole judge of contributions and I hope the administration will place all entries on our website. I have deliberately left the length open. However, the principle criteria on which I will judge entries is technical accuracy, sticking to the basics and, most importantly, clear simple writing.

The prize? A signed copy of *Exceptional C++ Style* (by Herb Sutter) and a signed copy of *You Can Program in C++*.

Another significant date

I cannot find exactly when ACCU started (under the name CUG(UK)) but it was some time in 1987. Forty-nine people joined that year (ten of them are still members today, and one other rejoined after his membership had lapsed for a few years). That means that ACCU will be celebrating its 20th anniversary some time around its Conference next year. I wonder if anyone remembers the actual date of the foundation meeting.

Which reminds me, the first ACCU Conference was a two-day event that coincided with the WG21 meeting in London in 1997. The second day consisted of entirely of talks from Bjarne Stroustrup, Bill Plauger, Tom Plum and Dan Saks. The most memorable of those was the one from Dan Saks that showed him to be one of the world's greatest technical presenters. Sadly, his family commitments and health problems have kept him away from recent conferences. Earlier this year he was chosen as a Democratic candidate (for Ohio) for Congress. Unfortunately, his health has again deteriorated and he has had to withdraw his candidature. Apart from taking this opportunity to wish him a full recovery, I am also hoping that we might entice him back to a reunion of that first Saturday's speakers.

Some of you will realise that we missed another significant date this April, it was the 10th ACCU Conference. So much has its success grown that this year we had to close the bookings (so make sure you are not one of those left on a reserve list next year). However, being programmers and so counting from zero, the next conference is the 10th and the ACCU conferences are ten years old next April. I think that is one more thing of which we can all be justly proud.

FRANCIS GLASSBOROW

Francis is a freelance computer consultant and long-term member of BSI language panels for C, C++, Java and C#. He is the author of 'You can do it!', an introduction to programming for novices. Contact Francis at francis@ronbinton.demon.co.uk

Symbian C++

Someone living close to me emailed me to say that he was using Symbian C++. Unfortunately, I lost the email. I would be happy if he could resend it.

I would also like to contact anyone who is using the Eclipse based Carbide C++ that is provided by Nokia. Even better would be a review of it.

I wonder if anyone can explain why the mobile phone emulator used by Symbian for development on a PC is so unbelievably slow at starting up. What makes it worse is that every time you want to test a program on the emulator you have to reload it.

I think its start-up time is bang in the middle of the worst option for a program. I classify programs as:

1. Avoid running during the working day
2. Have a meal break
3. Have a coffee break
4. Spend time cursing time wasted
5. I know something happened
6. Seamless

The start time for the mobile emulator is a 4, too short to do anything else, too long for happiness.

I am told that the 9.x releases of the Symbian OS support standard C++ exceptions. Has anyone tried using them? It seems to me that if it does then developers should start moving towards using the Standard C++ Library. This would make it much easier for competent C++ programmers to move to writing for the Symbian OS.

Problem 27

```
struct foo {
  int i;
};
int main{
  foo * f_ptr = new foo::foo();
  foo::foo f;
}
```

Comment on the status of the code above in respect of the 1998 C++ Standard, the corrected 2003 version and the proposals for the next full version.

Problem 26 commentary

```
#include <iostream>
int main() {
  int i(0);
std::cout << "Please type in a number "
"between 0 and 255: ";
  std::cin >> i;
  std::cout << i * i;
}
```

Please identify all the possible problems with the above program (both compile time and execution time).

From: "Balog Pal" <pasa@lib.hu>

Compile time:

- Only **`<iostream>`** is included: we need **`<ostream>`** for **`>>`** and **`<istream>`** for **`<<`**. It may work on some compilers if they happen to include those from **`<iostream>`**, but nothing in the standard indicates we can build on it.

  (In practice it works on almost all compilers, which does not help novices to learn the requirement. FG)
- Missing **`return`** in **`main()`**. It is *not* a problem with a standard-compliant compiler (leaving **`main`** without **`return`** has implicit **`return 0;`**) but some compilers may issue a warning or error instead in practice.

Runtime:

- The prompt is written to **`cout`** without **`endl`** or **`flush`**, so the text may sit in the buffer when the user is supposed to enter the number; then if he actually figures out to type something and hit enter the prompt will appear.

  (**`std::cout`** is tied to **`std::cin`** by default so the former is flushed before execution of the latter. FG)
- **`>> i;`** may fail and then **`i`** stays at 0.
- We bypass the ‘all input is evil’ principle and just use **`i`**. If the user types some large number, **`i*i`** will produce overflow that is undefined behaviour. There is no attempt to enforce the 0-255 range required in the prompt.

