the magazine of the accu

www.accu.org

{cvu}

Volume 26 • Issue 5 • November 2014 • £3

{cvu}

Volume 26 Issue 5
November 2014
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Silas S. Brown, Frances Buontempo, Pete Goodliffe, Ralph McArdell, Chris Oldwood, Roger Orr, Mark Radford, Giuseppe Vacanti, Emyr Williams

ACCU Chair
chair@accu.org

ACCU Secretary
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

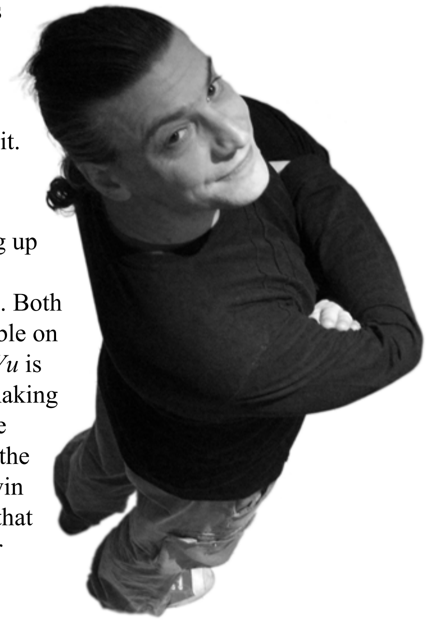
Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

Community service

There is change in the wind here at *C Vu*. It happens from time to time, because if it didn't it would be terribly boring!



C Vu has been the ACCU members' magazine since 1987, and in the intervening years it's changed quite a bit. The last big change saw *C Vu* take the format (roughly) you now see, in April 2006.

The next change being considered is the idea of opening up selected articles and items from the magazine to non-members as part of the online presence of the magazine. Both *C Vu* and *Overload* – our sister publication – are available on ACCU's website in PDF format, although currently *C Vu* is restricted to members only. However, it is hoped that making some of the articles we get for *C Vu* publically available might persuade more people to pay up, join up, and get the whole thing every two months – which obviously is a win for us as a magazine, and for them! The plan would be that selected items would be made available some time after publication, rather than immediately as is the case with *Overload*.

Your thoughts on this to the usual address will be much appreciated.

In the April 2006 edition I mentioned previously, the editor at the time, Paul Johnson, reported the demise of one of the last remaining print-copy magazines for programmers: CUJ. At the present time, I am aware only of *C Vu* and *Overload* being available *in paper copy* for programmers. There are many online magazines, blogs and other websites, but a printed magazine has the benefit that if you leave it lying around at work, someone who's never seen on may pick it up and flip through it, and might even like what they see.

I feel strongly about *C Vu* – and *Overload*, it's not all editorial bias! – that they perform a very important, even vital, role in publishing articles that are (and here's the really important bit) *peer reviewed*. The truth is that anyone can write a blog or respond to a question on one of the many tech-community websites, and whilst there is opportunity for people to make comments on those posts, it's not the same thing as having an article published that's been reviewed and critiqued by your peers. There aren't too many places left that make that claim, so help us to ensure that *C Vu* and *Overload* are able to continue this crucial service for all of you!

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

SUBMISSION DATES

C Vu 26.6:	1st December 2014
C Vu 27.1:	1st February 2015
Overload 125:	1st January 2015
Overload 126:	1st March 2015

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

Playing By The Rules

Pete Goodliffe makes up his own rules.

If I'd observed all the rules, I'd never have got anywhere.
~ Marilyn Monroe

We live our lives by many rules. This could be a dystopian Orwellian nightmare, but it's not. Some rules are imposed on us. But some we set ourselves. These rules oil the cogs of our lives.

Rules facilitate our play, describing how a game works: saying who has won and how. They make our sports fair and enjoyable, and provide plenty of opportunity for (mis)interpretation (see soccer's off-side rule).

They impinge on our travel, where security rules dictate you can only carry so much liquid, and no sharp objects, on airplanes. They describe traffic speed limits, and how to safely navigate a path on the road. Such rules ensure the safety of all.

Rules bound our social norms, stating that it's not appropriate to lick a stranger's ear when you first meet them, no matter how tasty it looks.

Yes, we live our lives continually observing a set of rules. We're so used to this that we often don't think about them.

Unsurprisingly, the same holds in our development work. There are a wide range of rules we follow at the codeface. Development process norms. Mandated toolchains and workflows. Office etiquette. Language syntax. Design patterns. These are the things that define what it is to be a professional programmer, and the way we play the development game with other people.

If you join a new project, there are various rules that you'd expect to be in place. Rules governing the responsible creation of high-quality code. Rules governing working processes and practices. And specific rules about the project and problem domain: perhaps legal regulations in force for financial trading, or safety guidelines for health markets.

These rules they help us work well together. They help orchestrate and harmonise our efforts.

We need more rules!

But sometimes all of these rules, good as they are, aren't enough. Sometimes the poor programmers need *more* rules. Really, we do.

We need rules that we've *made ourselves*. Rules that we can take ownership of. Rules that define the culture and working methods of development in our particular team. These needn't be large unwieldy draconian edicts. Just something simple you can give new team members so that they can immediately play the game with you. These are rules that describe something more than mere methods and processes; they are rules that describe a coding culture – how to be a good player in the team.

Programming teams have a set of rules. These rules define *what* we do and *how* we do it. But they also describe a *coding culture*.

Sound sane? Well, we think so. Our team's Tao of development is summed up in three short complementary statements. From these all other practices follow. These statements are now enshrined in our team folklore, have been printed out in large, friendly letters, and emblazon our communal work area. They reign over all we do; whenever we face a choice, a tricky decision, or a heated discussion, they help to guide us to the right answer.

Are you ready to receive our wisdom? Brace yourself. Our three earth-shattering rules for writing good code are:

- Keep it simple
- Use your brain
- Nothing is set in stone

That's it. Aren't they great?

We set these rules because we think they lead to better software, and have helped us become better programmers.

They perfectly describe the attitude, the sense of community, and the culture of our team. Our rules are purposefully short and pithy; we don't like lengthy bureaucratic dictats or unnecessary complication. They require developer responsibility to interpret and follow; we trust our team, and these rules empower the team. They are always new ways to apply them in our codebase; we are always learning and seeking to improve.

Set the rules

These rules make sense to us, in our project, in our company, and in our industry. They may not have the same import for you.

What rules are you currently working to? That is, apart from the ban on licking your colleagues' ears. Do you have a coding standard (either formal, or informal) in place? Do you have development process rules (perhaps the likes of: *Be in for 10 a.m. because we have a stand-up meeting. All code must be reviewed before check-in. All bug reports must have clear repro steps before being handed to a developer*)?

What rules govern your team culture? What informal, unwritten ways of collaborating, or approaches to the code, are particular to your team?

Consider formulating a small, simple set of rules that you can define your coding culture with. Can you distill it to something pithy like our three rules?

Don't rely on vague unwritten team 'rules'. Make the implicit rules explicit, and take control of your coding culture.

In the spirit of our third rule, don't forget that *nothing is set in stone* – including your rules. Rules are there to be broken, after all. Or rather, rules are there to be *remade*. Your rules may justifiably change over time as your team learns and grows. What is pertinent now may not be in the future. ■

Questions

1. List the software development process rules currently in place in your project. How well are these enforced and followed?
2. How does this project's culture differ from your previous projects? Is it a better or worse project to work in? Can the difference be captured or improved in a rule?
3. Do you think your team would rally around an agreed set of rules?
4. Does the shape, style, and quality of your code have any effect on a projects' coding culture? Does the team shape the code, or does the code shape the team?

Pete's new book – *Becoming a Better Programmer* – has just been released. Carefully inscribed on dead trees, and in arrangements of electrons, it's published by O'Reilly. Find out more from http://oreil.ly/1xVp8rw

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

Taming the Inbox

Chris Oldwood shares his tactics for keeping on top of the mail.

One of the things you get to deal with in 'The Enterprise' is how to manage the ton of email you get every day. For some reason corporate culture loves it. And I hold my hand up too as being one of those people who has probably overused it. Your inbox is bombarded every day by emails from various sources, such as the company's latest results or PR exercise, support notifications to keep you 'in the loop', location aware messages to find the owner of the dropped mobile phone, meeting invites, etc.

What I find confusing is the number of people who deal with this by burying their head in the sand, i.e. they just leave everything in their inbox and make no attempt to configure their email client to relieve (some) of the burden. When you do send them an email they seem surprised when you follow up in person and they say, "Sorry, I didn't see that message". In many cases these are developers who would never dream of sticking all their source files in a single source folder, and yet the same rules to manage complexity in software don't seem to apply to email.

In one sense I envy their attitude towards email – I wish I could be so blasé about the whole affair and just grumble about how much it interrupts my day and is just another distraction from writing code. But the thing is, it isn't and hasn't been for a long time as I've learnt to manage it, just as I've learnt to manage the complexity of the code I work on. What follows below are the practices I've developed to manage my email load. Of course one size never fits all, but you may either find some solace in my habits or confirm my position as a slave to an anachronism.

Message status

Given the environments I work in Microsoft Outlook is the email client of choice and it has one of the worst default settings I've ever come across: 'mark item as read when selection changes'. This means that every time you click (or cursor up/down) to another message in your inbox the previous one will be marked as read – irrespective of whether you actually read it or not. Given that your focus is on the target email, not the ones along your journey, you probably won't even notice the change in status of the ones you accidentally clicked on. If you are going to leave everything in your inbox, the read/unread status is probably the only way you're going to know what you have and have not attended to, so you owe it to yourself to switch this off. I don't leave emails in my inbox but I still turn it off as the 'read status' is one of my indicators for what I've yet to attend to.

Our esteemed *C Vu* editor informs me that this is also the default setting on a number of other email clients. If anyone can enlighten me to why this is the choice of the masses please send a postcard to the usual address, i.e. accu-general.

Filtering messages

My primary filter for what order to look at emails is not whether it's been read or not, but which folder it's in. After turning off the setting above I start to consider how I need to filter any incoming email so that stuff that is likely to demand my attention stays in the inbox (i.e. my highest priority mailbox) whilst the less exciting emails end up in some other folder. In either case it remains in an unread state as even the lowest priority of emails may have some redeeming content unless I know for sure I will have no false positives with a rule (e.g. I can match on an email address from an automated account).

As a general rule anything not addressed directly to me, or directly to my team's email distribution list(s) goes into the bin (the Deleted Items folder in Outlook). So much email comes through on so many email lists that aggregate various other lists *ad infinitum* that it would be a chore to continue maintaining the default rule by adding each new corporate spam list as it shows up. Hence I prefer a white-list to a black-list, where the white-list is me and my team mates.

That simple rule often suffices for many organisations, but it depends on how entrenched you are in the company, or whether you have a support role as well. Support is often done through an email list that I may or may not be interested in depending on the support rota. As such I may create a custom folder, perhaps under the Inbox, for mail addressed to the support account.

