the magazine of the accu

www.accu.org

{cvu}

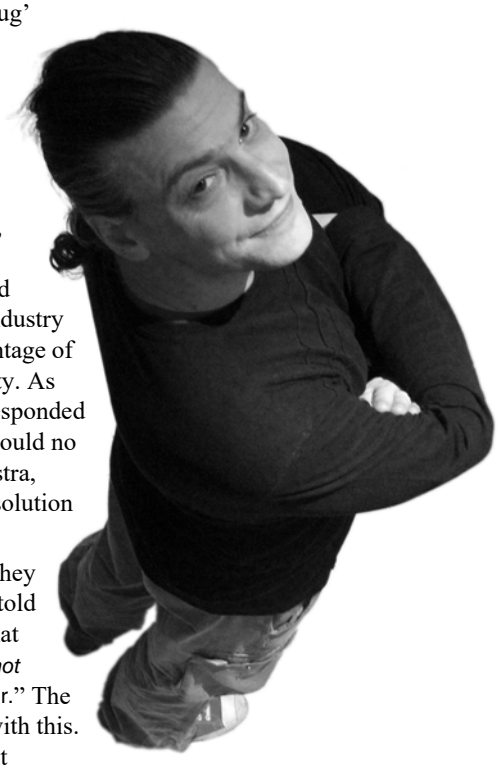
Volume 28 • Issue 6 • January 2017 • £3

Features

Regulars

{cvu}
Volume 28 Issue 6
January 2017
ISSN 1354-3164
www.accu.org

No obvious deficiencies



Edsger Dijkstra didn't approve of the term 'bug' when talking or writing about code, preferring instead the more brutal – and honest – 'error'. He was also an advocate of being able to formally prove code correctness, and famously spoke out against the unrestricted use of the `goto` statement. Much of his writing covered what was termed 'The Software Crisis', coined in 1968 to describe the fact that developments in computer hardware out-stripped those in software, resulting in a programming industry unable to produce software that could take advantage of it reliably. The crisis was really about complexity. As hardware became more capable, software had responded by becoming more complex to the point that it could no longer be understood by its creators. What Dijkstra, Tony Hoare, and others, identified was that the solution should be more *simplicity*.

Then they had a new crisis, because they knew they had no idea how to achieve it. In 1999, Dijkstra told the ACM Symposium on Applied Computing that "we have to keep the design simple [...] but we do *not* know how to reach simplicity in a systematic manner." The software industry as a whole is *still* struggling with this. We have found ways to solve ever more difficult problems, produce increasingly sophisticated systems, and have developed techniques and practices that address some of the issues of the 1970s and 80s regarding project management, software testing, parallelism and high-level abstractions. In doing so, however, we have introduced new complexity.

We have become so successful at applying technology and computing that it appears (to me, anyway) that we think we can apply it to *anything*. And we do so without apparently much thought for the consequences. The Internet of Things is growing, and already the lack of security in these devices is being called a Crisis. It is, at its heart, a software problem, so it's up to us – the programmers – to do something about it. The Software Crisis of the 1970s/80s never ended, it just changed its stripes to blend in.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 29.1: 1st February 2017
C Vu 29.2: 1st April 2017

Overload 138: 1st March 2017
Overload 139: 1st May 2017

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

Speak Up! (Part 2)

Pete Goodliffe talks to us about communication.

First learn the meaning of what you say, and then speak.
~ Epictetus

In the previous 'Becoming a Better Programmer' column I started to look at how we, as programmers, *communicate*. We reminded ourselves that code *is* communication, and considered how to improve our communication in that medium.

Now, let's look at the communication most people would expect us to 'talk about' – interpersonal communication. Programmers are never solely confined to the act of writing code. We have to work with other people: with other coders, with the wider development team (testers, UX, managers), and even – shock horror – with the customer.

Interpersonal communication

We don't just communicate by typing code. Programmers work in teams with other programmers. And with the wider organisation.

There's a lot of communication going on here. Because we're doing this all the time, high-quality programmers *have* to be high-quality communicators. We write messages to speak with, even gesticulate at, others all the time.

Ways to converse

There are many communication channels we use for conversations, most notably:

- Talking face-to-face
- Talking on the phone, one-to-one
- Talking on the phone in a 'conference call'
- Talking on VoIP channels (which isn't necessarily different from the phone, but is more likely to be hands-free and allow you to send files over the same communication channel)
- Email
- Instant messaging (e.g., typing in Skype, on IRC channels, in chatrooms, or via SMS)
- Videoconferencing
- Sending written letters via the physical postal system (do you remember that quaint practice?)
- Fax (which has largely been replaced by scanners and common sense; however, it still has a place in our comms pantheon because it is regarded as useful for sending legally binding documents)

Each of these mechanisms is different, varying in the locations spanned, the number of people involved at each end of the communication, the facilities available and richness of interaction (can the other person hear your tone of voice, or read your body language?), the typical duration, required urgency and deferrability of a discussion, and the way a conversation is started (e.g., does it need a meeting request to set up, or is it acceptable to interrupt someone with no warning?).

They each have different etiquettes and conventions, and require different skills to use effectively. It is important to select the correct communication channel for the conversation you need to have. How urgent is an answer? How many people should be involved?

Don't send someone an email when you need an urgent answer; email can sit ignored for days. Walk over to them, ring them, Skype them. Conversely, don't phone someone for a non-urgent issue. Their time is precious, and your interruption will disrupt their *flow*, stopping them from working on their current task.

When you next need to ask someone a question, consider whether you are about to use the correct communication mechanism.

> Master the different forms of communication. Use the appropriate mechanism for each conversation.

Watch your language

As a project evolves, it gains its own dialect: a vocabulary of project and domain-specific terms, and the prevalent idioms used to design or think about the shape of the software design. We also settle on terminology for the process used to work together (e.g., we talk about *user stories*, *epics*, *sprints*).

> Take care to use the right vocabulary with the right people.

Does your customer need to be forced to learn technical terms? Does your CEO need to know about software development terminology?

Body language

You'd be upset if someone sat beside you, sparked up a conversation, but spent the whole time facing in the opposite direction. (Or you could pretend they were from a bad spy movie; I hear the gooseberries are doing well this year...and so are the mangoes. [1])

If they pulled rude faces every time you spoke, you'd be offended. If they played with a Rubik's cube throughout the conversation you'd feel less than valued.

As a project evolves, it gains its own dialect: a vocabulary of project and domain-specific terms

It is easy to do exactly this when we communicate electronically; to not fully respect the person we're talking with. On a voice-only conversation, it's easy to zone out, read email, surf the Web, and not give someone else your full attention.

Having fully embraced our modern, always-connected, broadband age, I now default to selecting a *video-on* communication channel. Often I'll kick off a conversation that might have been via phone or instant message with a VOIP video chat. Even if my conversant will never enable their own video, I like to broadcast a picture so that my face and body language are clearly visible.

This shows I'm not hiding anything, and fosters a more open conversation.

A video chat forces you to concentrate on the conversation. It engages the other person more strongly, and maintains focus.

Parallel communication

Your computer is having many conversations at once: talking to the operating system, other programs, device drivers, and other computers.

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

It's really quite clever like that. We have to make sure that *our* code communication with it is clear and won't confuse matters whilst it's having conversations with other code.

That's a powerful analogy to our interpersonal communication. With so many communication channels available simultaneously, we could be engaging in office banter, instant messaging a remote worker, and exchanging SMSs with our partner, all whilst participating in several email threads.

And then the telephone rings. Your whole tottering pile of communication falls over.

How do you ensure that each of your conversations is clear enough and well-structured so it won't confuse any other communication you're concurrently engaged in?

I've lost count of the number of times I've typed the wrong response into the wrong Skype window and confused someone. Fortunately, I've never revealed company confidential information that way. Yet.

Communication within a team must be free-flowing and frequent

> Effective communication requires focus.

Talking of teams

Communication is the oil that lubricates teamwork. It is simply *impossible* to work with other people and not talk to them.

This, once more, underscores Conway's law. Your code shapes itself around the structure of your teams' communications. The boundaries of your teams and the effectiveness of their interactions shapes, and is shaped by, the way they communicate.

> Good communication fosters good code. The shape of your communications will shape your code.