Aesthetic problem:

- **`i*i`** is just dumped without telling what it is, and no newline at end.

  (Curiously, Balog missed the fact that this time the output might not be displayed before the program ends. FG)

Cryptic clues for numbers

Last issue's clue

Jeans? Maybe. ISO Labour Day? Definitely in Japan. (3 digits)

Some day I will learn what makes a popular clue. I had only one response to this clue for 501 (ISO date format, used in Japan, are ordered as year, month, day).

From Ray Butler

Ah, SO! Francis-san,

Is that really what you meant? I cannot believe we are stooping to national stereotypes :-)

I never noticed that as a possible reading of the last sentence. It was just a reference to the fact that Japan uses ISO Standard date format (year/month/day) so May 1st becomes 05/01.

Anyway, here is my alternative clue for 501. Hope it’s not too baffling this time:

> When in Rome, Di shows what she is equal to, only a minute after the office has closed.

Ray wins a signed copy of *You Can Program in C++* if he would like to claim it.

This issue’s clue

> Dial for a cab? All too easy on your mobile!

Hanging up my keyboard

The main reason that I am telling you this is that this is the last regular column that I intend to write for now. Other things in my life require my attention and I am not able to commit myself to meet future deadlines.

This is the 100th issues of C Vu to which I have contributed. Over those 100 issues I have written considerably in excess of a million words including reviews of more than 1000 books.

I intend to continue writing and reviewing but I want to feel free to miss an issue from time to time rather than feel under the hammer of yet another deadline.

I have it in mind to write a few articles for new C and/or C++ programmers dealing with common mistakes and misconceptions but I am not sure when I will get round to it.

Tidying up

Over the last two months I have stripped out 75% of the technical books that have sat faithfully on my office shelves. I decided the time had come to weed out the books that had gone out of date whilst I failed to find time to study them again.

Mailbox @ C Vu

Your letters and opinions.

Dear CVu,

Recently I noticed an accu-contacts email mentioning job opportunities at an online betting company. I'd recently seen the topic of the growing problem of online betting mentioned in a recent Telegraph article, so I thought it a responsible thing to highlight the possible impact of our work as professional software engineers on the wider community we live in. Hence I posted a response (soon to be rejected) to accu-contacts mentioning the newspaper article. I include an excerpt here:

Excerpt (from "Labour gambles away its principles" By Leo McKistry (filed: 22/07/2006) at www.telegraph.co.uk):

... In Britain, there are mounting concerns over the growth in gambling addiction. It is far less intimidating for women and novices to start betting on a screen than to enter a traditionally masculine betting shop or casino. Moreover, parting with large sums by credit card or bank transfer is less physically painful than using cash. In May, one charity, GamCare, reported a 40 per cent rise in the number of people seeking help with problem gambling.

Only last week, addict Bryan Benjafield was convicted of stealing more than £1 million from his employer to pay for his online habit, which was revealed in court to be costing up to £17,000 a day. Thanks to Benjafield's irresponsibility, his employer has now gone bust. ...

I received this reply from the moderator:

accu-contacts isn't really the place for this sort of disucssion, so I have rejected the message.

Now, I don't like the idea of censorship: the ACCU I joined was an organisation for programmers by programmers, and at the back of my mind was the idea "Who are you to tell me what I should or should not discuss". This is what I replied:

I'm surprised at censorship within the ACCU.. Are we afraid of open discussion ?

Is it out of order to discuss the implications of whether we, as software engineers, elect to work in particular industry sectors ?

If this is not the correct list, then pls tell me which is the correct list, and allow me to broach the topic on that list.

This led to my being informed that the accu-general is a more appropriate mailing list.

Now, as I mentioned to the original poster (OP) of the job advert, I think it is right that job opportunities be shared amongst members – this is a natural and positive activity for ACCU. But as I also mentioned to the moderator, I think it is wrong to simply reject a followup mail from a member that points out to readers of the orignal ad, issues that may make them think twice about applying for that particular job if they have a choice.

In particular, I feel moderators should at the very least voluntarily offer an alternative mailing list without being pressed. Had I not explicitly asked the moderator what is a more appropriate list, would I have been told ? Further, when I went to the website and clicked on the link for the charter of the accu-contacts mailing list, it came up empty. (The charters for two other mailing lists have content, in contrast.)

And, for those that appreciate the funny side, the first words in the moderator's response to my 'censorship' challenge were: "You haven't been censored."