I have known an organisation get wise to this 'tactic' of filtering internal spam by using read receipts to monitor consumption. They then started addressing mail to everyone directly (or perhaps it was just me because they knew I was not only binning it, but marking it as read automatically too). For these occasions the other filtering rules can usually be effective, such as matching a text string like 'Monthly Update from the CEO'. Hopefully by now we've reduced the burden so much that the odd false negative is not too onerous to deal with manually.

When to read

I know many people find email a huge distraction and dislike being interrupted when in a state of 'flow'. Personally it's been so long since I've ever achieved a state of flow I'm not sure what it looks like. A large part of that is probably down to having four children – their constant need for attention means I've developed the ability to context switch at will. Whether I'm anywhere near as effective being this way is likely open to debate, but it does mean I can attend to trivial emails without really pausing for thought.

The little envelope that Outlook displays in the notification tray can either be a useful little reminder to catch up on your mail or a horrible nag depending on how distracting you find tray icons. Toast-style notifications at least answer the question that the little icon only teases at – who's it from and could it be worth reading now? However their effectiveness even for someone like me depends on them being the exception rather than the norm. In an email heavy environment all these indicators, along with playing a sound when a new message arrives, can seem like you're at a fireworks display.

One problem with being elevated from last-through-the-door to longest-serving-member means that you acquire an awful lot of knowledge and experience that others (unfortunately) start to rely on. Even if I'm not the only one who knows the answer, if I feel someone is blocked in their work and I have the ability to reply quickly and that would unblock them, then I feel the team wins. However this kind of helpfulness does nothing to improve your own productivity and can lead to awkward questions if your company measures value in lines of code.

As such my emails are dealt with in one of two ways. If the email is relatively short and can be answered within a couple of minutes I'll just do it there and then. Rather than handle a single email, naturally I'll handle all the quick ones in succession. Of course because I do it that way my inbox never accumulates and so it mostly stays empty. Any that are too

CHRIS OLDWOOD

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

long to read or reply to immediately stay marked as unread so that I know I still need to look at them properly later.

That generally leaves the longer emails addressed to me and any others that were filtered automatically into the lower priority folders. These I can tackle at a time when a natural break occurs, such as running the clean + build + tests after integration and before checking in, or after one story is finished and I'm ready to pick up the next, or before resuming a story after having coffee/lunch, etc. This kind of email tends to be pretty rare these days as it can often be converted to a formal project task if the answer would be better served as a wiki page, or by teeing up a face-to-face chat to explore a problem directly (high-bandwidth communication).

If I'm on support then the priorities change so that I scan the support folder first and will probably interrupt my flow to handle a potentially lengthy support email as that is usually far more important.

Finally that just leaves the stuff that goes straight in the bin, but still marked unread, which I'm pretty sure is just junk. Most of the time I won't even go past the subject line and will either block select and mark as read or use the folder option 'mark all items read' for ease. Very occasionally something interesting gets filtered by mistake and so I might dealt with it then or move it back to my inbox to peruse later.

You might have noticed I've not mentioned the message priority anywhere. That's because I completely ignore it. Putting the word 'URGENT' in the subject line in capital letters does not make me feel the need to respond any quicker either. The priority flag is usually just an indication of their work priorities, not mine.

Message archives

Back when disk space was measured in MB it was not possible to keep every email sent, especially when they contained attachments. These days it's entirely possible to never delete anything ever again. The problem now is searching a vast email archive for the answer to that question you've just been asked… once again.

Rather than leave all my mail in the Inbox or 'soft-deleting' by moving it to the Deleted Items folder I developed a habit of creating a top-level archive folder with a set of suitably named sub-folders used to group emails into more manageable chunks. For example on my current project, which I've only been in for a few weeks, I already have the following folders: Build, Design, Infrastructure, Releases, Security and Tools. In essence I group emails just like I group code into namespaces.

Once I joined the corporate circle I found this technique to be very valuable. It has proved to be a good defence mechanism to remind both team insiders and outsiders about a prior conversation when amnesia appears to have set in. One manager I worked for asked the same question time-and-again and so I used to dig out the last copy of the email I forwarded him and prefixed it once more with 'Here it is, again'. This kind of passive-aggressive behaviour changed nothing of their behaviour but at least I got to chuckle as the email grew longer and longer with each new occurrence.

Earlier I mentioned that I can often answer an email very quickly and this easily searchable archive is one reason why that becomes possible. Even with years of accumulated emails it becomes relatively easy to find stuff. Naturally the tool's search feature is a good start, and narrowing to just one folder speeds things up immensely. Sometimes though I can't remember a distinct enough term to search on and so a manual search is required. This might seem like the proverbial 'needle in a haystack' but with a good folder structure you'll be surprised how quickly you can hone in on something just by flicking down the subject lines.

So, what do I archive? I keep every non-trivial, team-relevant conversation that passes by my inbox. Although in some cases I have kept even the replies that just say 'thanks' as proof that the email must have been read by the recipient (in case they plead ignorance). That might seem like a lot of mail but once you have a good archive structure it only takes moments to read it and file it away. In the past I tried to keep only the most recent message in any conversation but I found that became more of a burden and if the thread diverged it became hard to know which to keep.

One other reason for not just leaving everything in the bin or the inbox (if you want a record kept somewhere) is that some companies have a policy of deleting mail older than, say, 3 months in either of these two folders. Ironically, rather than attempt to reduce its dependency on the tool they prefer to recycle the storage more quickly.

Occasionally one archive topic's folder might get too general and so I then split it or create subfolders. I leave the existing pile in place and then start afresh which creates a sort of multi-level cache when searching. However these days I've done it so many times that I generally adopt the same structure each time and have a feel for what sort of breakdown is likely to work.

My original goal for all this hoarding was to allow me to dump the entire contents of my mailbox into a .pst file on some file share when I left so that any knowledge acquired was still accessible to others after my departure. While I have done this on later contracts I didn't in the early days because I was fearful that an email with a sensitive conversation in it might leak out. Consequently my desire to be able to just dump my mailbox on exit has helped keep me a little more honest too.

Handling request timeouts

Most replies will either be part of an ongoing conversation, in which case the original email can be archived, or will involve me firing off a request to which I will eventually expect a reply. I like to manage my Sent Items folder in a manner similar to my inbox (it is a mirror image after all) so I archive sent emails too alongside the ones received.

I used to use a special archive folder called '_pending' (the leading underscore sorts the mailbox at the top) to act as a reminder of how far the conversation has gone. However I eventually realised that by emptying the Sent Items folder in the same way as my Inbox I could leave pending items in there instead.

Treat the disease

By now I suspect many of you are screaming at this article telling me to treat the disease, not the symptoms. And you would be right. In some cases the burden of team communication by email has lessened over the years as common uses have been replaced by the daily stand-up, pair programming, code reviews, IRC style chat [1], wikis, feature trackers, etc. Convincing your own team to make better use of face-to-face communication or other tools is usually within your own control.

The question is how to convince those outside your control that you don't want the deluge of spam directed at you. Where's the link at the bottom of the weekly timesheet reminder that allows me to unsubscribe? Perhaps corporate email software will one day evolve to a point where all email lists become opt-in, not impossible to opt-out of. Until that day comes I'll continue to opt-out where possible using automatic filtering rules and opt-in for stuff that actually has some value for me and my team. ■

References

[1] 'In The Toolbox – Team Chat', *C Vu* 25-2, August 2013

Const and Concurrency (Part 1)

Ralph McArdell comments on comments to Herb Sutter's updated GotW #6b solution.

I have been following Herb Sutter on his *Sutter's Mill* [1] website and while reading, in 2013, GotW #6b Solution: Const-Correctness, Part 2 [2] and the comments subsequently posted, some thoughts popped into my head.

For those in need of a reminder or enlightenment, GotW is the commonly used short-form term for Herb Sutter's 'Guru of the Week' C++ posers. The 'GotW #6b Solution: Const-Correctness, Part 2' article is Herb's posted solution to the GotW #6b poser which concerns consistent usage of `const` and `mutable` in C++ to write safer code. The posed task for which the article provides Herb's solution is to add or remove `const` to/from the shown `polygon` class which caches the result of area calculations, with bonus points awarded if you could point out undefined results or uncompilable code caused by the erroneous use of `const`. The solution includes concurrency and synchronisation concerns including whether to protect the cached, double, area member by a mutex or make it `std::atomic<double>` (which is where `mutable` makes an appearance).

The first comments to catch my eye were about the overhead that may be incurred by `std::atomic` being only to do with what liberties the compiler can take with respect to optimising writes. This raised an eyebrow as I was under the impression that the need to force atomic operation effects to be globally and consistently visible has more of an effect on performance than reordering write restrictions. Even where a processor has atomic memory update support there is still a cost [3, 4] – although that cost is significantly lower in modern processors.

However, the main focus of my thoughts has to do with various comments noting that modifying the set of points added to objects of the problem's `polygon` class is not synchronised and performing such changes concurrently with other operations without external synchronisation will end in tears and confusion! One suggestion was to do all the updates on a single thread then allow shared concurrent read access to the finalised polygon. This got me speculating as to how to enforce such a usage pattern, rather than just arrange for violations to such usage to not occur by convention and hope everyone plays along.

What follows is the result of me following up on these initial thoughts, along the lines of "let's see where this goes…", and as such is not intended to necessarily solve any real world problems or even to produce any workable solutions. It is assumed that the types involved, like the GotW 6B `polygon` class, do not have immutable instances so any thread could potentially modify an instance at any time. If nothing else maybe these musing will serve as an example as to why immutability can be a good thing.

So, the pattern is:

1. Create an object – e.g. a `polygon` as per GotW 6b; this occurs on a single thread.
2. On this thread perform updates to this object (e.g. add points to the polygon).
3. Having performed all updates share the polygon for read-only access from potentially many threads concurrently.

The above omits what to do when we wish to delete the object – which of course should be considered a mutating operation! The simplest option might be:

4. When all threads using the object have died the object may be deleted.

Although this seems somewhat restrictive.

During the initial updating phase read-access will, in general, also need to be restricted to access by a single thread. Step 2 could be relaxed to allow access on different threads so long as access only occurred on one thread at a time. It would be nice to relax step 4 to allow the object to be safely deleted without all the reader threads having to terminate first.

Thus mutating and non-mutating (or `const` in C++ terms) operations have different usage restrictions:

- Mutating operations are allowed to be used initially from a single thread then disallowed.
- Non-mutating operations are allowed to be used initially from a single thread then from multiple threads.

Let's consider restricting multithread access first. An obvious lock-based approach would be to create an object in a locked, mutable, state and then change the state to being immutable and release the lock. This does not help with the deletion problem though. Using a readers-writer or readers-upgrade-writer lock could potentially solve this issue: create in mutable, write-locked state, when done setting up the object's state move to the immutable readers-locking state, when wanting to destroy the object obtain a writer lock, possibly via an upgrade lock.