Healthy communication builds camaraderie, and makes your workplace an enjoyable place to inhabit. Unhealthy communication rapidly breaks trust and hinders teamwork. To avoid this, you must talk to people with respect, trust, friendship, concern, no hidden motives, and a lack of aggression.

> Speak to others transparently, with a healthy attitude, to foster effective teamwork.

Communication within a team must be free-flowing and frequent. It must be normal to share information, and everyone's voice must be heard.

If teams don't talk frequently, if they fail to share their plans and designs, then the inevitable consequences will be duplication of code and effort. We'll see conflicting designs in the codebase. There will be failures when things are integrated.

Many processes encourage specific, structured communication with a set cadence; the more frequent the better. Some teams have a weekly progress meeting, but this really isn't good enough. Short *daily* meetings are far better (often run as *scrums*, or *stand-up* meetings). These meetings help share progress, raise issues, and identify roadblocks without apportioning blame. They make sure that everyone has a clear picture of the current state of the project.

The trick with these meetings is to keep them short and to the point; without care, they degrade into tedious rambling discussions of off-topic issues. Keeping them running on-time is also important. Otherwise they can become distractions that interrupt your *flow*.

Talking to the customer

There are many other people we must talk to in order to develop excellent software. One of the most important conversations that we must hold is with the customer.

We have to understand what the customer wants, otherwise we can't build it. So you have to ask them, and work in their language to determine their requirements.

After you've asked them once, it's vital to keep talking to them as you go along to ensure that it's still what they want, and that assumptions you make match their expectations.

The only way to do this is in their language (not yours), using plenty of examples that they understand – for example, demos of the system under construction.

Other communication

And still, the programmer's communication runs deeper than all this. We don't just write code, and we don't just have conversations. The programmer communicates in other ways; for example, by writing documentation and specifications, publishing blog articles, or writing for technical journals.

How many ways are you communicating as a programmer?

Conclusion

A good programmer is hallmarked by good communication skills. Effective communication is:

- Clear
- Frequent
- Respectful
- Performed at the right levels
- Using the right medium

We must be mindful of this, and *practise* communication – we must seek to constantly improve in written, verbal, and code communication. ■

Questions

- How does personality type affect your communication skills? How can an introverted programmer communicate most effectively?
- How formal or casual should our interactions be? Does this depend on the communication medium?
- How do you keep colleagues abreast of your work without endlessly bugging them about it?
- How does communication with a manager differ from communication with a fellow coder?
- What kind of communication is important to ensure that a development project runs successfully?
- How do you best communicate a code design? They say a picture speaks a thousand words. Is this true?
- Do distributed teams need to interact and communicate *more* than co-located teams?
- What are the most common barriers to effective communication?

Reference

[1] See the 'Secret Service Dentists' sketch from *Monty Python's Flying Circus*

A Case of Mistaken Identity

Chris Oldwood puts values to the test.

I recently unearthed a bug in some C# code where, superficially, the cause appeared to be a single character, but on closer inspection it was not entirely clear what the author's intentions really were. This was down to a number of factors, not least the lack of tests, but it got me thinking about what those intentions might have been and what other practices could have prevented it in the first place.

The bug

The line of failing code was a simple conditional statement that would induce some error handling logic when true. My suspicion there was a bug were immediately aroused when the tool reported an error code of 'none'. If you've ever seen a Windows application report an error with the text 'The operation completed successfully' then you'll know what I'm talking about. This was the statement:

```
if (result != Error.None)
{
  // report error
}
```

Naturally I switched to the `Error` class definition to see what it looked like. It was a pretty sparse class with `None` defined as a simple static property alongside a couple of other static methods (see Listing 1).

Initially I didn't spot the mistake, but I realised from the way the conditional statement was behaving that it had to be something to do with the object's identity. The lack of an `Equals()` implementation meant it would be using reference based equality, not value semantics, which suggested that the two objects were not exactly the same object in memory.

My gut instinct was that the return site was creating the object directly instead of using the static property and so I was surprised when it was correct, like so:

```
return Error.None;
```

Listing 1

```
public enum ErrorCode
{
  None,
  NotFound,
  AccessDenied,
}
public class Error
{
  public ErrorCode Code { get; }
  public string Message { get; }
  public Error(ErrorCode code,
    string message = "")
  {
    Code = code;
    Message = message;
  }
  public static
    Error None => new Error(ErrorCode.None);
  public static
    Error NotFound(string message) =>
    new Error(ErrorCode.NotFound, message);
  public static Error
    AccessDenied(string message) =>
    new Error(ErrorCode.AccessDenied, message);
}
```

Scratching my head I went back to the `Error` class, studied it more closely, and then I noticed the subtle mistake – the extra `>` in the property definition. This static property, instead of creating a *single* instance of the value when the class type is initialised, creates one *every* time it's invoked! This explains why the reference based comparison fails – it will *never* be the same object in memory.

New syntax, new bugs

Prior to C# 6, the syntax for defining static properties was somewhat more verbose. For example if you wanted to declare a static property backed by a value, you would write:

```
public static Error None
  = new Error(ErrorCode.None);
```

This creates a writeable property. If you wanted a read-only static property you could use a custom `get`ter like this:

```
public static Error None {
  get { return new Error(ErrorCode.None); }
}
```

With C# 6, however, came a simplified syntax especially for read-only properties which either have automatic backing storage or are implemented by an expression body. Hence, the former now becomes:

```
public static Error None { get; } =
  new Error(ErrorCode.None);
```

…and the latter can be turned into just this:

```
public static Error None =>
  new Error(ErrorCode.None);
```

These two forms together don't look so similar but if you compare the very first and last side-by-side you'll see they are incredibly similar:

```
public static Error None =
  new Error(ErrorCode.None);
public static Error None =>
  new Error(ErrorCode.None);
```

The only difference between these two statements in fact is a single character, the `>`, which turns the assignment operator into the lambda arrow operator. Semantically, this changes a writeable static property which has a value created at class initialisation, into a read-only static property backed by an expression body that returns a new value every time it's invoked. As you can see these two are functionally very similar but, as we've just seen from the bug, subtly different depending on the value semantics of the enclosing type.

The fix, or is it?

The 'obvious' minimal fix from the calling code's perspective is to remove the extraneous `>` character from the property definition and ensure that there is only one instance of `None` ever retrieved from it. With only a single instance in play, a reference based equality comparison will always succeed. In fact, we can go one better and avoid the backdoor that allows `None` to be replaced by using the `readonly` keyword to ensure that it

CHRIS OLDWOOD

Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race andcan be easily distracted via gort@cix.co.uk or @chrisoldwood

cannot ever be changed by a client or the class itself (outside the type constructor):

```
public readonly static Error None =
  new Error(ErrorCode.None);
```

That fixes the actual bug, but it still leaves a number of questions. For example, was the reference based comparison really the intention, and if so, how hard did the author try to avoid letting a caller create a duplicate object *like* `None`? As you can see the `Error` type's constructor was marked `public` and so the consuming code could have just as easily written this instead:

```
return new Error(ErrorCode.None);
```

Once again the reference based comparison would have failed because the returned object was never the same as the single instance held by the static property.

Another observation is that the underlying error value (from the `ErrorCode` enumeration) is accessible via a public read-only property:

```
public ErrorCode Code { get; }
```

Therefore, it's entirely possible the author may have intended for the conditional statement to be written like this instead:

```
if (result.Code != ErrorCode.None)
```

Now we're doing a comparison of two values from an enumerated type and therefore we get a value based comparison instead which will always succeed irrespective of whether `result` holds the `None` singleton or not. The big similarity in the name of the `Error` class and the name of the underlying `ErrorCode` enumeration could easily explain how one could have been written instead of the other.

Factory methods

Perhaps we are on the right lines with assuming that reference based equality was desirable if we bring in an observation around the two factory methods, which also use the new expression body syntax to make them light on ceremony:

```
public static Error NotFound(. . .) =>
  new Error(. . .);
public static Error AccessDenied(. . .) =>
  new Error(. . .);
```

Maybe there is one other mistake and the `Error` constructor was never intended to be made `public` in the first place. Perhaps it should have been marked `private` so that clients are forced to create an instance of the object thorough one of the factory methods, or use the `None` value. Hence, in the scenario of an error, the returning code would therefore write something this:

```
return Error.NotFound($"{path} not found");
```

Given the potential for a free format message, it's highly unlikely the calling code would ever attempt to perform an equality comparison on an instance of the `Error` type, except for a comparison against `None`. Consequently these two little fixes (one to the `None` property and the other to the constructor) might be enough for us to declare the problem 'properly fixed'.