Kind regards, (Mr) Fazl Rahman

ps It may be relevant that the moderator and OP were one and the same person; maybe this is pertinent.

I think the key here is actually contained in the PS - "It may be relevant that the moderator and the OP were one and the same ..."

This fact gave, unfortunately, a misleading impression – that the moderator, perhaps stung by criticism, was trying to strike down a legitimate discussion. This was not, and is not, the case.

While ACCU does not take a position any particular firm or industry sector, we do aim to promote professionalism in all its forms, which obviously includes ethics and behaviour. That being the case, however, the place for such discussion is not the accu-contacts list. That list exists to provide a place for members to list jobs to people or people available for jobs, and that's all. There are other venues for discussion of jobs offered or the companies behind them, including accu-general or here in CVu.

The rejection email you received was rather terse. A longer explanation might have avoided any misunderstanding. Whether to accept your email to accu-contacts was discussed by the committee, and we felt that we had to stick to the mailing list charter*. I'm sorry at the offence caused, and on behalf of the committee apologise for it.

* I don't know why the charter should have been inaccessible. Certainly no one else has reported any difficulty. (See http://accu.org/index.php/mailinglists/charters/contracts_charter)

Jez Higgins
ACCU Chair

Francis' Scribbles (continued)

I used the criterion of getting grid of anything that I was not certain that I wanted to keep, instead of my previous criterion of keeping everything that I hoped I might get to sometime. I suspect that I have still kept more than I should have.

I am not yet ready to close the door on my way out but I am certainly feeling the time is coming when I should be doing that. The last 18 years of writing and reading have been immensely educational.

I will go on reading and doing some writing but I find that I have lost my appetite for technical reading. A dozen books a year is more than enough these days. That may seem a lot to some of you, but during the 1990s it was not uncommon for me to read two technical books in a week.

The future

I think we are living in times where change is normal. Some of the changes enrich our lives but others are just the opposite. We can listen to great music played by great practitioners for a pittance, but too few actually enjoy making music.

We are approaching a time when I fear software will be widely available for a pittance but few people will understand the process of producing it.

I think we have a job to not only write good software but to help the mythical 'man in the street' understand how it is done. We are in serious danger of producing an era of magic with its elite magicians. That, in my opinion, does not lead to a healthy community. We have the irony of living in the greatest age of technological development this world has ever had, whilst being increasingly drowned by ignorance and fundamentalism.

How do you think an intelligent machine would view the human race? That is a serious question. In the lifetime of many readers, computers will match and then exceed the complexity of the human brain. Isaac Asimov invented the 'Three Laws of Robotics', but what happens when the machines are threatened by human fundamentalism?

OK that is the end of my speculation. I will try to stick to technical articles in the future. ■

Standards Report

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

Do you care about the C++ and C programming language standards? How much?

Those of us who volunteer our time and money to standards meetings are proud of what we have produced. For example, I glow with modest pride when I can point to the footnote in the C++ standard for which I drafted the original language. It is very far from being an important footnote, but it is a tangible contribution which I have made.

But everyone who is reading this also has the opportunity to say proudly, '*I* helped to make C++0x a success'. And it costs almost zero time and only a modest amount of money. If you haven't yet renewed your dues to ACCU, please tick the 'Fee for International Standards Development Forum' box when you do. If you have already renewed, please go back to www.accu.org and add on support for ACCU's standards effort as a single item. All it costs is £21, and this year your money has a direct link to making the C and C++ committee meetings a success, since ACCU are hosts for the international meetings next April. If only 200 members tick the ISDF box, the total would cover over a third of the total cost of hosting the two meetings, of which the main cost is the rental of the meeting rooms themselves. We are working to enlist corporate sponsors, but a substantial show of support by ACCU members would go a long way to ensure the association has the resources to take on this responsibility.

Even better, if you can persuade your employer to become a meeting sponsor, even on a modest scale, the organisers will be duly grateful. After all, if companies are willing to pay to have good programs written in C or C++, they should have some interest in seeing that those programs are as well-defined and portable as can be ensured by a good language specification.

Having the committees meeting in such close proximity to the ACCU conference in Oxford provides a direct benefit to the conference. Some number of delegates will be speakers at the conference, and some others will simply be around at the same time, and available for conversation. It is a chance to match a real face and personality to the folks whose books fill your bookshelf and from whom we have all learned so much.