Such locking strategies allow for more flexible usage patterns than we require here. Because multithread access is forbidden when in the initial mutable state we can just treat such accesses as errors when in this state. It would be nice to be able to prevent such usage by failing compilation – possibly restricting usage patterns such as use as stack objects only – but (I am fairly certain) this is not attainable. Hence we should look to detecting and raising such misuse as errors at runtime.

One obvious approach would be to use a `std::atomic<bool>` flag instance member that is tested, set and reset around each mutating operation. An exception is thrown if testing the flag indicates the method is currently being used by some other thread (see Listing 1).

This will be monotonous to do repeatedly but the logic could be centralised – for example bundled up into a functor using execute-around to plumb in the boilerplate code around the actual work passed as a lambda function. A rough bare bones implementation might look like Listing 2.

Listing 1

```
void the_type::mutating_operation()
{
  bool expect_false{false};
  if (!in_use.compare_exchange_strong
     (expect_false, true))
  {
    throw std::runtime_error
      {"!!Concurrent access: illegal usage!!"};
  }
  // Do state changing stuff…
  in_use = false;
}
```

RALPH MCARDELL

Ralph McArdell has been programming for more than 30 years with around 20 spent as a freelance developer predominantly in C++. He does not ever want or expect to stop learning or improving his skills.

Listing 2

```
class enforced_exclusive_executor
{
  std::atomic<bool> in_use;

public:
  enforced_exclusive_executor() : in_use{false}
  {}

  template <class WkFnT>
  void operator()(WkFnT do_work)
  {
    bool expect_false{false};
    if (!in_use.compare_exchange_strong
       (expect_false, true))
    {
      throw std::runtime_error
      {"!!Concurrent access: illegal usage!!"};
    }
    do_work();
    in_use = false;
  }
};
```

Which reduces the mutating operation implementation to:

```
void the_type::mutating_operation()
{
  exclusive_exec([&]()
  {/*Do state changing stuff…*/;});
}
```

In which **exclusive_exec** is an instance member of **the_type** of type **enforced_exclusive_executor**.

Other than the various overheads incurred this technique's main problem is not necessarily detecting access by multiple threads immediately, if at all. This means that pattern-misuse exceptions are raised indeterminably. Then of course there is the question of what to do about non-mutating operations that do not alter the object's state.

A better approach may be to work with a token: if the token matches the current valid token then updates are OK otherwise it is erroneous usage. A convenient token would seem to be a thread id, represented in C++11 by the **std::thread::id** type (see Listing 3).

Listing 3

```
void the_type::mutating_operation()
{
  if (std::this_thread::get_id()!=update_id)
  {
    throw std::runtime_error
     {"!!Concurrent access: illegal usage!!"}; }
     // Do state changing stuff…
}
```

Listing 4

```
void the_type::validate_call_context()
{
  if (std::this_thread::get_id()!=update_id)
  {
    throw std::runtime_error
    {"!!Concurrent access: illegal usage!!"};
  }
}

void the_type::mutating_operation()
{
  validate_call_context();
  // Do state changing stuff…
}
```

update_id is an instance member of **the_type** of type **std::thread::id** which is initialised to the thread id of the thread creating the object. Of course the check logic can be pulled out – for example into a private instance method maybe called something like **validate_call_context()** (see Listing 4).

This scheme will throw an exception any time any thread other than that the object expected to be called on calls any instance member function that validates its call context – which should be most if not all operations. The scheme has potential to be extended. Transferring the call context to another thread is simply a matter of updating the value of the object's **update_id** to the id of the new calling thread. As such a transfer only makes sense during the initial updating stage of the object, like all such operations during this stage, it would be restricted to being performed only by the current calling context. As a bonus when moving to the shared, immutable state the calling context can be 'transferred' to the single distinct **std::thread::id** value that does not represent a thread – as produced by default constructing a **std::thread::id** object – which would ensure that all operations that validate their call context will fail.

A concern is that as **std::this_thread_get_id()** is called for each validation it should be a cheap – preferably very cheap – operation which may not be the case for all implementations.

I shall pause here and defer developing my musings further for a later article. ■

References

[1] http://herbsutter.com/
[2] http://herbsutter.com/2013/05/28/gotw-6b-solution-const-correctness-part-2/
[3] 'Is Parallel Programming Hard, And, If So, What Can You Do About It?' v2013.01.13a especially sections 2.1.3, 2.21, 2.2.3 https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
[4] What Every Programmer Should Know About Memory, section 6.4.2 http://www.akkadia.org/drepper/cpumemory.pdf

Parsing Configuration Files in C++ with Boost

Giuseppe Vacanti describes how to deal with program options, C++ style.

In the (Unix) world of command-line tools I inhabit, parsing configuration files is a common first step most of my tools perform before starting a long and complex computation. Although command-line tools may make you think of command-line switches, I often write simulation or data analysis tools that require up to a few tens of parameters to describe either an experiment or a complex geometrical model: not something you want to type out often on the command line. Storing the parameters in a file, possibly allowing the user to perform a command line override of some of them, is one possible solution.

After thinking about the format of these configuration files, I chose to use the INI format. The latter is loosely defined, and a number of variations on the theme exist, some of them rather complex, involving nested sections and entries extending over multiple lines. I have settled for the classical format consisting of a series of **key=value** lines, possibly grouped into sections, like the following example:

```
energy = 10
mass = 22
[detector]
position_x = 1
position_y = 2
position_z = 3
[detector.material]
atomic_number = 43
```

I am aware that the format has some limitations, especially when it comes to string values that either extend over multiple lines or contain escaped characters, but I do not care for these features.

There are a number of libraries able to read INI files, but most of them require the client to perform all the work of checking whether parameters are present (or misspelled), and to what type they should be converted. For instance:

```
ini_file i_f(file_name);
if(i_f.has_key(section_name, key_name)){
  cout << i_f.key<double>(section_name,
     key_name) << endl;
}
```

Why this is so can be understood by building a minimal INI file parser in C++. In this parser the INI file could be represented by a **map<string, map<string, string> >**, and with a bit of encapsulation one would arrive to the interface in Listing 1.

With such a parser the number of checks and conversions the client must perform in order to extract all the required parameters is large, and prone to error.

Enter the Boost library Program Options [1]. The main purpose of the library is to parse command line options, but it offers the possibility to read the options from an INI file, which is exactly what I want. Additionally, the library will check whether a key has a value that can be converted to the appropriate type, it can deal with duplicate/multiple entries (allowed or not allowed), can be used to provide default values for missing keys, and more. In the following I explain how to use the library for this purpose.

Listing 1

```
class ini_file {
  public:
  ini_file(istream & is) { ... }
  bool has_key(const string & section,
               const string & name) const {
    Iter p = m_ini.find(section);
    bool ret_val = false;
    if(p != m_ini.end()){
      section_t & sec = m_ini[section];
      ret_val = sec.find(name) != sec.end();
    }
    return ret_val;
  }

  template<typename T> T key(const string &
      section, const string & name) const {
    assert(has_key(section, name));
    return boost::lexical_cast<T>
      (m_ini[section][name]);
  }

  private:
  typedef map<string, string> section_t;
  typedef map<string, section_t> ini_t;
  typedef ini_t::const_iterator Iter;
  mutable ini_t m_ini;
};
```

Let's start by going back to my sample INI file, and let us map it to a configuration data structure like the one in Listing 2.

Listing 3 is the piece of code that reads the parameters from the configuration file and assigns them to the appropriate variable in the data structure.

The library lives inside the namespace **boost::program_options**, here shortened to **po**.

We must first define a variable of type **po::options_description**, and add options to it. We do this by associating a name to a type and the corresponding variable in the configuration structure. Note that keys in a section have their name prefixed with the section name.

Having filled in the options description we can parse the configuration file (in this case passed in through the standard input, but any istream will do). As you can see, we need three lines of code to do this. The reason is that

Listing 2

```
struct configuration {
  unsigned int version;
  double energy;
  double mass;
  struct detector {
    double position_x;
    double position_y;
    double position_z;
    struct material {
    double atomic_number;
    } material;
  } detector;
};
```

GIUSEPPE VACANTI

Giuseppe Vacanti is a physicist who works at a small high-tech company in the Netherlands, and who likes to solve problems with C++ and Python. He can be reached at giuseppe@vacanti.org

Listing 3

```
configuration cfg;

namespace po=boost::program_options;
po::options_description desc;
desc.add_options()
  ("version", po::value<unsigned int>(&cfg.version))
  ("energy", po::value<double>(&cfg.energy))
  ("mass", po::value<double>(&cfg.mass))
  ("detector.position_x", po::value<double>(&cfg.detector.position_x))
  ("detector.position_y", po::value<double>(&cfg.detector.position_y))
  ("detector.position_z", po::value<double>(&cfg.detector.position_z))
  ("detector.material.atomic_number",
   po::value<double>(&cfg.detector.material.atomic_number))
  ;
const bool allow_unregistered = false;
po::variables_map vm;
po::store(po::parse_config_file(std::cin, desc, allow_unregistered), vm);
po::notify(vm);
```

the library allows one to merge options from multiple sources (for instance, default values from a configuration file that can be overridden on the command line), so that multiple calls to **`po::store`** may have to be executed before calling **`po::notify`**, that actually makes the parsing happen.

I mentioned earlier that the library has a number of features that simplify the life of the programmer.

Type checking

A first feature worth mentioning is that if the key value cannot be converted to the type specified, an exception will be thrown. For instance, if the configuration file contained the key

```
version = one
```

the program would be terminated with something like Listing 4.

Key name checks

What happens if the configuration file contains an extra key, or a key whose name is misspelled? This behaviour can be configured, but in most cases you'll want the program to tell you by disallowing keys whose name has not been registered, as shown in the example. In this case, if we had

```
versionx = 1
```

another exception would be thrown, with the message

```
what():  unrecognised option 'versionx'
```

Default values and required keys

What if one of the keys is missing from the configuration file? For instance, comment the version key out:

```
#version = 1
```

Without any measure from our part, the variable version will be uninitialized, and it will be assigned some random bit pattern:

```
version=1710991640
energy=10
mass=22
```

etc

We have now two options. The first one is to give certain keys a default value, by writing for instance:

```
("version", po::value<unsigned int>
  (&cfg.version)->default_value(99))
```

The second is to make version mandatory

```
("version", po::value<unsigned int>
  (&cfg.version)->required())
```

which leads to an exception when version is absent:

```
what(): the option 'version' is required but
missing
```

Multiple key instances

By default each key can appear only once in a configuration file, or an error is generated:

```
what():  option 'version' cannot be specified
more than once
```

But for some keys it may make sense to have multiple values, for instance

```
energy = 10
energy = 20
```

We make this possible by changing the declaration of energy to

```
std::vector<double> energy;
```

and the option description to

```
("energy",
po::value<std::vector<double>>(&cfg.energy))
```

Multiple configuration sources

There can be multiple sources of configuration data, and these can be processed one after the other to obtain the final configuration. In my case the user may want to have a standard configuration file, and then under some circumstances modify one or more parameters without modifying the standard configuration file. This can be achieved by adding a second call to **`po::store`**, before the first one (the value of a key is set by the first parser that encounters the key name):

```
po::store(po::parse_command_line(argc, argv,
          desc), vm);
```

So now the usage is as shown in Listing 5.