Value semantics

Up until this point, I have worked on the assumption that the author really did want the comparison using the `==` operator to work correctly and so we've found ways to make that happen for the scenario that initially brought it to our attention. However, these little tweaks feel somewhat akin to moving the deckchairs around on the Titanic as we're not really fixing the type itself to perform all comparisons correctly.

The `Error` type is composed of two values which should both take part in a proper value based comparison. As anyone who attended Steve Love and Roger Orr's ACCU talk a few years back about implementing equality comparisons will remember, these are non-trivial things to implement in languages like C# and Java. There is a lot of boilerplate code to add (and a `GetHashCode()` implementation for completeness) which many IDEs will happy produce for you to save typing and ensure all the options are correctly covered, but the meat is essentially just a comparison of the two embedded properties:

```
public static
  bool operator==(Error lhs, Error rhs)
{
  return (lhs.Code == rhs.Code) &&
         (lhs.Message == rhs.Message);
}
```

Now when we invoke our original failing comparison against `None` in the client's error handling code, we will get a fuller comparison of the object's embedded members instead of just their references. Additionally, because we're performing a deeper comparison, we don't *have* to change the `None` definition because any instance of `None` will now be equivalent to any other (although creating one every time could be considered wasteful).

This feels like a more complete fix for our equality bug but I'm still left wondering if it's the right thing to do. Making a type into a value type just because I want an equality comparison to do the right thing in one scenario does not feel overly compelling. When pondering whether to make something a value type or not, I have an acid test that asks if I'd ever likely use it as a key in a dictionary or put it into some kind of set.

Once again, the free-text error message property suggests to me that this isn't really a classic value type. Yes, the comparison with `None` feels more succinct but in any other situation you are likely to ignore the message part and only consider the `ErrorCode`. This implies to me that the message isn't really part of the object's identity *per se* but is being bundled along with the code for convenience (as C# does not support multiple return values). This leads me back to believing the real mistake is in the conditional statement in the caller and should have been this one which we covered earlier that compares the `Code` property directly:

```
if (result.Code != ErrorCode.None)
```

Software archaeology

At this point, I've pretty much exhausted my analysis based on the current state of the callers and implementation and so I decided to do a spot of software archaeology [2] and trawl the version control logs to see what else it could tell me about the history of this code.

It turns out it has very little to say. The `Error` class only has one revision which means that whatever the initial implementation was, that sufficed. We can't tell if the design changed at all in response to its use in the client code but given the nature of the bug I'd posit that any tweaking that did occur during testing would have happened in the caller instead.

Switching to the call site changes, we also see only a single check-in. Looking at the diff for the file where the bug occurred we see that there are five modified call sites – three of them directly compare the `Code` property and the other two use the buggy comparison. What's also noticeable is that the format of the error handling code is slightly different in the former and latter cases – the code is logically the same but spaced out differently.

If the original author copy-and-pasted the error block, I'd expect them all to look the same, which makes me wonder if they were, to start with, but during testing when the silly mistake was fixed the formatting was cleaned up too. This hypothesis gains a little more ground when we consider that the three call sites that work are in code paths which are exercised by a smoke test which is usually run by developers before check-in. In contrast the other two sites are in a code path which is very rarely exercised and has to be done manually, if anyone remembers.

Epilogue

Luckily this piece of code was authored by someone who was still present in the team (although away at the time this showed up, hence the somewhat over-engineered analysis). Consequently I have been able to find out which of my various hypotheses, if any, were correct.

Turnabout is Fair Play

Baron M is still game for a wager.

Wihy, you look chilled to the bone Sir R-----! Come, sit by the hearth and warm yourself whilst I fetch you a medicinal glass of brandy.

To your very good health sir! Will you join me in a wager whilst you recover?

Good show!

I propose a game that I learned upon the banks of the river Styx whilst my fellow travellers and I were waiting for the ferry. This being the third time that I had died, I was quite accustomed to the appalling service quality of the Hadean public transport system and so was most appreciative of a little sport to pass the time.

When the ferry eventually arrived, the ferryman, a cantankerous old curmudgeon I must say, set to beating me upon my back with one of his oars for making him wait whilst I finished a game. Needless to say, I should not tolerate such treatment at the hands of a common matelot and so I snatched his other oar and engaged him in a duel!

He was remarkably spry given his advanced years and put up quite the fight, I can tell you! Nevertheless, he was ultimately no match for my skill at arms and I bested him with a particularly well placed blow to his head. Realising that I was now faced with yet another interminable wait for the next ferry, I decided to take the long trek back to the land of the living instead.

But I digress! I must tell you the rules of that game!

Here, I have laid out twenty five coins face up in five rows of five coins apiece and have turned one of them face down. For a price of one coin if, by turning pairs of horizontally or vertically neighbouring heads over to tails, you can turn every coin to tails then you shall have two coins as your prize! What say you sir?

When I told that loathsome student, who it appears that despite my very best efforts I am quite incapable of eluding, about this game he paid it no heed whatsoever but instead commenced to harping on about his and his fellows' mutual hatred of board games. I suppose that one shouldn't be altogether surprised to find that those whose wits would be sorely tested by snakes and ladders eschew such sport entirely, but let us not concern ourselves with the likes of them! Come, take another glass and weigh up your chances! ■

Baron M

Courtesy of www.thusspakeak.com

BARON M

In the service of the Russian military the Baron has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.

Speak Up! (Part 2) (continued)

It turns out that the type was never intended to be used as a value type but just a simple bucket for holding the two bits of error information, i.e. a way to return multiple values in C#. The three call sites that directly compared the `Code` property with the enumeration value `None` was the intended pattern of use and was fixed when the smoke test failed. The other two were missed as they didn't break any existing tests.

Although the smoke test (which is also run automatically on deployment) picked up the bug, it also goes to show that it does not help in conveying the author's intentions. One of the roles of lower level tests (e.g. unit tests) along with verifying our expectations is to document how our code is intended to be used and illustrates the scenarios we've considered. In this instance, we had to piece that together purely from the way the code is used in production, which was inconsistent. The cost of rewriting would have been minimal this time, but in other cases we're not so lucky. It also shows that code which is not tested at all automatically will be forgotten about. Once again, the broken feature didn't matter this time but next time it might.

The only other niggle I had was why the two missed code paths didn't get picked up at commit time. The pre-commit review [3] is an excellent point to jog our memory about such matters, particularly if the change didn't go smoothly, i.e. we wrote some code and we didn't get it right first time. To me the difference in code formatting between the working and failing cases was another little sign that something was out of place. However, we need to be careful what our tools are showing us because the patch-style diff made this apparent, whereas the full context diff showed that actually the code was in keeping with its surroundings. So the author didn't really miss anything after all.

Somewhat ironically, all the above analysis actually turned out to be a waste of time. A static analysis tool highlighted some dead code which, when investigated, further revealed that the only property used was the `None` one, for the happy path. All other code paths resulted in an exception and therefore the `Error` class was in fact redundant and the intervening methods could all be changed to return nothing (i.e. `void`). Once done, all the error handling blocks where the bug had shown up could be deleted too.

Did I say waste of time? As Ken Thompson once said, "One of my most productive days was throwing away 1,000 lines of code." This was only 50 lines of code but it still feels like time well spent. ■

References

[1] Some objects are more equal than others, Roger Orr & Steve Love, ACCU 2011 Conference, https://accu.org/content/conf2011/Steve-Love-Roger-Orr-equals.pdf

[2] In The Toolbox – Software Archaeology, *C Vu 26-1*, http://www.chrisoldwood.com/articles/in-the-toolbox-software-archaeology.html

[3] In The Toolbox – Commit Checklist, *C Vu 28-5*, https://accu.org/index.php/journals/2306

How Do You Read?

Sven Rosvall shares his perspective on electronic publications.