If you would like to sit in on some portion of the international committee meetings, just to see what happens at these events, I will be glad to arrange for this to happen. And there is a standing invitation to join the UK standards panels, if you are interested. There is no cost, and no remuneration except the enjoyment of deeply technical discussions – which to some of us is rich reward indeed.

Please write to standards@accu.org for more information. ■

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

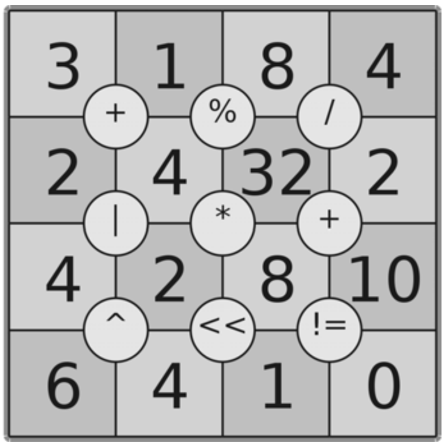
Mental Gymnastics

Pete Goodlife provides us with a little more mental exercise...

The puzzles in C Vu 18.3 got a lot of people's mental faculties firing, and some have asked for more of the same – you gluttons for punishment! So here you are.

See what you make of this little brain teaser. Send your answers to me at pete@cthree.org. I'll print your answers in the next issue. The first person to come up with the correct answer will receive untold fame and glory. And a smug feeling of satisfaction.

Good luck!



What's the longest sequence of numbers?

You can't reuse a square as another result in the sequence.

```
while (you care about code)
{
    read ( cvu && overload );
}
do(it);
```

Bookcase

The latest roundup of book reviews.

C Vu has had, for a very long time, a reputation for top-notch reivews of programming and technical books. We have a great relationship with many publishers, some of whom publish quotes from the reviews on their literature.

We are independent of any publishing company and as such have been know to slate a book from one publisher and, in the same breath, praise another book from the very same publisher. C Vu and its reviewers are respected for their impartiality and independent knowledge.

Books for review are available to any member who requests one. The current list of books available is in the members' area of the website at http://accu.org/index.php/newbooks

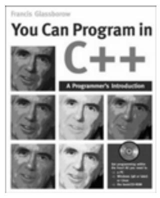
The range of books covers C++, Java, C#, PHP, Ruby, project management, tools, design patterns, and more. To review a book all you need is to send an email and pay a small admin charge of £5 to cover postage.

By contributing a book review you are contributing to the greater knowledge of the membership. Books are expensive and the last thing anyone wants to do is spend upwards of £30 on a book which is an utter turkey! When you review a book the worst that can happen is you lose a fiver. You'll probably gain some grateful friends, and if the book is "Not Recommended" your next book is free. What could be fairer than that?

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

C++ Programming

You Can Program in C++



by Francis Glassborow
Publisher: John Wiley and Sons Ltd (12 May 2006)
ISBN: 0470014687

Reviewed by: Richard Elderton

I started learning C, then C++ back in 1997 so you'd think I would be pretty expert by now wouldn't you? Wrong! There were problems. I was doing it on my own with only books and C Vu magazine to turn to for help. At first it was like floundering about in a deep quagmire – with approaching nightfall. My first compiler was the Zorland C Compiler Ver 2, but I quickly decided that C++ was a better language. My day job is cabinetmaking and restoration (www.relderton.co.uk) and I was doing this as a mind-stretching exercise. I wanted a pure, powerful and versatile language that would endure, having played with Sinclair Basic and Z80 assembly language. Of course assembly language is the purest approach to computer programming, but one has to compromise if one wants to produce powerful programs before doomsday. C++ fits the other criteria but it also carries the weight of history (the need of new developments to build on previous work rather than start afresh). The next IDE (integrated development environment)/compilers were Borland Turbo C++ 3.0 for DOS, then Borland C++ Builder 5 for Windows, then a rapid retreat back to Borland Turbo C++. I was reading avidly all the time but had to content myself with the poetic qualities of computer speak if I failed to comprehend its deeper meaning. The trouble was that the various books, compilers, articles etc. in the late 1990s were deeply incompatible, but I was quite innocent of this.

Then came FGW's *You Can Do It!* This was like manna from heaven. The book came with an easy to use Windows IDE/C++ compiler (Al Stevens' Quincy), all the programs could be made to work with very little trouble and the author's instructions were easy to follow. At last I was making real progress. The book is an introduction to programming using C++ only as a vehicle. All elements of this publication are compatible including the book's website and the network of mentors who give support to readers via email. My mentor was Roger Orr, who's genius never ceased to amaze me. (To my family Roger is known as 'the archangel Gabriel', whilst FGW is known simply as God.)