Listing 4

```
> ./main < example1.cfg
terminate called after throwing an instance of
'boost::exception_detail::clone_impl<boost::exception_detail::error_info_injector<boost::program_option
s::invalid_option_value> >'
  what():  the argument ('one') for option 'version' is invalid
Aborted (core dumped)
```

Perl is a Better Sed, and Python 2 is Good

Silas S. Brown sweats the differences between tools on common platforms.

If you've done any Unix shell scripting, you've probably come across the Stream Editor (sed). It's most often used for simple substitution, for example:

```
for N in *.wav ; do lame "$N" -o "$(echo "$N"|sed
-e 's/wav$/mp3/')"; done
```

which goes through all `*.wav` files and calls the MP3 encoder 'lame' on each one, passing a `-o` parameter as the filename with the `wav` at the end changed to `mp3` – it's the `sed -e s/x/y/` that does this substitution. [The `-e` argument allows you to provide multiple commands for a single invocation. Ed]

In this example, the `$` at the end of `wav` is there so that the substitution is made only at the very end of the filename; I don't want to confuse things if a filename happens to contain 'wav' part-way through. In other situations you might want to add a `g` after the closing `/` to globally replace a regular expression many times in a line.

As this example shows, however, you do have to think carefully about your regular expressions (regexps), especially if you don't know what input you're going to get. In the above example, if I knew in advance exactly

SILAS S. BROWN

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

Parsing Configuration Files in C++ with Boost (continued)

Listing 5

```
>./main --mass=99 --energy=123 < example1.cfg

version=1
description=this is the description
energy=123
mass=99
detector.position_x=1
detector.position_y=2
detector.position_z=3
detector.material.atomic_number = 46
```

Properties file format

You will have certainly noticed that in the options description code, keys in a section are specified as `section.key`, à la properties file. And in fact, the library can also ingest a properties file.

Custom types

The library support custom types, as long as they can be handled by Boost Lexical Cast. For a type to be handled by Lexical Cast it must be OutputStreamable, InputStreamable, CopyConstructible, and DefaultConstructible. If your type can be constructed from a string of tokens without any white space character in it, then there is nothing special you have to know.

On the other had, if you have a type like

```
struct special_type {
  double x, y;
  special_type(double x_, double y_) : x(x_),
     y(y_) {}
  special_type() : x(0), y(0) {}
};
```

things are not obvious (I had to dig into the Boost archives [2] to figure this one out), and the input stream operator for your type must be written as illustrated below because Lexical Cast does not ignore white space. See Listing 6.

Now the following works

```
special_type st =
  boost::lexical_cast<special_type>("12 13");
```

Listing 6

```
std::istream & operator>>(std::istream & is,
  special_type & val)
{
  is >> val.x;
  if((is.flags() & std::ios_base::skipws) == 0)
  {
    char whitespace;
    is >> whitespace;
  };
  is >> val.y;
  return is;
}
```

and as a consequence Program Options can handle something like

```
special = 99 101.1
```

and on the command line you can say `--special="100 200"`.

Options style

Being primarily intended for the command line the Program Options library can be configured to handle command line options in various manners (one or two dashes, case sensitive or not etc.). Most of the default style options are not going to cause any surprise, but you want to be aware of the fact that by default Program Options will accept a shorter spelling of an option (or key in a configuration file) if it unambiguously identify the complete option. This is possibly not what you want, in which case you will have to alter this behaviour. Note that this only applies to the command line parser, because the configuration file parser is very strict, case sensitive, and does not allow shortening of the keys.

By combining INI/properties file parsing and command line options parsing in one interface, the Boost Program Options library allows one to easily layer multiple input sources and feed data of moderate complexity into a program. While the file format it supports is not as rich as others, the library has additional functionality that makes it worth considering. ■

References

[1] Available at http://www.boost.org/
[2] Start here for the complete story: http://stackoverflow.com/questions/10382884/c-using-classes-with-boostlexical-cast

which filenames the command will be working with – say, a particular set of a dozen or so `.wav` files – and I knew that none of them contain the letters ‘wav’ except at the very end of the filename, then I wouldn’t need to worry about including the **$** character in the regexp. (Also, if I knew there were no spaces or other special characters in the filenames, then I wouldn’t have to put quite so many quote marks around everything.) But if, instead of writing a one-liner to do something with a particular set of filenames, I’m writing a script that I’ll be using later, or even sharing with other people, then I must be more careful.

Sed is a fairly universal tool: it’s installed ‘out of the box’ on nearly every version of Linux, even many small ‘embedded’ versions, and also on other Unix systems, such as BSD and its derivative Darwin which runs Mac OS X. So if you use sed for small jobs like this, it should work on all of these systems. At least, that’s the theory.

In practice, there are a few annoying differences between BSD’s version of sed (on the Mac) and GNU’s version of sed (on Linux). If you develop and test a script on Linux, it might not work on the Mac, and vice versa. For example, on Linux you can include **\n** in the replacement string to indicate an extra newline should be added, but you can’t do that on the Mac’s version of sed.

I often need to run my scripts on other people’s computers (or give them to others to run), so I want to make a reasonable attempt to minimise the amount of system setup that’s needed

Yes you can install GNU tools on the Mac, but I like my scripts to be able to run ‘out of the box’ to the extent possible, without requiring the installation of too much extra software. That’s because I often need to run my scripts on other people’s computers (or give them to others to run), so I want to make a reasonable attempt to minimise the amount of system setup that’s needed before the script will run. (That’s also why I tend to be parsimonious about how many third-party libraries my programs rely on: if such libraries won’t already be there on the system, and aren’t very easy to bundle, then they’d better be good enough to be worth the hassle of an extra dependency. A large library I want to make extensive use of, like the Tornado web framework in Python, might be a justifiable dependency, but I wouldn’t want to bring in an extra dependency just to save myself from writing a 10-line function – not unless I know for a fact that I’ll never have to set up this program with its dependencies anywhere else. The trouble with dependencies is you never know when someone will come along with a system on which they don’t compile, or doesn’t give them enough rights to run the installer, or something, and if it’s not your code then it’s that much harder to figure out what to do about it.)

And so we come to perl. I’m not an expert perl programmer (most of the perl I’ve done has been making changes to other people’s scripts rather than writing my own), but perl does have a very nice (and often overlooked) command-line option to sort-of ‘emulate’ sed: the **-p** option. Try:

```
perl -p -e 's/wav$/mp3/'
```

and you’ll find it behaves just the same as **sed -e**, except it’s the same across Linux and BSD (and supports things like ‘newline in replacement text’ on both platforms). Also, you don’t have to put backslashes in front of any parentheses you use (in fact you shouldn’t), which makes your regexps more readable. The other thing to watch for is, if you’re doing multiple substitutions then you should separate them with semicolons rather than supplying additional **-e** commands as with sed.

Apart from these minor differences to be aware of (which generally go in perl’s favour), **perl -p** is more or less a ‘drop-in replacement’ for most uses of sed, except it’s more powerful (and you don’t have to backslash-escape so much) and it’s more likely to work across platforms. So if you find yourself using **sed -e** in scripts a lot, I’d recommend being aware of this.

Of course, there will be some ‘embedded’ systems out there that have sed but not perl. But generally speaking, perl is quite ubiquitous these days, and it has for some years ‘settled down’ to a nice stable language that’s not likely to change under your feet, so it is very well suited for use in shell scripts like this.

What I call a ‘stable’ language, some people might call ‘stagnated’. But I don’t see what’s wrong with a bit of stability: if you want your code to be portable to many systems ‘out there’ with minimum fuss, it’s probably easiest if you’re using a language that has ‘settled down’ to being pretty much the same everywhere, even if this does mean you’re ‘living in the past’ to an extent.

Python 2 is now a nice stable language as well, especially since Python 3 has syphoned off all new development but Python 2 is still (just about) supported for essential bug fixes and security checks. Python 2 is pre-installed on nearly every Linux and Mac OS X machine, is available for all kinds of older systems that Python 3 has yet to be back-ported to – Windows Mobile, Android SL4A, Series 60, EPOC, even RISC OS – and there’s also a tool to turn a Python program into a standalone Windows executable, including interpreter, which can be run without needing any administrator privileges on the Windows machine (later versions of this tool began to require administrator privileges, which rules out use in a computer lab; I have a nice early version which even lets me update the Windows package from the comfort of Linux without having to go into Windows at all, athough it does mean I can’t add new libraries to it).

It’s even possible to write code in such a way that it will run on very old 2.x versions of Python, on older systems. For example, for Python 2.2 and earlier, do this:

```
try: True
except: exec("True = 1 ; False = 0")
```

which defines **True** and **False** as variables if the keywords don’t yet exist. And try to avoid writing ‘string1 in string2’ where **string1** can be more than one character (not supported in versions of Python before 2.3). You could also do:

```
try: set
except:
  def set(l):
    d = {}
    for i in l: d[i]=True
    return d
```

to emulate the **set()** constructor (from a list) on versions of Python before real sets were introduced.

But these days I usually target Python 2.7 if there is no great need to be *that* multi-platform (i.e. the script I’m writing will probably not be useful on Series 60 etc, but I still want it to work on any Linux or Mac system from the last few years). Even still, I try to code in such a way that it won’t be *that* much of a hassle to back-port to earlier versions of Python 2 if necessary (although if I have to depend on a library like Tornado then there’s no point even trying to support versions of Python that are older than the library supports – or at least there’s no point going before the oldest version of Python that’s supported by the oldest sensible version of the library).

I do remember writing for Python 1.x, and I’m glad I’m not doing that any more. But it now seems Python 2 has reached a nice balance of features and stability, and I really don’t see the need to move to Python 3: its advantages are not worth the extra dependency of installing it on every system I want my programs to work on (including older Mac OS X machines). Perhaps a Python 3 enthusiast would like to point out what’s so good about Python 3? But it had better be amazingly outstanding if I have to insist all my users install it first instead of using what’s already on their systems.

Debuggers Are Still For Wimps

Frances Buontempo shows how to remote debug python from Visual Studio.

Suppose you have a script you want to run on Linux and you only know how to drive the Visual Studio debugger. By installing an add-in for Visual Studio locally, installing the python tools for Visual Studio debugging on the remote machine, e.g. with **`pip install ptvsd==2.0.0pr1`** and adding a (minimum of) a couple of lines to your script you can debug in Visual Studio even if the remote machine is running Linux. The additional lines are highlighted in the script in Listing 1.