There have been discussions/opinions/thoughts about electronically distributed club magazines in ACCU and many other organizations I am a member of. These discussions have sometimes been calm and sometimes heated. Some of my clubs have gone entirely electronic for cost reasons and to save volunteer labour in printing/folding/enveloping/sending their magazines. Even though I am a computer professional, it took me a while to get comfortable with electronic magazines. I'd like to share my journey with others to make their lives easier. Note that I am not going to suggest that ACCU goes entirely electronic, even though this is an organization with technologically advanced members. Hopefully the text below will also help others become comfortable with electronic magazines or help them decide whether they want to stick with printed magazines.

First we need to look at the 'reading experience'. (A term I first heard when I worked for Amazon when they introduced the Kindle reader.) How do we read and what do we want to get out of the reading? The 'how' starts with 'where', as in where do we read? I prefer to read my magazines at bedtime when I relax after a good day. Another place is on the crowded tram on my way to work when I have nothing else to do than to wait for arrival. Sometimes reading happens in the armchair, half-listening to the TV and my wife. Sitting at a desk in front of a big screen does not appeal to me as it is not a relaxing place. A desk is the only place you can have your 10 kg PC. In bed, you can use your 3 kg laptop, but the keyboard is in the way and you have to sit up in the bed. The laptop works fine in the armchair though. Instead I prefer to use my ½ kg tablet in bed. It is light enough that I can hold it in my arms. A thick book is just as heavy but more awkward to hold. The best device for the bed is an e-book reader like the Kindle which only weighs 200g. On the tram, I mostly use the Kindle thanks to its size and weight. Sometimes I use the phone but it is a bit fiddly. I don't like using the tablet on the tram as I have to stand most times and there is a greater risk that I will drop it when people squeeze by to get on or off the tram. The Kindle is slightly smaller but the lower weight means it is easier hold on to and move aside to let people pass. And the Kindle doesn't break that easily if I drop it. An A4 magazine just doesn't work on this tram as there is not enough space between the passengers or in my pocket.

A note on the Kindle is that my version is not suitable for some magazines as it doesn't do colours. The Kindle Fire has colour but I would call it a tablet in this context.

It has been said many times that a computer screen is hard on the eyes compared to a printed book. This is certainly true for the old CRT (Cathode Ray Tube) screens where the pixels were blurred. An old or badly adjusted CRT is certainly difficult to read and the eyes get tired very quickly. The flat LCD screens found in laptops are much easier on the eyes. Modern ones are even better. I just got a new laptop with twice the resolution found in HD TV screens. Each letter is nicely rounded just like in print. The electronic ink screens found in electronic readers like a Kindle also have very good resolution and these screens reflect light rather than using back-lighting. This makes the electronic ink comparable to printed paper.

SVEN ROSVALL

Sven has been on the software scene a while and been exposed to many markets and technology shifts. Living in Dublin, he enjoys his spare time with his bike, his model railway and the odd pint of the black stuff. He may be brought back to reality by sending a message to sven@rosvall.ie

Different kinds of texts require different reading techniques. A fiction text is usually flowing from the start to the end. You just need to remember where you were between reading sessions. For a book, you just put your bookmark between the pages you have just read. E-book readers remember where you were. PDF documents, like the electronic version of *CVu* and *Overload*, are trickier to keep track of. One trick is to never close the document and never shutdown the computer/tablet. This is not always practical. Adobe Reader allows you to add sticky notes. Create one such note and write "Read to here" or something similar and save the PDF document. Later, when you open it again you look up that note and continue reading. Adobe Reader can also highlight text just as you would do on paper with a highlighting pen.

Non-fiction text usually has references to previous or following sections. When you read a printed book or magazine it is easy to hold a thumb where you are and flick through the pages to find the section you need. It is easy to go back to your thumb again. It is trickier to browse through an electronic document in the same way. Electronic documents on the other hand sometimes have links that refer to the sections with the explanation I needed. Most electronic readers have a search feature to aid searching.

Another challenge when reading electronic magazines is which ones you have not yet read. I put printed magazines in a pile on my bedside table. When I have finished reading one, I archive it on my book shelf and continue with the next magazine in the pile. Electronic magazines require a different organization. They arrive as PDF documents attached to emails. Initially I tried to use my inbox to keep track of which magazines I hadn't read yet. This filled up my inbox as the PDF documents can be quite large, and the magazine emails were soon hidden in the large amounts of emails in my inbox that I hadn't cleaned up. Instead, I save each magazine file from the email as they arrive in a folder for each organization I am a member of. This is my archive, which is also backed up regularly. Once archived, I copy the magazine file to a separate TO READ folder. When I have finished one magazine I simply remove it from this folder. As I use several devices for reading my magazines, I synchronize this folder with my tablet and my phone. Personally I use Microsoft's OneNote for this synchronization. You may use cloud based solutions like Dropbox or use a local file server if you have set one up. Using a synchronized folder between different devices helps you remember which magazines to read as you now see the same list of magazines whatever device you use. When you have finished one, you just delete it and it will disappear from the other devices too. The Kindle does not have any good tools for such synchronization so I have to copy the file manually and remember to remove the magazine from the Kindle and the shared folder. Of course, it is a lot easier if you only use one device for reading the magazines.

If you are reading on a laptop, tablet or a smartphone you have another challenge: the distractions of social media. Usually these social apps are running in the background and show a notification now and then. Of course, you want to see the cute kitten and respond to your friend. And then you find it hard to get back to your reading and catch up where you were.

This is my way of reading electronic magazines. Others may have other systems that suit them better. Now that I found my way, I actually prefer electronic versions as they don't take any space in my bookshelf. And I find it easier to find old articles in my electronic bookshelf. ■

A Class What I Wrote

Paul Grenyer reduces the boilerplate with simple abstraction.

When I was a member of ACCU, their regular publications always appealed for people to write articles for them. There were a few suggested topics, but the one which stuck in my mind was to write about a class you'd written. I often used to wonder about doing this, but it's quite difficult as I rarely wrote a class which was stand alone enough to write about, without having to write about a load of other classes too. Maybe that's a symptom of a design which is not loosely coupled, but I'll leave that for a late night discussion with Kevlin Henney.

Today I wrote such a class, and was very pleased with it as it reduced a lot code which was repeated in a number of methods down to a single line of code – it even manages a resource! The code I started with is in Listing 1.

It's Java. It gets an output stream from a `HttpServletResponse` instance passed into a Spring MVC controller method, writes some JSON to it, flushes the buffer and cleans up. If there's an error and an exception is thrown, the output stream is still cleaned up, the exception is handled and logged. All reasonably simple and straightforward.

With the class that I wrote, it's reduced to:

```
try
{
  new ServletResponseWriter(response)
      .write(JsonTools.toJson(...));
}
catch (ServletResponseWriterException e)
{
  log.warn(e);
}
```

An instance of the class is initialised with the `HttpServletResponse` instance and a single method called to write the JSON to the output stream. If an error occurs and an exception is thrown, it's handled and logged, just as before.

There is far less code to maintain by using the class instead of repeating the original code.

Let's take a look at the class itself, `ServletResponseWriter` in Listing 2.

Let's start at the top and work our way down. The constructor takes a `ServletResponse`, which is an interface implemented by `HttpServletResponse` containing the `getOutputStream` method. The `ServletResponse` is saved within the class as an immutable field.

The first of the write overloads allows the user of the class to write to the output stream using UTF-8 without having to specify it every time. It calls the second overload with the UTF-8 encoding.

The second write overload is much the same as the original code. It gets an output stream from the response and writes the supplied string to it, flushes the buffer and cleans up. If there's an error and an exception is thrown, the output stream is still cleaned up, the exception is handled, translated and re-thrown.