You Can Program in C++ is FGW's second book in what will become a series and it is a tour de force. It progresses from introductory work through to a level of proficiency that will allow the student to develop quite powerful yet easily constructed programs. It is designed for those who have some prior programming experience (in any other language). The author takes pains to explain how those proficient in specific other programming languages can easily make wrong assumptions regarding many aspects of C++. I found those sections particularly interesting because they helped to show C++ in a wider context.

FGW's writing style is what I would call direct. He holds one's attention because it seems that he is talking directly to one. The delivery does not consist solely in facts but it contains elements of programming history an personal experiences. This helps to bring the subject alive and give it depth and colour, making the details easier to remember. It goes without saying that an author has to know his subject, but for an author to succeed in making a complex subject digestible to the novice, wide experience and authority are required. FGW has these in spadefuls.

The book is a manageable size (19 x 23.5 cm) and has something less than 400 pages. As such it does not cover every aspect of C++ and the novice should be grateful for this – it makes it all do-able. One nark is that the index does not work, one has to add either 8 or 10 to the numbers in the index to arrive at the correct pages. Once one has got over the irritation of this, one should treat it as an exercise in pointer arithmetic and not complain. The slightly garish cover design looks as though the publishers are aiming at the secondary school market. If so, the kids will need considerable self-discipline to stay the course, or very persuasive supervision. The book demands a lot of commitment on the part of the reader and many hours at the computer doing practical programming and problem solving. Here I have to make a teeny confession. I stopped doing practical work after Chapter 4. It was hard to find the necessary spare time to do a really thorough job of working through this book (the same was true of YCDI), but having paid close attention to the text I felt that I could visualise the results of most of the program listings. I intend to use the book only as a tool for fixing problems with other programs (God forgive me) but I would not advise any real novice to follow this example.

Two large sections of the book concern the development of playing card entities and chess board entities. The former exercise could be accomplished by students with very little domain knowledge but I think the latter would

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 us know). 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

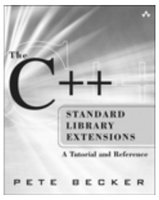
be quite difficult and even plain tedious for anyone who is not a chess player. Some study of the game would be a great help. However, both these props served excellently well for introducing the required programming functions. Presumably, professional programmers too have to submit to the tedium of dealing with domain knowledge in which they have absolutely no personal interest.

When you buy YCPCPP you also get a CD which includes an IDE called MinGW Developer Studio along with a C++ compiler from The Free Software Foundation (the same package is available on the internet). There is also an IDE called JGrasp which can be used with a variety of compilers. Each package is present in both Unix and Windows flavours. In addition, you get a large selection of supporting text from YCDI – an excellent idea. I'm glad of the opportunity to use MinGW Developer Studio because it serves as a stepping stone to Microsoft Visual C++ 2005 Express Edition (also free from the internet). I have that installed too but am taking it very cautiously at the moment because its fearfully complex (Microsoft describe it as fun and lightweight, however).

All in all You Can Program in C++ is very student friendly. Highly recommended.

The C++ Standard Library Extensions

by Pete Becker
Publisher: Addison Wesley (3 Aug 2006)
ISBN: 0321412990
Reviewed by: Francis Glassborow



You may know that ISO recently voted in favour of a Technical Report (TR) from the C++ Standards Committee (WG21) that substantially extends the C++ Standard Library. More recently WG21 incorporated almost all these extensions (the Special Maths Functions are currently an exception) into the Working Draft (colloquially called the Working Paper) of the next full release of the C++ Standard.

This book documents the material included by the TR as well as some additions and changes that have been provided by the 1999 release of the C Standard. The sum total almost doubles the size of the Library.

The back cover describes this book as the perfect companion to *The C++ Standard Library* by Nicolai Josuttis. I do not agree. The style and the depth of the tutorial aspect is markedly different (as you would expect from the difference of authorship).

Before going further, I should say that I did not like the layout and presentation of code in this volume, and regret that this was not more like that in Nico's book. The layout is much more like that used by P.J. Plauger in his excellent book on the Standard C Library (hardly surprising as Pete has worked closely with PJP for quite a number of years).

Whilst I am on the negative aspects, I should add that I find end of chapter exercises completely inappropriate for a book of this nature. Some of the exercises would be better (in my opinion) presented as worked examples and others add nothing useful and will be skipped by almost every reader. Perhaps the relegation of potential worked examples to exercises for the reader goes some way to explain why this book is relatively so much shorter than *The Standard C++ Library* (533 pages of main text versus 742 pages). The other part of the explanation is that Pete gives nothing more than bare bones documentation for the numeric and special maths functions. I think it would have been useful if he had included a commentary on the discovered (by him and PJP) problems with the implementations of the Special Maths Functions, several of which have very limited accuracy in certain circumstances.