Listing 1

```
#!/usr/bin/python

"""
You will need to insert both these in your script
The remote box requires the ptvsd package
(otherwise the import fails)
"""
import ptvsd
ptvsd.enable_attach(secret = 'joshua')
#use None instead of joshua but that is not
secure
#The secret can be any string - but this is not
#properly secure

def say_it(it):
  """
  This inserts a breakpoint
  but you can add new breakpoints in Visual
Studio
  if required too/instead
  """
  ptvsd.break_into_debugger()
  print(it)

if __name__ == "__main__":
  #pause this script til we attach to it
  ptvsd.wait_for_attach()
  say_it("Hello world")
```

Be wary of line endings in VS, which may be inappropriate for Linux. More details are available online. [1]

You'll also need to install the ptvs from the relevant msi for your version of Visual Studio. Then start the script on the Linux box:

```
$python VSPyNoodle.py
```

It will hang, since it has a **`wait_for_attach`** call in **`main`**. ctrl-Z will stop it on the remote box if something goes wrong.

Select 'Attach to process' in the Debug menu on Visual Studio, and change the 'Transport' to 'Python remote debugging (unsecured)'. Add the secret (joshua in this script) @ hostname to Qualifier, for example:

```
joshua@hostname
```

Hit 'Refresh'. It should find the process running on the Linux box and add the port it uses to the Qualifier. Select your process in the list box and hit 'Attach' then debug as you are used to in VS.

If it complains about stack frames and not being able to see the code you may need to make a VS project from a local version of the code. having made sure it exactly matches the remote code. ■

Reference

[1] See https://pytools.codeplex.com/wikipage?title=Remote%20Debugging%20for%20Windows%2C%20Linux%20and%20OS%20X

FRANCES BUONTEMPO
Frances has a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been programming professionally for over 12 years. She can be contacted at frances.buontempo@gmail.com.

Perl is a Better Sed, and Python 2 is Good (continued)

Incidentally, this year the Ubuntu distribution of Linux declared an intention to eventually ship only Python 3 by default, and to make Python 2 an optional package. This has not yet come to fruition, but if it does, it still won't help non-Ubuntu distributions, or BSD, especially all the older Mac OS X machines that for various reasons might not be upgradable to whichever future version of Mac OS X actually ships Python 3 by default (as far as I know none of the existing versions of Mac OS X do this). In the current climate, if Ubuntu were to ship Python 3 by default then I'd just tell Ubuntu users to install the Python 2 package, because I'm concerned about all those other systems as well, some of which don't have easy-to-use package managers like Ubuntu does. But I don't understand why anyone would want to 'kill off' Python 2 anyway: why can't they leave it alone like Perl 5 as a super-stable ubiquitous tool? Yes I'm all for playing with new languages, but not when I'm trying to write something that's supposed to run everywhere (well, not unless I can first compile my code into a more widespread language to ship, but that's not the case with Python – if you want it to run somewhere then you need a suitable version of Python 'on site' there, and that usually means Python 2). ■

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

This article is only available in the printed version of *C Vu* due to rights issues.

C Vu magazine is available from www.accu.org

Not quite the reaction you were expecting to the latest release?

Clearly stated...

It may not be the software. How clear are the release notes? What about the product manual, online help, training materials, ...?

Changes may meet a business need, but if what worked yesterday doesn't work today, people may resent them. And when today's way involves extra steps, people work around them. After all, their priority is getting the job done.

Result: those shiny new features remain unused, and your application appears not live up to its promise.

If you would like some help in turning nervous cats into contented ones, get in touch.

T 0115 8492271

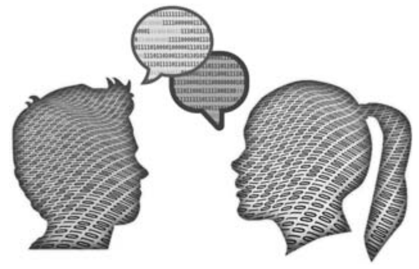
E info@clearly-stated.co.uk

W www.clearly-stated.co.uk

Our employees are members of the Institute of Scientific and Technical Communicators, the UK professional body for technical authors and related professions. For more information about the ISTC, visit www.istc.org.uk

Code Critique Competition 90

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



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

Note: we are investigating putting code critique articles online: if you would rather not have your critique visible please inform me. (We will remove email addresses!)

Last issue's code

> I'm trying to write a simple program to shuffle a deck of cards, but it crashes. What have I done wrong?

The code is in Listing 1.

Listing 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum
{
  black,
  red
};

enum
{
  Hearts,
  Diamonds
};

enum
{
  Clubs,
  Spades
};

typedef struct Card
{
  int color;
  int suit;
  int value;
} Card;

typedef Card Deck[52];

void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 51; i++)
  {
    myDeck[i]->color = i % 2;
    myDeck[i]->suit = i % 4;
    myDeck[i]->value = i % 13;
  }
}
```

Listing 1 (cont'd)

```
// -- example11.cpp -- (C++11 example)
#include <cassert>
#include "wrapped_vector.hxx"

int main()
{
  wrapped_vector<int> iVec;
  iVec.push_back(1);
  iVec.push_back(0);
  iVec.push_back(2);

  int total(0);
  for (auto & p : iVec)
  {
     total += p;
  }
  assert(total == 3);

  wrapped_vector<int>::dump();
}

void PrintDeck(Deck * myDeck)
{
  int i = 0;
  for(;i < 52; i++)
  {
    char *colors[] = {"black", "red"};
    char *suits[][2] =
       {{"clubs", "spades"},
        {"hearts", "diamonds"}};
    printf("Card %s %d of %s\n",
      colors[myDeck[i]->color],
      myDeck[i]->value,
      suits[myDeck[i]->color]
           [myDeck[i]->suit]);
  }
}

void Shuffle(Deck * myDeck)
{
  int i = 0;
  for (; i < 52; i++)
  {
    int n = sizeof(Card);
    int to = rand() % 52;
    Card tmp;
    memcpy(&tmp, myDeck[i], n);
    memcpy(myDeck[i], myDeck[to], n);
    memcpy(myDeck[to], &tmp, n);
  }
}
```

ROGER ORR

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

Listing 1 (cont'd)

```
int main()
{
  Deck myDeck;
  memset(&myDeck,0,sizeof(Deck));
  LoadDeck(&myDeck);
  PrintDeck(&myDeck);
  Shuffle(&myDeck);
  PrintDeck(&myDeck);
  return 0;
}
```

Critiques

Paul Floyd <paulf@free.fr>

Well, there's quite a lot not to like in this example. On first reading, I didn't like the **enum**s. There is redundancy – why two **enum**s for the suits and why a separate **enum** for the colour? The colour of a suit is something that is 'well known'. Also I didn't like the inconsistent capitalization. In any case these enums aren't used. I would remove the **color** field of **Card**, and infer the colour from the suit (see below for the corresponding change to **PrintDeck**).

The main issue is that there is confusion as to what a **Card** and what a **Deck** is. Specifically the **LoadDeck**, **PrintDeck** and **ShuffleDeck** take pointer to **Deck** arguments and treat them as arrays. They should be treated as pointers to arrays of **Card**s, not pointers to arrays of **Deck**s. Simply removing the asterisks in these prototypes (and changing **->** pointer dereferences to **.** member accesses) will fix this extra level of indirection.

Next, **LoadDeck**. This only iterates over 51 cards. I would recommend using a constant like **CARDS_IN_DECK = 52**. I don't like the single loop, which works because 4 and 13 are relatively prime. If this code were used for a game like 'belote', which uses 32 cards, then **LoadDeck** would no longer work correctly. Two loops for suit and value would be a little more verbose but much clearer.

PrintDeck just prints values from 0 to 12. Again much clearer would be names and values offset by 1, e.g., using

```
char *values[] = {"ace", "2", "3", "4", "5",
"6", "7", "8", "9", "10",
"jack", "queen", "king"};
```

The attempt to read the suit from the **suits** 2D array is wrong. This is a 2x2 array, but the ranges of the subscripts are 0..1 and 0..3. I would make **suits** a 1D array and index it with the 0..3 suit field. Also I would infer the colour from the suit, either directly in the code or by writing a little function.

I can only see one fault in Shuffle. There is no test to see if **i** equals **to**, in which case the second **memcpy** is undefined. This could be corrected by using **memmove** or adding a little check like

```
if (i == to)
{
  continue;
}
```

One last point. It bothers me using common names like **red** and **black**. To avoid conflicts, I'd use some prefix like **FooRed** and **FooBlack**.

Silas S.Brown <ssb22@cam.ac.uk>

I'm writing this reply on a Psion Revo PDA (made in 1999) while visiting my parents in rural West Dorset (with no Internet connection and not even a phone signal), and as I don't have a proper keyboard I'll be brief. (The reason why I say this at all is it might inspire other members, somehow or other. I do wish someone would come up with a decent modern version of the Revo though.)

Bug 1: there's an inconsistency between **LoadDeck**, which says **i<51** in the **for** loop, and **PrintDeck**, which says **i<52**. Maybe the writer was thinking **<=** instead of **<** when writing **LoadDeck**? Anyway it's better to keep it consistent and write **<52** in both cases (I wouldn't advocate using a constant here, because anyone who knows about decks of cards in Western culture will be familiar with the number 52, but that's an opinion that others are entitled to disagree with).

Bug 2: **suit** is set to **i % 4** but there are only 2 suits of each colour. (Personally I'd have dropped colour and just listed the 4 suits, perhaps with a **getColour** function that effectively tests bit 2, but I won't argue if you want to represent colour explicitly.)

Bug 3: the modulus operations in **LoadDeck** will not do what I think you meant. For example, all cards with odd value will also have colour set to red, whereas all cards with even value will also have colour set to black. Rather than trying to correct this using more complex assignments, I'd suggest simply writing three nested loops thus:

```
int pointer=0, color=0, suit=0, value=0;
for (; color < 2; color++) {
  for (; suit < 2; suit++) {
    for (; value < 13; value++) {
      myDeck[pointer]->color=color;
      myDeck[pointer]->suit=suit;
      myDeck[pointer]->value=value;
      pointer++;
    }
  }
}
assert(pointer==52);
```

Note the final **assert** as a sanity check. (You could also add items like **MaxSuits** and **MaxValues** to the ends of each **enum**, but it's probably not worth doing so in this small example.) Registers are cheap these days, so it's really no big deal to write multiple loops in this way, and it's more readable than the corrected version of the one-loop approach which I won't confuse you with.

The **Shuffle** function is OK (there are arguments about better approaches but let's not worry about that for now); you might want to think about seeding the random number generator (if I had a copy of the standard with me right now, I'd look up whether or not it's done for you by default).

James Holland <James.Holland@babcockinternational.com>

Having had a quick look at the code and given the fact that it crashes, I get the impression that the problem is all to do with pointers and structures. In fact it is reminiscent of some of the items in Alan Feuer's *The C Puzzle Book*. I now wish I had paid more attention to what Alan was saying. Despite this, and not being an expert on C, I thought I would make an attempt to discover the defects in the code and get the program running.

The first thing to notice is that the three anonymous enumerated types at the start of the listing are not referenced in the code, so I shall ignore them for the time being. I now observe the main structure that represents the deck of cards. It is an array of 52 cards. Each card being a structure containing three integers representing the card colour, its suit and value respectively. The fact that the colour of the card as well as its suit is being stored is a bit of a worry. It is always possible to deduce the colour of the card from its suit. Clubs and spades are always black and hearts and diamonds are always red. Maintaining a variable to keep track of the card's colour is redundant and runs the risk of becoming out of kilter with the colour of the suit. It is best removed.

The next thing to note is that the **LoadDeck** function initialises all but the last card in the deck. The loop variable should loop from 0 until it is not less than 52, not 51 as shown in the original listing.

The main problem lies in the way the loop variable, **i**, accesses the **Deck** array. The code incorrectly manipulates the **myDeck** pointer as if it were an array of pointers to a **Deck**. In fact, it is simply a single pointer to the **Deck**. To correctly access the suit, for example, of the card with index **i**, the following construction should be used.