Listing 1

```
try
{
  final OutputStream os =
    response.getOutputStream();
  try
  {
    IOUtils.write(JsonTools.toJson(...), os,
      "UTF-8");
    response.flushBuffer();
  }
  finally
  {
    os.close();
  }
}
catch (IOException e)
{
  log.warn(e);
}
```

Listing 2

```
public class ServletResponseWriter
{
  private static final String UTF8 = "UTF-8";

  private final ServletResponse response;

  public ServletResponseWriter
    (ServletResponse response)
  {
    this.response = response;
  }

  public void write(String data)
  {
    write(data, UTF8);
  }

  public void write(String data, String encoding)
  {
    try
    {
      final OutputStream os =
        response.getOutputStream();
      try
      {
       IOUtils.write(data, os, encoding);
        response.flushBuffer();
      }
      finally
      {
        os.close();
      }
    }
    catch (IOException e)
    {
      throw new ServletResponseWriterException
      (e.getMessage(), e);
   }
  }
}
```

PAUL GRENYER

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com

Kate Gregory: An Interview

Emyr Williams returns with a new interview from the world of programming.

Kate Gregory is a C++ expert, who has been using C++ for nearly four decades. She is the author of a number of books, and is an in-demand speaker who's given talks at the ACCU, CppCon, TechEd and TechDays among many others. She is a Pluralsight author, a Microsoft Regional Director, and a C++ Microsoft Valued Professional, and despite her hectic schedule she still manages to write code every week.

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

I did my undergrad work at the University of Waterloo. I started in the Faculty of Mathematics and they taught us algorithms and Fortran as a first year course. I didn't choose it, but I had to do it. Other such courses followed, and when I transferred to engineering I discovered this was a useful and in-demand skill. I got opportunities to program on my co-op jobs, and it kind of grew from there.

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

I don't actually remember, but it's a good bet it was an assignment 1 in that first year Algorithms/Fortran course. And yes, punch cards were involved. My first program for money was a simulation of the way scale grows inside a pipe – in the piping of steam turbines scale forms in layers that can spall off and cause tremendous damage, so understanding that is an important problem. It led to a published paper for the researcher who hired me, and an award-winning co-op work term report for me. I probably should have demanded credit in the paper.

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

Sleep when the baby sleeps. It's the best advice ever, and I pass it on whenever I can. Second best: the smaller the problem is, the more ridiculous it will be when you finally find it, the harder it is to find. Don't feel bad about that. Laugh when you finally find the one character typo or the wrong kind of bracket or whatever the tiny thing is that has kept you aggravated for hours.

How did you get in to C++? What was it that drew you to the language?

In the late 80s, I needed to write some numerical integration programs for these multiple partial differential equations I was tackling for my PhD work on blood coagulation. Fortran, PL/1, COBOL, MARKIV and the like were just not going to work for me. My partner was doing some C++ at the time and it seemed like it was going to be much better. Turns out, it was! I had experienced the misery that was the Fortran "common block" so I didn't want to use Fortran any more, and the other languages were mostly about manipulating text and records, turning input into output. C++ was a better fit for working with numbers, for implementing an algorithm, for giving me what I needed to show some properties of those equations.

Since 2004, you have been a Microsoft Valued Professional in Visual C++, how did that come about? That must feel pretty awesome?

MVPs are chosen primarily for their generosity. You can be a complete and utter expert on the C++ language or on Microsoft's products, but if you don't share that and help people with those tools, you won't get the award. Mine I believe was triggered by my books, and more recently my activities on Stack Exchange sites, backed up of course by conference speaking. It's nice to have that effort recognized. What I like best about the MVP program is the access it gives us to the team. I can reach just the right person if I have some issue with the Microsoft tools, and get advice or an explanation or "we'll fix that in the next release." Of course, the award certificates look good on my "bookshelf of showing off" as well.

You hold an incredibly busy schedule between speaking at conferences, travelling, and doing Pluralsight courses, how do you keep your skills up to date?

It's part of my job to stay current. If I spend an hour (or an afternoon) swearing at a development environment on a platform I don't normally use, well that counts as work. It's as valuable work as preparing a talk or doing something billable for a client. So is reading long documents about what's new in C++ or trying out a new library someone has released. What's nice for me is that once I've put that learning time in, I can use it in many different ways – as the backbone of a talk, a blog post, to help a client going through the same thing, as part of a course, and so on.

If you were to start your career again now, what would you do differently? Or if you could go back to when you started programming what would you say to yourself?

I came up through a sort of golden age. You had to teach yourself things, or find someone to teach them to you, because there wasn't

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

A Class What I Wrote (continued)

The keen-eyed among you will have noticed that there are two new classes here, not just one. I'm not a fan of Java's checked exceptions. They make maintenance of code more laborious. So I like to catch them, as I have here, and translate them into an appropriately named runtime exception such as `ServletResponseWriterException`.

`ServletResponseWriter` implements the finally for each release pattern of the original code and the common pattern implemented by classes such as Spring's `JDBCTemplate` which wraps it in a reusable class intended to manage resources for you.

Resource management is vital, but the real advantage here is that the code is more concise, more readable and reusable. And, I've had the chance to write about a class I once wrote. ■

a lot of training available. But then again, people didn't demand credentials or challenge your background. If you said you could do something, the general response was to let you go ahead and show that you could. I think I would just reassure myself that my somewhat unusual path was going to work out to be amazing. I really only had a traditional job for two years after I finished my undergrad work. By the time my grad work was done I had a business with my partner, and we have made our own path for three decades now. It's had some ups and downs, but I don't think I would actually change any of it.

If there is such a term, what would an average working day look like for you?

Oh there are most definitely no average days. I have routines when I'm home – swimming in the morning, coffee and email before I get out of bed – but I do so many different kinds of work that it's hard to characterize. I try to react to my moods if I can – some days are better for writing a lot of code, others are better for big picture design whether of code, a course, or something I'm writing, and still others are the days when you have to catch up on emails, phone calls, paperwork, and buying things. Some days I might be elbow-deep in code when it's time for my evening meal and I just keep right on working well past when I should have gone to bed. Other days I stop in the afternoon and for the rest of the day I just do something – anything - that isn't work. It's nice to be able to work according to my own rhythms. I have to be diligent about deadlines and promises, and some days I have to do things that are a little suboptimal because I have no more room to rearrange, but for the most part I do things I like from when I get up until when I go to bed, I do them side by side with my partner (my husband is my business partner) and I get paid for it, so that's a pretty nice life, isn't it?

What would you say is the best book/blog you've read as a developer?

The Mythical Man Month got me thinking about the big picture of managing teams and people, managing projects, instead of just writing code. And it showed me that people can disagree and best practices can change. While I rarely draw on specific facts or quote from it, it changed the way I thought about creating software.

Do you mentor other developers? Or did you ever have a mentor when you started programming?

Yes, I mentor others – I've done so as part of a paid engagement and I occasionally just offer unsolicited advice to those I think need it. People ask me to help them and if I can, I do. That doesn't mean I'm going to write half their application pro bono, but I answer questions and suggest things to learn or try. I've been the happy recipient of a great deal of marvellous advice from friends and peers, folks who were a little further ahead on one aspect of all the huge difficulty that is being a developer, and would tell me things I needed to know or introduce me to the right people. I try to do the same for others as often as I can.

If you do mentor others, how did that come about? Do you do face to face mentoring, or do you do electronic mentoring?

Because I live in the middle of nowhere, most of my advising is not in person. I have had regular Skype calls with those who I am advising, and that works really well. Sometimes people email me their questions, or even message my public Facebook page, but the nice thing about Skype is I can see their screen, or show mine, while we're talking live. That's generally a lot better than email or other kinds of asynchronous messaging.

Then again, some of my most valuable advice has been given in restaurants and pubs. There's no need to be in the same place if I'm explaining C++ syntax or architecture or "good design", but career advice, soft skills things like dealing with difficult people or knowing if you're charging enough for your time – that works better when we're in the same place and relaxed. It's one of the great things about conferences and other in-person get-togethers – a chance to give and get advice, or to listen to other people's advice sessions.

Finally, what advice would you give to someone is looking to start a career as a programmer?

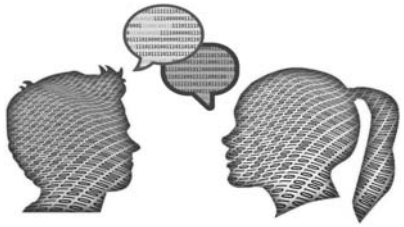
Be prepared to keep learning your whole life. Be prepared to spend a long time learning something, to use it for a while, and then to see it become useless. Don't fight that, move to the next thing. Watch for the big architectural and people lessons that still apply even when you don't work on that platform, in that language, or for that kind of business any more. Hold onto the wisdom you build up, while realizing you still need to learn new knowledge (language syntax, tool use, platform idiosyncrasies) every day.