By now you may think that I am against this book. You would be mistaken. Pete has tackled a difficult but very necessary task with a high degree of technical competence and I have no doubt that this book belongs on the reference shelf of every competent C++ Programmer.

There is nothing else current available that gets even close to providing the wealth of information you will find here. Many readers will find the extensive section on regular expressions to be just what they need to get to grips with this important aspect of C++ programming.

The chapter on random number generators surely must be compulsory reading for anyone who wishes to use a PRNG in their code. The chapter on smart pointers should be compulsory study for all post-novice C++ programmers.

Though this book is not the one I would have wished it is a tour de force covering some very difficult material. The extensions to the Standard C++ Library are exactly those that are needed as the fundamental building blocks for competent programmers. These kind of library resources are so much more useful than the plenitude of higher level libraries that exponents of other languages seem to expect.

If you are serious about your C++, go and buy this book today because it will be some time before anything better comes along (however for future readers of this review, note that that statement is made in August 2006 and might not remain true indefinitely).

Miscellaneous

Fit for Developing Software

by Rick Mugridge and Ward Cunningham
Publisher: Prentice Hall PTR (Oct 2004)
ISBN: 0321269349
Reviewed by: Anthony Williams

As the subtitle of this book says, Fit is the Framework for Integrated Tests, which was originally written by Ward. This is a testing framework that allows tests to be written in the form of Excel spreadsheets or HTML tables, which makes it easy for non-programmers to write tests. This book is divided into several parts. Parts 1 and 2 give an in-depth overview of how to use Fit effectively, and how it enables non-programmers to specify the tests, whereas parts 3-5 provide details that programmers will need for how to set up their code to be run from Fit.

Though I have been aware of Fit for a long time, I have never entirely grasped how to use it; reading this book gave me a strong urge to give it a go. It is very clear, with plenty of examples. I thought the sections on good/bad test structure, and how to restructure your tests to be clearer and easy to maintain were especially valuable – though they are obviously focused on Fit, many of the suggestions are applicable to testing through any framework.

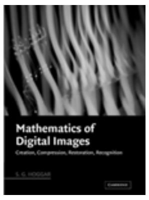
Fit was developed as a Java framework, and so all the programming examples are in Java. However, as stated in the appendix, there are ports for many languages including C#, Python and C++. The way of structuring the fixtures that link the Fit tests to the code under test varies with each language, but the overall principles still apply.

The book didn't quite succeed in convincing me to spend time working with Fit or Fitnesse to try and integrate it with any of my existing projects, but I still think it's worth a look, and will try and use it on my next greenfield project.

Recommended

Mathematics of Digital Images

by S. G. Hoggar
Publisher: Cambridge University Press (14 Sep 2006)
ISBN: 0521780292
Reviewed by: Francis Glassborow



The subtitle of this book from Cambridge University Press is 'Creation, Compression, Restoration, Recognition'. As the title indicates, the book is heavy on mathematics and those who struggled to achieve success at High School math (or GCSE in the UK) will likely be drowned by the requirements of this book.

The tough mathematics requirement is unavoidable, what could have been avoided is the academic writing style. I wish that academic authors would take on a co-author whose job would be to make the contents more accessible by, for example, breaking excessively long sentences into shorter ones. This author adopts the style of writing in the first person plural. That is certainly an improvement on those who insist on writing in the third person passive.

The back cover mentions 'pseudocode', do not be led into expecting that there will be much to help you transfer theory into practical code that you can use to help your understanding. You will have to provide the understanding and probably write the pseudocode for yourself as part of the journey towards enlightenment.

There are some surprising sections in the book (at least surprising to those not already familiar with the computational needs of computing with digital images).

90 pages on matrices will surprise no one, but more than 180 pages on probability is likely to be a major surprise.

This book succeeds in its objective and if you need to understand all the intricacies of handling digital images it is about as good as you can get n the theory side. For practical programmers I would like to have seen the material supplemented with far more pseudocode. Even better would have been real compilable code in some widely available programming language. My preference would be for C++ but Python, C, C# or even Java would be acceptable.

This is a good book if you can cope with the mathematics and have the time to study the contents properly. To some extent the individual sections stand alone, but you would need to read a section linearly and not try to dip in in search of something relevant to a current programming problem. This is a book to develop your overall knowledge, understanding and skills.