```
(*myDeck)[i].suit
```

This first dereferences the `myDeck` pointer to obtain the first card in the deck. The card with index `i` is then obtained by means of the `[]` operator. Finally, the suit of the card is obtained. With the variable to store the card colour removed, the `LoadDeck` function becomes as shown below.

```
void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for (; i < 52; i++)
  {
    (*myDeck)[i].suit = i % 4;
    (*myDeck)[i].value = i % 13;
  }
}
```

We now come to the `PrintDeck` function. Again, the incorrect method of referencing the cards in the deck has been used here and can be corrected in the same way as for the `LoadDeck` function. Also, now that the variable to store the suit colour has been removed, the way the suit colour is obtained has to be amended. Specifically, the `suit`s array within `PrintDeck` has been simplified to become a one-dimensional array of suit names. The colour of the suit can now be obtained by taking the `suit` index of the card and determining if it is odd or even. If the `suit` index is odd, the suit colour is red. If the suit index is even, the suit colour is black. This is calculated by taking the `suit` index, dividing by 2 and using the remainder as in index into the colours array. This is achieved using the `%` operator. The revised `PrintDeck` function is shown below.

```
void PrintDeck(Deck *myDeck)
{
  int i = 0;
  for (; i < 52; i++)
  {
    const char *colours[] = {"black", "red"};
    const char *suits[] = {"clubs", "hearts",
      "spades", "diamonds"};
    printf("Card %s %d of %s\n",
           colours[(*myDeck)[i].suit % 2],
           (*myDeck)[i].value,
           suits[(*myDeck)[i].suit]);
  }
}
```

The `shuffle` function, like the previous two, incorrectly references the cards in the deck and can be corrected in a similar way.

```
void Shuffle(Deck * myDeck)
{
  int i = 0;
  for (; i < 52; i++)
  {
    int n = sizeof (Card);
    int to = rand() % 52;
    Card temp;
    memcpy(&temp, &(*myDeck)[i], n);
    memcpy(&(*myDeck)[i], &(*myDeck)[to], n);
    memcpy(&(*myDeck)[to], &temp, n);
  }
}
```

The program should now work as expected. Incidentally, there is no point in initialising the `Deck` array with all zeroes as is done in the second statement of `main()`. The `Deck` is correctly initialised by the (revised) `LoadDeck` function.

Marcel Marré <marre@links2u.de>

The code has several problems. The one actually leading to the crash is two different interpretations of the `Card` member `suit`. In `LoadDeck`, `suit` is initialised for each card with a value from 0 to 3. In `PrintDeck`, however, the strings for the suits are arranged in a two-dimensional array of two colours by two suits per colour. When these strings are used in the `printf` statement, the `Card`'s suit ranges from 0 to 3, and we read beyond the valid range of the array. We thus get an undefined `char*` for `printf` to output, which is liable to crash.

Let us fix `LoadDeck`. Apart from the inconsistent initialisation of `suit`, the function also initialises only a total of 51 cards. This does not lead to a crash, because the whole deck has been `memset` with 0s. This does mean, however, that one card is missing from the deck, and one card (value, suit and color all 0) is in the deck twice.

A more readable, logical version of `LoadDeck` would therefore be:

```
void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 52; i++)
  {
    myDeck[i]->color = i % 2;
    myDeck[i]->suit = (i / 2) % 2;
    myDeck[i]->value = i % 13;
  }
}
```

The initialised deck is somewhat oddly sorted, so another change would be to initialise `value` thus:

```
  myDeck[i]->value = (i / 4) % 13;
```

which is easier to follow.

Another point is that the `enum`s were not actually used at all. Using them to initialise a card is possible, although the following card:

```
Card myCard;
myCard.color = black;
myCard.suit = Hearts;
myCard.value = 5;
```

will be displayed as `"Card black 5 of Clubs"`.

Commentary

The code demonstrates confusion with pointers and arrays, not helped by the way that in C arrays will implicitly 'decay' to pointers to the first element in the array. (There are other problems, but I think those were more straightforward to understand and resolve.)

The code creates a single `Deck` object called `myDeck` and passes its address to `LoadDeck`. Inside `LoadDeck` this pointer treated as an array (`myDeck[i]`) of `Deck` objects. Unfortunately we've only passed in a single `Deck` and not an array of them, so every iteration round the loop after the first one accesses data beyond the end of `myDeck` and hence the crash.

Following the array subscript we have a pointer – what is this actually doing? Perhaps a simpler example will help:

```
struct data { int field; } items[10];
items->field;
```

The second line uses `items` which is of type '`array` of 10 `struct data`' but this type decays to a pointer to the first item when used in this context. The second line is equivalent to:

```
items[0].field
```

So returning to `LoadDeck` the first assignment to `myDeck` could be written as:

```
myDeck[i][0].color = i % 2;
```

The pair of subscripts makes it a little more obvious what's gone wrong, as a `Deck` is a simple array of one dimension.

It can be helpful, even when the data type is an array, to wrap it inside a `struct` for clarity.

```
typedef struct Deck
{
  Card card[52];
} Deck;
```

With this restructuring the 'broken' **LoadDeck** function no longer compiles as the **struct**, unlike an array, is no longer interchangeable with a pointer type; it can now be fixed by changing it to, for example:

```
void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 52; i++)
  {
    myDeck->card[i].color = i % 2;
    myDeck->card[i].suit = i % 4;
    myDeck->card[i].value = i % 13;
  }
}
```

While more verbose, it may be more understandable.

The winner of CC89

This critique contained several different problems as well as the 'it crashes' problem originally reported. The entrants did a pretty good job of identifying these problems. Paul pointed out that the code relies on 13 and 4 being mutually co-prime and Silas noted that populating **suit** and **color** separately was error-prone and the code as shown assigned the *wrong* color to some of the suits. James actually changed the code to remove the problematic field color. I thought Marcel's explanation of why the 2-dimensional **suit**s array was wrong was the clearest.

Silas also pointed out that it would be a good idea to seed the random number generator as otherwise the likelihood is that every run of the program will shuffle the deck exactly the same way!

Overall it was a hard call but I eventually decided to award Silas the prize for this code critique

Code Critique 90

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

I'm trying to instrument some code to find out how many iterator operations are done by some algorithms I use that operate on vectors. I've got some code that worked with C++03 and I'm trying to get it working with the new C++11 features. Mostly the C++11 code is just nicer but I can't get my wrapper vector to work with the new style for loop. I've stripped out the instrumentation for other methods in this example code so it only counts the increment operations and when I run the two simple examples below I get this output:

```
C: >example03.exe
Increments: 3

C: >example11.exe
Increments: 0
```

I expected the same output from both examples as all I've done is re-write the **for** loop using the new style **for** syntax.'

Can you find out why it doesn't work as expected and suggest some ways to improve (or replace) the mechanism being used?

The code is in Listing 2.

Listing 2

```
// -- wrapped_vector.hxx --
#include <vector>

template<typename T>
struct wrapped_vector : std::vector<T>
{
  typedef typename std::vector<T> vector;
  typedef typename vector::size_type
    size_type;
```

Listing 2 (cont'd)

```
  // Sort the constructors
  #if __cplusplus >= 201103
    using vector::vector; // Nice :-)

  #else
    // legacy: need to spell them out ...
    wrapped_vector() {}

    explicit wrapped_vector(size_type n,
      const T& value = T())
    : vector(n, value) {}

    template <class InputIterator>
    wrapped_vector(InputIterator first,
      InputIterator last)
    : vector(first, last) {}
    // ...
  #endif // C++11

  // print the stats
  static void dump();

  // instrumented iterator
  struct iterator : vector::iterator
  {
    typedef typename vector::iterator base;
    iterator() {}
    iterator(base it) : base(it) {}
    iterator& operator ++();
    // (other instrumented methods removed)

    static int increments;
  };

  // const_iterator (unused so removed)
};

#include "wrapped_vector.inl"

// -- wrapped_vector.inl --
#include <iostream>

template <typename T>
void wrapped_vector<T>::dump()
{
  std::cout
    << "Increments: "
    << iterator::increments
    << std::endl;
}

template <typename T>

typename wrapped_vector<T>::iterator&
wrapped_vector<T>::iterator::operator ++()
{
  base::operator ++();
  ++increments;
  return *this;
}

template <typename T>
int wrapped_vector<T>::iterator::increments;

// -- example03.cpp –
// (also works with C++11)
#include <cassert>
#include "wrapped_vector.hxx"
```

ACCU London – October 2014

Chris Oldwood reports from the latest meeting of the London Chapter.

Lies, damned lies and estimates – Seb Rose

Now that I'm not working in London it's a little more effort to make it into town to catch up with the ACCU London crowd, but I managed to make it along to the new Equal Experts offices near Warren Street tube station to see Seb Rose talk about the thorny issue of estimates.

There was a good turnout of just over 30 people to hear Seb talk. There was also some beer and nibbles laid on too which is always helps you deal with skipping dinner, especially when you know you'll be frequenting the local pub afterwards.

The title for his talk comes from a misattributed quote of Disraeli, but it nicely sums up what many people feel about the whole issue of attempting to predict how long a software project, or even just a single task, will take. The initial part of the talk focused on how bad we are at the process of estimating and looked into some of the 'science', such as Planning Poker, that suggests we are better at estimating relative to some known level of complexity rather than trying to predict an absolute amount. However it seems this has now been debunked too. Hopefully the crowd-sourcing aspect still provides value, as does it being another conduit for conversation and the breaking down of stories.

We didn't just get to listen to Seb but got to take part in a little experiment too. This involved us answering some general knowledge questions, such as when Bram Stoker was born, with a range estimate rather than a single value. The range however was supposed to be something we thought was

CHRIS OLDWOOD

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

Code Critique Competition 90 (continued)

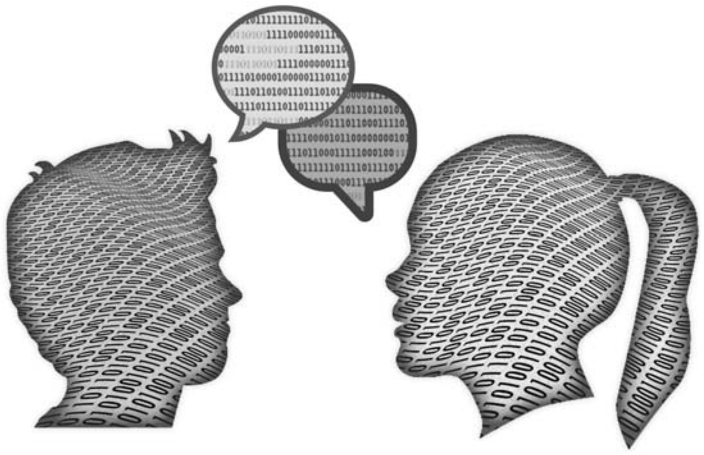
Listing 2 (cont'd)