You can learn from online courses, from working on a project on your own time, on the job if you're lucky, from just trying things and then frantically Googling when they don't work. You can combine dozens of different ways of learning things, getting unstuck when you're stuck, and realizing when to give up and start over. We all feel stupid from time to time, that doesn't mean we really are. (If you're working with the pre-release of something, you may have found a bug – I've done it and most of my friends have too. It isn't always you who's wrong.) And we all have to start over – new languages, new tools, new teams, new platforms – from time to time. If you know how to learn, how to start at something and recognize where you can use things you know from before, how to ask the right questions and how to make sure you don't have to ask the same question twice – you'll be doing very well indeed.

Oh, and sleep when the baby sleeps. Do not forget that. In the larger sense, that advice applies even for people who never raise a baby. There are times in your life when there just isn't time to do everything, so you have to do the most important thing whenever you get the chance. Don't waste time doing the second most important thing if there's a good chance you won't get another opportunity to do the most important thing. When you have a new baby, that means you don't tidy during naps – your most important thing is sleeping and you do it whenever you can. When you're writing software, there's never enough time for everything. If you spend time doing less important things, you may never get to do the most important ones. That's a disaster. Know your priorities and don't skimp on what's most important when there isn't enough time to go around – which is most of the time, to be honest.

Code Critique Competition 103

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



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

Note: If you would rather not have your critique visible online, please inform me. (Email addresses are not publicly visible.)

Last issue's code

I am trying to parse a simple text document by treating it as a list of sentences, each of which is a list of words. I'm getting a stray period when I try to write the document out:

```
-- sample.txt --
```

This is an example.

It contains two sentences.

```
$ parse < sample.txt
This is an example.
It contains two sentences.
.
```

Can you help fix this?

Listing 1contains `parse.cpp`.

Listing 1

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

// pointer type
template<typename t> using ptr =
  std::shared_ptr<t>;

// forward declarations
struct document;
struct sentence;
struct word;
document read_document(std::istream & is);
sentence read_sentence(std::istream & is);
void write_document(std::ostream & os,
  document const & d);
void write_sentence(std::ostream & os,
  sentence const & s);

// A document is a list of sentences
struct document
{
  ptr<sentence> first_sentence;
};
```

Listing 1 (cont'd)

```
// A sentence is a list of words
struct sentence
{
  ptr<sentence> next_sentence;
  ptr<word> first_word;
};
struct word
{
  ptr<word> next_word;
  std::string contents;
};

// read a document a sentence at a time
document read_document(std::istream & is)
{
  sentence head;
  auto next = &head;
  std::string str;
  while (std::getline(is, str, '.'))
  {
    std::istringstream is(str);
    ptr<sentence> s(new sentence(
      read_sentence(is)));
    next->next_sentence = s;
    next = s.get();
  }
  document d;
  d.first_sentence = head.next_sentence;
  return d;
}

// read a sentence a word at a time
sentence read_sentence(std::istream & is)
{
  word head;
  auto next = &head;
  std::string str;
  while (is >> str)
  {
    ptr<word> w(new word{nullptr,str});
    next->next_word = w;
    next = w.get();
  }
  sentence s;
  s.first_word = head.next_word;
  return s;
}

// write document a sentence at a time
void write_document(std::ostream & os,
  document const & d)
{
  for (auto s = d.first_sentence; s;
       s = s->next_sentence)
  {
    write_sentence(os, *s);
  }
}
```


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)

```
// write sentence a word at a time
void write_sentence(std::ostream & os, sentence
const & s)
{
  std::string delim;
  for (auto w = s.first_word; w;
       w = w->next_word)
  {
    std::cout << delim << w->contents;
    delim = ' ';
  }
  std::cout << '.' << std::endl;
}

int main()
{
  document d(read_document(std::cin));
  write_document(std::cout, d);
}
```

Critique

Paul Floyd <paulf@free.fr>

First, let's see what happens when the code is compiled. Firstly clang++:

```
clang++ -g -Wall -Wextra -std=c++11 -stdlib=libc++
-pedantic -o cc102 cc102.cpp
cc102.cpp:89:36: warning: unused parameter 'os' [-
Wunused-parameter]
void write_sentence(std::ostream & os, sentence
const & s)
                                    ^
```

Next, Oracle CC 12.5:

```
CC +w2 -g -std=c++14 -o cc102 cc102.cpp
"cc102.cpp", line 48: Warning: is hides the same
name in an outer scope.
"cc102.cpp", line 88: Warning: os is defined but
not used.
2 Warning(s) detected.
```

Both of these are relatively harmless. The 'os' error can be fixed by streaming to os rather than cout, i.e., change

```
  std::cout << delim << w->contents;
```

to

```
  os << delim << w->contents;
```

In this case, it makes no difference since **write_sentence** is called from **write_document** which is passed **cout**. However, if ever **write_document** were called with another ostream the program would encounter a 'discovered bug' (similar to 'discovered check' in chess).

The second warning can be removed by renaming the second **is** variable, e.g. to **iss**.

Neither of these affect the behaviour of the program as it stands. The issue is due to how **std::getline** works in the following loop:

```
while (std::getline(is, str, '.'))
{
  std::istringstream is(str);
  ptr<sentence> s(new sentence(
    read_sentence(is)));
  next->next_sentence = s;
  next = s.get();
}
```

Basically, **std::getline** has 2 overloads (if I ignore the rvalue reference overloads). The first takes just an **istream** and a string and reads to the next newline (or **eof**). The second adds a character delimiter, and **getline** reads to this delimiter (or **eof**). Here we have the second overload. When it is called the first time, before the call we have the input like this:

```
This is an example.\n
^
```

Stream pointer here

Then the read takes place and we read to the **.** delimiter:

```
This is an example.\n
                   ^
```

Stream pointer here (under newline).

So far so good. Or is it? The stream pointer is not pointing to the null termination at the end of the line but rather the newline character.

On the second call to **getline**, we read **"\nIt contains two sentences"** into **str**. The leading newline gets stripped when it gets streamed into the **str** string variable in **read_sentence** which is breaking the line into words.

After the second call to **getline**, the stream is as follows:

```
It contains two sentences.\n
                           ^
```

Stream pointer again under newline

There is then a third call to **newline**. This just reads the second newline up to **eof**, and this is the source of the spurious extra line in the output.

There are many ways that this could be fixed. One would be to use the **getline** overload without the character delimiter and to then extract the sentence up to the full stop. As a quick and simple solution, I just added an extra call to **getline** to eat the newline characters:

```
while (std::getline(is, str, '.'))
{
  std::istringstream iss(str);
  ptr<sentence> s(new sentence(
    read_sentence(iss)));
  next->next_sentence = s;
  next = s.get();
  std::getline(is, str);
}
```

In terms of design, this code needs a lot more work to be able to handle other punctuation like question and exclamation marks, more than one sentence on a line, and sentences that span several lines.

Jim Segrave <jes@j-e-s.net>

The error in this program is trivial – the **while** condition **std::getline(is, str, '.')** will return **true** when you reach end of file. Adding a simple **if(is.eof()) { break; }** at the top of the loop will cause the **while** loop to terminate without trying to make a sentence from an empty read.

But there are other issues:

A minor one:

The code uses a large number of forward declarations. Rearranging the code so that **struct word** is defined before **struct sentence** and **struct sentence** before **struct document**, **read_sentence()** before **read_document()**, **write_sentence()** before **write_document()** allows all the forward declarations to be removed. If this were a multi-module program, the forward declarations would need to be in a header file, and would have some value. As it's a single file, removing them removes duplication, so changing any of the struct definitions or function signatures has a smaller impact (DRY– don't repeat yourself).

Another minor one:

write_document() and **write_sentence()** are passed an **ostream** parameter. **write_document()** simply passes it on to **write_sentence()**, but it's not actually used there. It should be removed from both functions or the function should be altered to actually use the **ostream**s rather than **cout**.

Most importantly, the author has chosen to build his own singly-linked list. One has to ask 'Why?' While linked lists are good when you need to do

inserts and deletes at random locations within the list, but otherwise they are a performance and memory waste (pointers which aren't needed, lack of contiguous allocation).

This program simply needs to maintain the order of words encountered in sentences and sentences encountered in documents. For that, the **std::vector** class is ideal – it performs well, maintains the insert order and keeps the data contiguous.

If you are going to use a linked list in spite of this, there's already a container type for this purpose which is likely to be implemented better and tested more thoroughly than a one-off version. But as I said, there's no reason to use a linked list here.

Finally, why not use the **make_shared** library function when you want to create an object and get a shared pointer to it? To me at least, the resulting code is clearer.

I attach a version using **vector**s and **make_shared** (with a heavy use of **auto** to save typing and shared pointers for all the robustness they bring with them.