XPday

6th annual agile software conference run by Extreme Tuesday Club

Monday 27th and Tuesday 28th November 2006 in London, UK

We offer a wide variety of interactive and tutorial sessions covering every aspect of software development. The sessions we run are designed for a mix of project managers, developers, business analysts and testers – catering for all experience levels from beginners to advanced practitioners.

The event features:

- ❑ Invited speakers from UK and Europe
- ❑ Keynotes: Joshua Kerievsky, James Noble and Robert Biddle
- ❑ Tutorials about agile software development methods
- ❑ Workshops and practical hands-on sessions

The conference is always a sell-out, so register early to avoid disappointment!

Find out more at www.xpday.org

The event is organized on a not-for-profit basis by the Extreme Tuesday Club (XTC), which has been meeting every Tuesday at the Old Bank of England pub in Fleet Street since 1999.

View From The Chair

Jez Higgins
chair@accu.org

Before I became Chair, and was a normal committee member without any particular portfolio, I often used to get the impression that there was stuff going on in ACCU that I didn't know about. I didn't know what the stuff was or how it was happening, but thing were definitely going on. I occasionally raised it in committee meetings. "When there's this stuff going on", I would lament, "please can you tell us." It wasn't that anything bad was happening, simply that I felt it was important that the committee, and the wider membership, knew that things were happening. While I can't honestly say that traffic on the committee list rocketed as a result of my little campaign, one quite visible result is Alan's Secretary's report. Should his description of committee life pique your curosity, I'd remind you that members are quite welcome to attend committee meetings and see what goes on. Just mail Alan or me for details.

Now I am the Chair (or in the Chair, or whatever the correct term is), I find that I'm involved in this mysterious stuff and have discovered the reason people rarely wrote emails about it. Truth is that most of it is pretty boring, until it's actually done and finished. Fortunately, when they are finished they're usually pretty exciting. As Ewan announces elsewhere, the venue for next year's conference has now been finalised and it should be super. ACCU is also involved in the organisation of next year's WG21 meeting, a meeting that will finalise the shape of C++ 0x, not to mention bringing a bevy of C++ big-brains to Oxford at exactly the right time to pop along to the conference. By the time the column is printed, Tim, Tony and Allan should have further new chunks of accu.org open for business.

While these are driven by the committee, the real heart of ACCU is its members. Discussion on accu-general is, by turns, entertaining, technical, frivolous. Reg Charney's inspiring and address at the AGM is seeing its first fruit, as a regular meeting in London is being arranged even as I type this report. Overload and CVu have carried some terrific articles from both new and established authors. Participation in the current mentored developer projects remains strong, with recent Effective-C++ discussions revealing new insights to virtual function interface design. This is the real exciting stuff of ACCU. The best part is we can all join in and enjoy it.

Conference Report

Ewan Milne

Last issue Jez alluded to news about next year's conference location. I can now confirm that we will indeed be moving to a new venue. For the past three years the Randolph Hotel has provided a wonderful backdrop to our event. Bursting with character and with an undeniably great central location, it is with some regret that we have had to conclude that it is time to move from this home in order for the conference to grow and develop. Any of you who have wilted as the air-conditioning struggled to cope or suffered the elbow-to-elbow lunch-breaks will, I am sure, recognise that we face capacity issues. Note that we do not have plans to aggressively increase numbers at the conference, as we want to maintain the crucial community atmosphere of the event. But we have outgrown the Randolph.

We believe we have found the solution in the Paramount Oxford Hotel. This boasts a modern conference centre with ample spaces for our tracks, and additionally public spaces which will allow us significantly more breathing space. In fact we are currently planning more for how to use the extra space at our disposal rather than how to fit everything in. Also, I can report the air-con is efficient enough for me to have had to keep my jacket on when we met there.

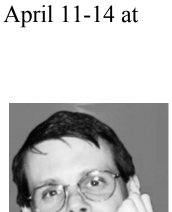
The one issue that must be addressed is location. The name is something of a give away, yes it is of course in Oxford. But rather than the central location we have recently enjoyed, it is on the outskirts. This is something that we are highly sensitive to, but we feel that the advantages offered by the facilities fully make up for this drawback. Not least of these is a large car park which should ease travel to the event by road considerably. And while we will not be right in the centre, we'll only be a quick bus ride away, and closer by are the restaurants and shops of Summertown.