```
int main()
{
  wrapped_vector<int> iVec;
  iVec.push_back(1);
  iVec.push_back(0);
  iVec.push_back(2);

  int total(0);
  for (wrapped_vector<int>::iterator
    iter(iVec.begin()), past(iVec.end());
    iter != past; ++iter)
  {
    total += *iter;
  }
  assert(total == 3);

  wrapped_vector<int>::dump();
}

// -- example11.cpp -- (C++11 example)
#include <cassert>
#include "wrapped_vector.hxx"
int main()
{
  wrapped_vector<int> iVec;
  iVec.push_back(1);
  iVec.push_back(0);
  iVec.push_back(2);
  int total(0);
  for (auto & p : iVec)
  {
     total += p;
  }
  assert(total == 3);
  wrapped_vector<int>::dump();
}
```



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

while (you care about code)
{
read (cvu && overload);
}
do(it);
because good code matters
ACCU
WWW.ACCU.ORG
PROFESSIONALISM IN PROGRAMMING

Seb Rose: An Interview

Emyr Williams continues the series of interviews with people from the world of programming.

Seb Rose is well known to most ACCU members. For those who do not know who he is, Seb is an independent software developer, trainer and consultant based in the UK. He specialises in working with teams adopting and refining their agile practices, with a particular focus on automated testing.

Since he first worked as a programmer in 1980 (writing applications in compiled BASIC on an Apple II) he has dabbled in many technologies for many companies, including Linn Smart, Amazon and IBM. He has just finished writing *The Cucumber-JVM Book* for The Pragmatic Programmers. His website can be found at www.claysnow.co.uk

How did you get in to computer programming? Was it a sudden interest? Or was it a slow process?

The maths department at my school had a PDP11 and a room with 8 terminals. I spent an afternoon watching someone write BASIC and wrote my first program the next day. The head of department then gave me a few photocopied sheets from Kernighan & Ritchie's *The C Programming Language* and I started learning more about what was happening under the surface.

A few years later I got my first holiday job working at a service station on the A3, getting paid the princely sum of 60p an hour. It sounds grim, but I earned enough to buy a record deck (anyone remember the PL-512?), which meant I needed even more money to buy albums. A neighbour put me in touch with his accountant who was in partnership with a guy in Teddington who was writing accountancy software in his attic. I went round for a chat and showed off my BASIC skills and landed the job. He apologetically told me that all he could afford to pay was £3 an hour – a 500% pay rise! I was ecstatic and I've been programming (on and off) ever since.

I treat most of the people I meet as potential mentors in one way or another

What was the first program you ever wrote? And in what language was it written in? Also is it possible to provide a code sample of that language?

BASIC. Nothing very exciting, just asking for input and printing out a response. I then took a huge leap and tried to write a text-based Dungeons and Dragons game in C. It never worked, but I spent many a happy hour tinkering about with source code on huge print-outs. I returned to BASIC (compiled for the Apple II) for my holiday job and spent a lot of time doing screen layouts using 80 x 24 grids. There were some lovely hacks for positioning the cursor at a specific location on the screen which we used to access via `GOSUB`s whenever we needed to present output.

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

Tricky question. Maybe the advice from Steve Freeman and Nat Pryce to "Listen to your tests". It's all too easy to blame a technique (or tool) that you're trying to learn, when actually it's your failings in other areas that are the root cause of the problem. When I find it hard to write a unit test, I remember their advice and look at the code I'm trying to test with a critical eye.

The old favourite "don't optimise (yet)" is regularly useful. As developers we often think about performance prematurely. The 'shameless green' stage of TDD encourages us to get the test working without thinking too hard about the design. It's the 'refactor' step where we improve the design of the code, but even here performance should generally be a subsidiary concern to readability. There will always be situations where you need every ounce of performance you can get, but they are few and far between in most domains, and you should only pursue optimisation once you actually have the data to show what actually needs optimised.

If you were to go back in time and meet yourself when you were starting out as a programmer, what would you tell yourself?

I'd probably tell myself, "Good choice." It's hard to think of a career that has so many varied opportunities or that would have allowed me to work in so many different areas. I've been a freelancer for most of my career, and the occasional breaks between contracts have been invaluable for learning new skills and trying different things.

I ran an organic smallholding for 12 years, at the same time as working as a contractor. I tried to have the contracts end in the spring, so that I could spend the sunny, Scottish days doing more physical work outdoors. For 3 years around 2003 I ran the organic business full-time and didn't do any commercial programming. I actually found that I spent more time at a desk during this period than I did when I was contracting – maintaining the website and order system, generating delivery reports, dealing with customers etc.

It was a relief to be able to return to programming when, after our best year trading, the accounts showed I'd only made £12,000 from delivering organic produce.

Do you currently have a mentor? And if so, what would you say is the best piece of advice you've been given by them?

I haven't got a specific mentor, but I treat most of the people I meet as potential mentors in one way or another. Recently Jon Jagger told me he spends 1/3 of his time earning money, 1/3 working on OSS and 1/3 fishing. I'll not be taking up fishing, but this seems like a mix to aspire to.

How do you keep your skills up to date? Do you get a chance to do some personal development at work?

I spend a lot of time reading blogs and books, as well as going to conferences. In fact, you could say that my job IS keeping up to date, at least with a small segment of the industry.

Twitter is a great source of information – not the throw-away one liners, but the links to blogs that I wouldn't normally notice.

And I organise the local BCS branch events, which forces me to talk to a lot of people and actively seek out new and interesting topics and speakers.

EMYR WILLIAMS

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

What would you describe as the biggest "ah ha" moment or surprise you've come across when you're chasing down a bug?

Working with Weblogic in the early days of EJB 1.0 we were getting inexplicable , intermittent failures. Days of investigation later, having boiled the sample code down to something really small, it became clear that there actually was a problem in the application server. We submitted a bug report and, in due course, received a patch. Months later the problem reappeared – the supplier had rolled out an upgrade *without* the patch.

I remember the first time I heard you talk was at the ACCU Conference where you were talking about Test Driven Development. There has been a lot of talk of late of TDD is dead (among the more extreme things I've seen) (http://www.infoq.com/news/2014/06/tdd-dead-controversy) I'm guessing you wouldn't agree with that assessment? Or is it a case of the way TDD is done, and how it's implemented that's the issue?

TDD is not dead, but neither is it a silver bullet

I'm still talking about TDD. My current opinion is that TDD is a technique that is generally useful, but that context is, as always, an important consideration. The arguments stem from consultants making general statements that address segments of the development community without making it clear which segments they apply to. I say more about this in my session "So long, and thanks for all the tests" which is online at http://vimeo.com/105861375

The short answer is that TDD is not dead, but neither is it a silver bullet. It's a useful technique to have in your toolkit, but like so many techniques it's easier to describe than do. That's why I think all developers should learn and practise TDD, at least until they know it well enough to make a considered judgement about whether it's useful for them in their current role.

How scary was it to go from full time developer to freelance developer? And how long did it take for you to feel confident to go for it?

I graduated in 1987 and did my first contract in 1992. In the intervening period I did 2 full-time jobs, one for 8 months and one for 5 months. So, I think it's fair to say that I've always been freelance. I guess that confidence has not been too much of a problem for me.

That's not to say that I haven't worried about where my next paycheck will come from. I have, and at times where continuity is more important to me, such as when my children were young, I have returned to permanent employment. Strangely, though, I'm generally less stressed working as an independent than an employee – and I don't think I'm alone in this.

Do you have any regrets as a programmer? For example wishing you'd followed a certain technology more closely or something like that?

I wish I'd read more books and bought fewer.

Where do you think the next big shift in programming is going to come in?

I have no idea. In the nineties, when first introduced to HTML and the internet, I said the equivalent of "it'll never catch on." Niels Bohr is credited with saying "Prediction is hard (especially about the future)" and I'm worse than most at making predictions.

Finally, what advice would you offer to kids or adults who are looking to start a career as a programmer?

I think I'd give the same advice for any domain, not just programming. Have a go. Do something that interests you and keep trying. Qualifications may help, but enthusiasm and aptitude are by far the most important ingredients.

ACCU London – October 2014 (continued)

90% accurate, which should have lead to us getting around 9 out of 10 questions right. Of course most of us were way out even with that kind of leeway.

why we are often being asked to provide these estimates in the first place – what use are they to The Management that demands them?

This nicely brought home the point of how bad we are at estimating and the quiz theme continued with us trying to answer the age old question of 'how long is a piece of string'. Although again, rather than giving a measurement we just needed to compare it to the others and say which was longer. Naturally they weren't all laid out end-to-end but looped around in different ways to make the process much harder; another simple but effective game.

There were various references to popular books that cover these topics, such as *Waltzing with Bears* and *Impact Mapping*, with plenty of quotes and snippets that intrigued me. I already own a couple but suspect a visit to my local online bookshop is imminent as there is plenty more to lap up here, particularly in the area of breaking down user stories. One particular slide with a flowchart titled 'How to Split a User Story' looked most promising [1].

The final part of the talk looked at why we are often being asked to provide these estimates in the first place – what use are they to The Management that demands them? Seb delved into the area of Return on Investment (ROI), which is a fairly simple calculation, assuming you can of course quantify the 'value' that a project will bring. This area of business analysis seems even murkier than the process of estimating that we are being asked to do. Presumably this was aimed at internal projects as he'd already covered estimating for external clients at the very beginning with a selection of quotes from builders.

The talk lasted well over an hour and certainly provided plenty of leads for those that wanted to learn more. The evening was rounded off nicely with a trip to a local pub (Bree Louise) which had a number of real ales on tap. This was a great reminder that I should make it into London more often to catch these meet-ups.

Reference

[1] http://www.agileforall.com/wp-content/uploads/2012/01/Story-Splitting-Flowchart.pdf

Standards Report

Mark Radford brings the latest news from C++ Standardisation.

Hello and welcome to my latest standards report.

Once again the timing of this report is a bit off. Why? Because the big news should be that C++14 has become an international standard. However, because of the timing, the chances are that everyone reading this already knows about it (*CVu* has a production deadline approximately a month before it appears in print, therefore I submitted my previous report for publication just before C++14's ratification). At the end of my last report, I said that C++14 was at its Draft International Standard (DIS) stage and, if none of the national bodies submitted a 'no' vote, then it could become an international standard. That is what happened: by the end of the DIS period (August 15th) all the member countries' national bodies had voted 'yes', and the DIS became the ISO standard for C++14.

The above could happen owing to a recent change in ISO procedures aimed at helping to speed up the release of standards. Previously, if any national bodies issued comments with their vote on the DIS (even if it was a 'yes' vote) then there would be a Final Draft International Standard (FDIS), and another round of voting. With the recent change, if the 'yes' vote is unanimous – even if there are comments, which there can be, even with a 'yes' vote – an FDIS will be issued only if the standards committee explicitly requests it.

Is there anyone out there with strong Ruby knowledge? IST/5 – that's the BSI committee handling programming languages – is interested in enrolling a Ruby representative. I don't have any more details: the obvious assumption is that this would/could lead to the formation of a BSI Ruby Panel (just as there is a BSI C++ Panel). In any case, if there is anyone out there able to (and interested in) represent Ruby on IST/5, feel free to contact me in the first instance and I'll put you in touch with the right person.

I last reported on C standardisation a couple of reports ago, so here's what's been happening since. Part 1 of the floating point TS is now published (last time it had been approved but was awaiting publication). Part 2 (which, last time, had recently been moved to the DTS stage) has now been approved for publication, but is not yet published (but that's just a matter of time). Parts 3 and 4 have gone through their PDTS stage: the next WG14 (ISO C Committee) will consider responses to ballot comments on the PDTS. There is a part 5 to come, but as yet no draft is available. There is, however, a public draft of the document 'Programming languages – C – Extensions for parallel programming' (N1862) available, which can be found at: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1862.pdf. I assume this is to be a TS (it looks like that sort of document to me), but it doesn't say so on the document and I don't, as yet, have any clarification.

The next ISO C++ meeting will be held in Urbana-Champaign, IL, USA, 3rd–8th November (which means by the time this report is published the meeting will have already happened). I see that the mailing for this meeting is not yet published (and probably won't be before I submit this report for publication), so I won't be talking about any of the papers included in it. There was a face to face meeting of SG1 (the study group for concurrency and parallelism) in early September, and I can report briefly on day one of that meeting (sorry but I don't have any information about day two). More of that below, but first let me mention that the transactional memory extensions TS passed the ballot needed for the new work item to be official. Therefore we now have: 'TS 19841: C++ Extensions for Transactional Memory'.

if the 'yes' vote is unanimous – even if there are comments, which there can be, even with a 'yes' vote – an FDIS will be issued only if the standards committee explicitly requests it

Moving on to the SG1 meeting, in my previous report I pointed out that the 'Executers and Schedulers' section had been removed from the draft concurrency TS (N4107). Until its removal, that section had come from the proposal 'Executors and schedulers, revision 3' (N3785). It was removed to allow for the consideration of the 'competing' proposal 'Executors and Asynchronous Operations' (N4046) by Christopher Kohlhoff, the latter proposal having received a very positive reception at the last ISO meeting (Rapperswil). That's where things had got to before the SG1 meeting. When the meeting took place a draft follow-up to N4046 was available, containing actual wording for the TS (N4046 did not contain wording, confining itself to a description of the proposal, to check the level of interest). Also available was a draft revision of an N3785 follow-up, updated in response to N4046.

So far, so good, but then things started to go wrong. Originally there was going to be no facility to attend the meeting remotely. Although this facility was provided at the last minute, it was too late for Christopher Kohlhoff (who was not able to attend in person) to be able to attend remotely. On the other hand Chris Mysen (N3785 co-author) attended in person, and therefore was able to represent that proposal and its draft follow-up.

The discussions ended in a straw poll, the result of which (fourteen in favour, two against) suggested that Chris Mysen's proposal should serve as the starting point, and what parts of Chris Kohlhoff's proposal cannot be implemented in terms of it should be researched. The results are to be discussed at the forthcoming Urbana-Champaign ISO meeting. I don't understand this as it seems to be starting in the wrong place, because Chris Kohlhoff's proposal has a far better track record of standing up to scrutiny than Chris Mysen's.

Before wrapping up I have some acknowledgements. Thanks to Roger Orr for drawing my attention to the news about the transactional memory TS, and for letting me know about the change in ISO procedure that allowed C++14 to become an international standard without the need for an FDIS. Thanks to Jamie Allsop for his excellent report (to the BSI C++ Panel) on day one of the SG1 meeting (which he was able to attend remotely); I have drawn heavily on Jamie's report for my reporting on this meeting. Finally thanks to BSI C Panel convenor Joseph Myers for updating me on C standardisation progress.

MARK RADFORD

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

Bookcase

The latest roundup of book reviews.

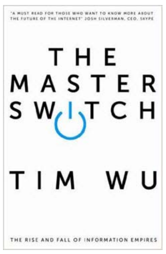
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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

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

Astrid Byro (astrid.byro@gmail.com)

The Master Switch



By Tim Wu, published by Atlantic Books, 368pp ISBN 978-1-84887-9867

Reviewed by Alan Lenton

Highly recommended.

Subtitled 'The Rise and Fall of Information Empires' Tim Wu's book is a tour de force history of the four great information technologies of the 20th Century – the telephone, radio/television, movies, and the internet. The book is both a history and an analysis of these industries. The lessons we can draw from the stories he tells have serious implications for the current struggle over what is now known as 'net neutrality.

The individual stories of the technologies themselves are interesting enough in their own right, but what is striking is the common themes of the histories of the telephone, radio and movies. In each case as the new disruptive technologies came into existence and there was a period of free for all, anarchy if you like, in which innovators thrived, anyone could join in, and the cost of entry was minimal.

Then came a period of consolidation, often assisted by government desire to regulate and consolidate. Politicians are notoriously wary of their constituents doing this for themselves, while the bureaucrats who run the regulatory bodies always push for consolidation. After all it's a lot easier to talk to, and come to agreement with, a few large bodies that have a similar culture, than hundreds of small organization filled with fractious non-conformists!

And of course, once you have a monopoly or semi-monopoly situation, it becomes easier to suppress new, disruptive, innovations – the suppression of FM radio in the early 30s by RCA being a classic case. In other cases the leadership of the monopoly involved simply could not conceive of any way of working other than the one currently in use. Thus the officials at AT&T thought the concept of packet switched networks (the basis of the internet) was 'preposterous'. In fact, so wedded were the AT&T officials to the circuit based network (the AT&T slogan was One company, One system, Universal Service), that they even turned down a US Air Force offer to pay for an experimental packet switched network!

But this isn't just a technical history. It's also a social history of the struggle to keep those technologies in the hands of ordinary people, and that is as important as the technical issues, because that is exactly what is happening now in both the internet and the software forums. In the internet the struggle is being waged under the rubric of 'net neutrality, while the software struggle is being waged through patent reform.

Both are important. At the moment anyone can post material onto the net – you don't require anyone's permission to do so, or to check what you've written before it's posted. Anyone can write software – all you need is a general purpose computer, usually a desktop PC, and a compiler or a browser, depending on your language of choice. Do I really have to tell you that the politicians and big business would prefer it otherwise?

We are on a cusp when it comes to questions of how the new and currently cheap enabling technologies of computing and the internet will be used in the future, and Tim Wu's readable and fascinating book is an important chronology and analysis of what happened on previous occasions. We need to understand that and learn its lessons, because those who fail to learn from history are doomed to repeat it.

ACCU Information

Membership news and committee reports

accu

View from the Chair

Alan Lenton
chair @accu.org

There's not a lot happening on the surface at the committee at the moment. We are looking at revised guidelines for the sponsorship of local groups, and that should be available in the near future. In the meantime I'm wrestling with putting a discussion paper together to lay out some guidelines for the future development of ACCU.

The premise is easy. How do we reverse the decline in membership, and build up a thriving and dynamic organization? The solution is not, and data is not easily obtainable. For instance, I'd like to know to what extent the age of the membership has changed since (say) the late 1990s, but no one thought to ask that at the time, so I have no baseline. I suspect, but I have no proof, that the membership has aged along with the organization.

That makes a lot of difference. People who were footloose and fancy free fifteen or twenty years ago, by now will probably be married and have children. Inevitably they will have less time to devote to an organization like ACCU. So we have to take that into account, when we decide where we are going.

But, of course, that's not the only thing that's changed. Even in the late nineties you could still get into the profession of programmer without formal qualifications, but that opening is now really closed off, and you need a degree. This has implications about where we look for new and younger members.

Technology has changed. On a pure programming level most of the C++ programmers among us are now writing code in 'Modern C++' , and new languages abound, as do new versions of older languages. The rise of mobile devices has given rise to a completely different mindset for this sort of programming.

We haven't quite got to the stage of computing power as a utility yet, but cloud computing is steadily making headway. In fact I suspect that there will be a lot more private clouds around than most people expect, especially given the revelations about government snooping. In the meantime we have seen the rise of DevOps, and interesting new technologies like Docker, all of which will change the environment in which we work.

At a completely different level we are looking at the start of the 'Internet of Things'; if it really does take off, and personally I think it will – eventually. However, it will probably take a lot longer than its proponents expect. That has implications too, for the balance of how and where programmers are employed.

A lot of programmers are currently employed by software houses. Those of you who know me will know that it's my belief that a lot of large organizations don't yet realize that they are now software houses. Banks are the classic case. The people running them are under the illusion that they are banks with a large software department, whereas any rational analysis of their work would make it clear that they are software houses with a banking licence!

But I digress. My point is that, with the rise of the Internet of Things, the balance will change, as the number of embedded programmers grows, and most of them will be working in small software departments in businesses producing physical objects with digital controllers. Is this an opportunity, for instance to provide the peer discussion and keeping up with trends that we take for granted in larger environments? If so, how do we do it?

But even our regular activities are subject to change. At the moment our key activities are our journals, the conference each year, and a small number of local group meetings. But even these, successful as they are, are products of the time when we started the ACCU. Much publication is going digital. On the conference side we see the rise of webinars and the likes of TED. Local meetings socially derive from the flourishing pub culture of ten to fifteen years ago – a culture which sadly no longer exists.

Does this mean we should abandon what we already have, and jump onto the next passing bandwagon? Absolutely not. The new stuff is important. We ignore it at our peril, but to move forward we need to build on what we already have, and what our strengths are. And therein lies the rub. How do we bring in the new, without damaging the old?

Answers on a postcard – oops, sorry, showing my age there – email to: chair@accu.org. All suggestions will be read. Ones that I disagree with will be ruthlessly trashed and ones that I agree with will be ruthlessly ...assimilated... without attribution!

Joking apart, I would welcome ideas from members about how we should go forward in the future. I know people have ideas – they don't have to be fully developed, and they don't have to be a master plan covering everything.

I could write a way forward without any input from the members, but it would probably be very one sided and would generate all sorts of angst. More to the point, the more people who put their ideas in at this stage, the wider choice of options we will have for the future. By having this discussion now, we are having it at a point where we do have a choice of what to do. If we don't have this discussion soon we will have no choices left to make!