The bottom line is that we feel it is time for a move, and we have found a new location which we believe will meet our needs. The committee are in fact very excited about the possibilities, and I hope you will be too when you come along to the 2007 conference: April 11-14 at the Paramount Oxford Hotel.

Website report

Allan Kelly

June 2006



Managing the ACCU website project is kind of different to managing a project at work. In some ways the ACCU project is more like an open source project, there are volunteers involved, people who give their time free of charge for... well, why do any of us give our time to the ACCU? Something to do with believing in the organization, its principles, the bond with other members and wanting to belong.

Even Tim Pushman at Gnomedia does this because he believes in the organization, yes we pay him but I know he does some work he doesn't charge for. Tim too does it because he wants to, the fact that we pay him means he can just devote more time to the project than Tony and I.

This mix does mean that things don't always happen in the predictable fashion they might do in a commercial organisation. Actually, scrub that, things don't happen like that in the office but we like to tell ourselves they do.

No, the ACCU website project is a true demonstration of the way projects unfold, half planned, half random, half done because we're paid, half done because we love it!

So, with that build up you're not going to be surprised when I tell you the things I set out in the last report aren't done. For example, I never mentioned the Wiki in the last report but we now have a Wiki to track everything that is happening, and a good job too, there is a lot, just not very fast!

At the conference a few of you were approached by myself or Tony about writing blogs for the website. Well, at the time of writing we have our blogging system up and the first blogger is about to start testing it. Hopefully by the time you read this the blogs will be public.

The mailing lists should transition to the new system within the next few months, the schedule is kind of vague, Tim keeps plugging away at it.

We've set the ball rolling in the US for a migration of their site. The US team will handle that themselves with support from us when and where they need it.

But the big news of for this report is the journals, or specifically, Overload. In the last report I briefly mentioned the options for putting journals online. Little did I know that the Overload editor, Alan Griffiths, was thinking along the same lines. He brought the issue to May committee meeting and it was decided to start putting Overload online. All of Overload, every issue, as soon as practical after the issue is published.

I think its a good balance, Overload is online and CVu isn't. If you're shy of your writing appearing on the web try CVu, things that are ACCU specific, like this report, go in CVu. This may change, for now it seems sensible.

Now you could argue that this will devalue your membership, *why pay when you can get it for free online?* Well I think there are some good reasons to continue paying. First, you still get a printed version, many of us (old fogies) prefer printed matter; second, your membership buys you not just Overload but CVu and a conference discount. Third, and to my mind most important, by joining the ACCU you are making common cause, you are saying, *I agree with these guys, their values, their beliefs and I want to belong to it.*

Even if we do lose a few members as a result of this move I think we'll more than make up for it in new members who discover the ACCU and our values through reading Overload online.

And then, even if thousands of members don't flood in I don't think we'll loose money. Because, the other thing we've just about go ready is website advertising in the form of banner ads. If you, or your company, would like to advertise on the ACCU website please drop me a line, we'll be going live in the next month or so.

I'm kind of conscious that once again I'm finishing this report by saying "Jam tomorrow"... thing is, I've turned into a project manager. Que twilight music.

September 2006

As every keen reader will undoubtedly have noticed, my report carried in the August CVu was in fact the same as the one in the April CVu. Such things happen even in the best publishing houses. Fortunately the Editor has kindly agreed to publish it in this issue so you should find two reports here not one.

Of course things have moved on from August and we now have our blog pages up and running on the website complete with RSS feeds. So far the main bloggers are Alan Lenton and our web site developer Tim Pushman, thanks to both of you!

Behind the scenes the journals are nearly ready for release, me and a few other lucky people have had an early preview and given some feedback so hopefully these will be available before I have to write the next report.

Another feature you won't have seen used yet is the banner ad system. We have installed a banner ad server on the site and its ready to go. Our problem is lack of adverts! About the time of the conference we thought we had several advertises interested and I think they probably still are. Although we have the technology we lack an advertising officer to coordinate the advertisers, billing, etc. Simply neither the website editor, Tony Barrett-Powell, nor myself have the time to organize this. Web advertising should be covered by the same person as journal advertising and I see great opportunities to sell print and online ads at the same time. If you feel like trying your hand at advertising officer please let Jez or me know and we'll talk some more.

The US Chapter haven't started their move to the website yet. We've made the site available; we're just waiting for them to move some of their content across.

Still, there is more work to do. The next major piece of work is the membership system. We need to get this up and running for the new membership secretary. Also, we hope to move the mailing lists to the new site before long.

On a personal note I'm hoping that I'll soon be able to step back from the website and reclaim some of my personal time. After two years I'm ready for a break.