```
#include <iostream>
#include <sstream>
#include <memory>
#include <string>
#include <vector>

using word = std::string;
using word_shp = std::shared_ptr<word>;
using sentence_vec = std::vector<word_shp>;
using sentence_vec_shp =
  std::shared_ptr<sentence_vec>;
using doc_vec = std::vector<sentence_vec_shp>;
using doc_shp = std::shared_ptr<doc_vec>;

// read a sentence a word at a time, return a
// vector of the words in it
sentence_vec_shp read_sentence(
    std::istream & is) {
  std::string str;
  auto  s = std::make_shared<sentence_vec>();
  while (is >> str) {
    s->push_back(std::make_shared<word>(str));
  }
  if(s->size()) {
    return s;
  }
  // if no words found, don't return an empty
  // vector, let it die here
  return nullptr;
}

// read a document a sentence at a time,
// return a vector of the sentences in it
doc_shp read_document(std::istream & is) {
  std::string str;
  auto d = std::make_shared<doc_vec>();
  while (std::getline(is, str, '.')) {
    if(is.eof()) {
      break;
    }
    std::istringstream is(str);
    auto s = read_sentence(is);
    if(s) {
      d->push_back(s);
    }
  }
  if(d->size()) {
    return d;
  }
  // if no sentences found, don't return an
  // empty vector, let it die here
  return nullptr;
}

// write a sentence one word at a time
void write_sentence(
  const sentence_vec_shp & s) {
  std::string delim;
  for (auto wp: *s) {
    std::cout << delim << *wp;
    delim = ' ';
  }
  std::cout << '.' << std::endl;
}

// write document one sentence at a time
void write_document(const doc_shp & d) {
  for (auto s : *d) {
    write_sentence(s);
  }
}

int main() {
  auto dp = read_document(std::cin);
  write_document(dp);
}
```

James Holland <James.Holland@babcockinternational.com>

On first glance, the problem could be anywhere within the student's code. In such a situation it is best to divide the code into roughly two equal parts and to attempt to discover which half contains the bug. This process is repeated until the defect is found. Fortunately, the student's code is already divided into two parts; one that puts information into the linked list and another that reads it out.

An investigation revealed that the unwanted full stop is displayed because there are three sentences in the linked list; the last sentence being empty. It would appear that the linked list is being displayed properly and so it must be the part of the code that assembles the list that is at fault. Therefore, **read_document()** requires further investigation.

Closer inspection shows that the body of the **while** loop within **read_document()** is executing three times. This is despite there being only two sentences in the sample document. I think we are getting close to the root of the problem. It all hinges on what keeps the **while** loop executing.

The controlling clause of the **while** loop consists solely of the function **std::getline()** and so the loop will keep executing for as long as the value returned by **std::getline()** can be evaluated as **true**. **std::getline()** returns its first parameter which, in our case, is of type **std::iostream**. A Boolean function is defined for **std::iostream** that returns **true** if the stream has not failed. The function does not consider the end of file being encountered as a failure. This has the effect that when **std::getline()** is attempting to read the third sentence (that does not exist) no fault is reported (despite **eof** being set) and the loop body is executed for a third time. No characters are read into **str**, with the result that a blank sentence is added to the linked list. This explains why an unwanted full stop appears when the content of the linked list is displayed.

What is needed is for **read_document()**'s **while** loop to stop executing as soon as **eof** is encountered. Possibly the simplest way to achieve this is to modify the **while** loop controlling statement by calling **good()** as shown below.

```
while (std::getline(is, str, '.').good())
```

good() returns **false** when the stream has failed or when **eof** is encountered. This is exactly what is required. The student's code will now behave as expected.

There are some aspects of the use of **std::shared_ptr** that should be drawn to the student's attention. When using a shared pointer to point to

a newly constructed object, it is best to use **std::make_shared<>()** as it has the advantages of exception safety and executes more quickly. I suggest the second line of **read_document()**'s **while** loop should be replaced by

```
auto s(std::make_shared<sentence>(
  sentence(read_sentence(is))));
```

and the first statement of **read_sentence()**'s **while** loop should be replaced by

```
auto w(std::make_shared<word>(
  word{nullptr, str}));
```

It is, however, questionable as to whether **std::shared_ptr** is required in the student's program. **std::unique_ptr** is an alternative that should be considered as it is faster and occupies less memory. Unfortunately, it is not possible to simply replace **std::shared_ptr** by **std::unique_ptr** within the students code, mainly because copying **std::unique_ptr** is not permitted. Some restructuring of the code would be necessary which I have not undertaken. Instead, I propose a different approach.

From reading the student's code, it is clear that extensive use of linked lists is made. While it is an interesting challenge to construct linked lists from first principles, there is no need as the C++ standard library provides such containers ready to use. Doubly linked list have been available since C++98 and singly linked lists since C++11. The advantages of using the library linked lists include the following.

- There is no need to explicitly allocate and release memory.
- There is no need to directly manipulate pointers.
- The lists are generic. They can contain elements of just about any type.
- Programs using library linked lists are simpler to read and write and therefore less error prone.

I have rewritten the student's program to use standard library linked lists. In keeping with the student's approach I have used **std::forward**, a singly linked list.

It is a simple matter to add elements to the front of an **std::forward** list; **push_front()** is provided for that. Unfortunately, that results in the list being populated in the reverse order. The last sentence of the document would be printed first and the last word of a sentence would be printed first. What is needed is for elements to be added to the back of the list. This is not quite so easy as there is no such function as **push_back()** for **std::forward** lists.

It may seem slightly odd that **std::forward** does not provide a function to add an item to the back of the list but this omission results from the desire for **std::forward** to be as memory efficient as possible. All is not lost. We simply have to keep track of where in the list to insert the next element. When there are no elements in the list, **before_begin()** provides the appropriate location. When there are elements in the list, it is always the location of the element at the back of the list that is required. This location is conveniently returned by **insert_after()**; the function used to insert elements at the back of the list. Given this, a simple **while** loop can be constructed that adds the required elements to the **std::forward** lists. I use this technique twice in my version of the program, once in **read_sentence()** and once in **read_document()** as shown below.

```
#include <forward_list>
#include <iostream>
#include <sstream>
#include <string>
using Sentence =
  std::forward_list<std::string>;
using Document = std::forward_list<Sentence>;
Sentence read_sentence(std::istream & is)
{
  Sentence sentences;
  auto position = sentences.before_begin();
  std::string str;
  while (is >> str)
  {
    position =
      sentences.insert_after(position, str);
  }
  return sentences;
}
Document read_document(std::istream & is)
{
  Document document;
  auto position = document.before_begin();
  std::string str;
  while (std::getline(is, str, '.').good())
  {
    std::istringstream is(str);
    position = document.insert_after(position,
      read_sentence(is));
  }
  return document;
}
void write_document(const std::ostream & os,
  const Document & document)
{
  for (const auto & sentence : document)
  {
    std::string delimiter;
    for (const auto & word : sentence)
    {
      std::cout << delimiter << word;
      delimiter = ' ';
    }
    std::cout << '.' << std::endl;
  }
}
int main()
{
  Document d(read_document(std::cin));
  write_document(std::cout, d);
}
```

Commentary

There were a number of different problems with the code presented, and I think that between them the authors of the critiques covered most of the issues.

As both Jim and James noted, the writer of the code has implemented their own singly linked list; there is no need to do this nowadays! Additionally, there is no separation of concerns with the implementation embedded in the code – if a special sort of list were required it would probably be better to write a separate class to do the list management, templatized on the contained type.

One problem that no-one commented on is that the code contains a potential bug when exiting **main**. Naive use of smart pointers to build up complex data structures can cause stack overflow on destruction as the destruction of the tree of objects can consume a large amount of stack. In this case the **document** object **d** is the root object, its destruction results in calling the destructor of **first_sentence**, which then destroys the **next_sentence** member of this object, and so on, recursively down the list of objects.

The bug might be avoided if the compiler is able to perform some tail-call optimisation in the destructor, but this is not possible in all cases.

For example, I found this example crashed for large input when compiled without optimisation with g++ (64-bit) and MSVC (both 32-bit and 64-bit). Enabling optimisation resolved the crash for gcc, and for one of the MSVC builds. This is not desirable behaviour!

So, how might you solve this problem?

In order to resolve the stack overflow you need to write an explicit destructor that iterates through the objects rather than using recursion.

For example, in this case:

```
document::~document()
{
  while (first_sentence)
  {
    first_sentence =
      first_sentence->next_sentence;
  }
}
```

We also ought to make a similar change for the list of **word**s in **sentence**, in case we try processing James Joyce's *Ulysses*!

This is an unfortunate issue with using smart pointers for managing object graphs, especially as debugging the root cause of a stack overflow can be quite hard.

(Herb Sutter's talk at CppCon this year touches on some other options that solve the same sort of problem.)

The winner of CC 102

I was amused to note that while all three critiques found and fixed the problem they used three slightly different techniques. It is surprisingly hard to use C++ standard input correctly and the range of possible solutions makes it less obvious when a given piece of code is correct.

Paul gave a fairly detailed explanation of the presenting problem with the original code, which would hopefully make the problem and its solution clear to the writer of the code.

Each critique went on to cover additional problems; these included

- design issues such as dealing with 'real world' sentences containing punctuation (perhaps more could have been made of this)
- preferring use of **make_shared**
- changing to use **unique_ptr** rather than **shared_ptr**
- replacing the list logic with **forward_list**

I liked James' direction in using **std::forward_list**; this solution has the additional benefit of (silently) solving the stack overflow problem I discuss in my commentary. Hence, by a short head, I have awarded him the prize for this critique.

As always, thank you to all those who entered the competition!

Code critique 103

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

> I am trying to keep track of a set of people's scores at a game and print out the highest scores in order at the end: it seems to work most of the time but occasionally odd things happen...
>
> Can you see what's wrong?

The code – scores.cpp – is in Listing 2.

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://accu.org/index.php/journal). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
#include <functional>
#include <iostream>
#include <map>
#include <sstream>
#include <unordered_map>

// Best scores
std::multimap<int, std::string, std::less<>>
best_scores;

// Map people to their best score so far
std::multimap<int, std::string>::iterator typedef
entry;
std::unordered_map<std::string, entry>
peoples_scores;
entry none;

void add_score(std::string name, int score)
{
  entry& current = peoples_scores[name];
  if (current != none)
  {
     if (score <= current->first)
     {
       return; // retain the best score
     }
     best_scores.erase(current);
  }
  current = best_scores.insert({score, name});
}

void print_scores()
{
   // top down
   for (auto it = best_scores.end();
        it-- != best_scores.begin(); )
   {
      std::cout << it->second << ": "
        << it->first << '\n';
   }
}

int main()
{
  for (;;)
  {
    std::cout << "Enter name and score: ";
    std::string lbufr;
    if (!std::getline(std::cin, lbufr)) break;
    std::string name;
    int score;
    std::istringstream(lbufr)
      >> name >> score;
    add_score(name, score);
  }
  std::cout << "\nBest scores\n";
  print_scores();
}
```

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?

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)

Ubuntu Server Essentials LiveLessons

By Sander van Vugt, published by Prentice Hall, ISBN10: 0-7897-5549-1, ISBN13: 9780789755490

Reviewed by Tim Green

Ubuntu Server Essentials LiveLessons is a series of videos divided into 8 'lessons' totalling 3 hours 45 minutes. The complete download is 615MB in h.264 MP4 format. I have been an Ubuntu Desktop user for about 9 years (Debian and Slackware before that), and have recently started looking into Ubuntu Server. Would I learn anything new? After a brief introduction in Lesson 1, the useful content starts in Lesson 2 with installation. The presenter covers most of the options encountered on a default install plus some detail on LVM (Logical Volume Management), but saves installing a web server until Lesson 8.

Lesson 3 covers some admin tasks, including selection of a text editor. Nano is presented, but vim is used through out for its syntax colour highlighting. If you already love emacs then you know what to do! Lesson 4 covered networking, including how the old 'ifconfig' command has been replaced by 'ip'. ssh is mentioned, as is ufw, the Uncomplicated Fire Wall. Unfortunately the lessons are only shown on an installation in a virtual machine (on a Mac!) and only localhost networking used within. Later on the topic of virtual webservers the presenter mentions needing to talk to your ISP for network addresses and domain name resolutions, so beyond the scope of these lessons.

After installing ssh there is no mention of ssh keys, nor rate limiting attackers with Fail2Ban or similar.

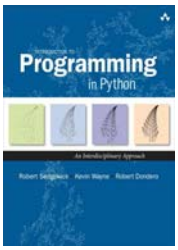
Lesson 5 is all about software management, apt and dpkg. In the video they say "apt" supersedes apt-get and apt-cache in Ubuntu 14.10 and later. Just in the last month Ubuntu have back ported apt to 14.04 and it will install with a standard "apt-get upgrade" (now just "apt upgrade"). A very useful command I learnt was "dpkg -S filename" to find which package installed a file matching that filename.

Disk storage is the topic of Lesson 6. After a brief look at partitions there is a lot of meat on the topic of LVM. It is very tempting when installing to a single (large) disk system to think LVM is not useful, but it certainly comes into its own for separating data from software, and with multiple disks. As someone who has never chosen LVM the benefits could have been better explained to me.

In Lesson 7 the presenter covers top, ps, nice and kill for processes. For services there is upstart from Ubuntu 14.04 and earlier, and systemd in 14.10+. In Lesson 8 we install Apache and learn about how to configure it for virtual hosts. The presenter's clever trick is to add the new domain names to /etc/hosts so local testing will show it working before configuring external DNS. The presenter also recommends "apt-get install apache2-doc" which installs all the documentation in http://localhost/manual/

Overall the videos are good at getting you going with a web server. If you want to follow along judicious use of pause button will be needed to read the commands and output in the screen-cap sections. The presenter Sander van Vugt is from the Netherlands and with his accent, and quite flat delivery style, I needed to concentrate hard to avoid the beeping distractions of email, Twitter and the rest of the Internet. At a list price of $150, I am not sure of the replay value of the videos for individuals, but in a corporate environment this is a lot cheaper than having van Vugt attend to speak in person.

Introduction to Programming in Python: An Interdisciplinary Approach



By Robert Sedgewick, Kevin Wayne and Robert Dondero, ISBN: 97801340769430, published in 2015, 771 pages

This book has been independently reviewed by two people.

Reviewed by Barry Nichols

This book is very ambitious, attempting to teach programming to first year undergraduates of any scientific discipline. It not only aims to introduce basic programming concepts, but also recursion, object oriented programming and the fundamentals of algorithms and data structures. I believe this would be challenging for a computer science student within an introductory course let alone a student from another subject, especially if using this book for self study in addition to their own workload. However, this book generally explains these concepts well and contains some interesting problems from a range of application areas, including mathematics; physics and biology.

A problem encountered by most beginner Python programmers is that Python 2 is still widely used alongside Python 3. The two versions having significant differences, such as print being changed to a function and the behaviour of the division operator on two integers. The authors go out of their way to ensure this book is compatible with both versions. Unfortunately, the approach they take is to write their own "std" modules which the student must download and use to accomplish even basic tasks, without being clearly instructed to do so. Even the hello world program requires the module "stdio" for the function "writeln". This could have been avoided by using the __future__ module. This standard Python 2 module allows the use of the print function and can change the division operator to be equivalent to Python 3. The standard Python functions, i.e. without using the authors modules, are not discussed. This continues throughout the book, even where the module functions simply call built-in functions.

If this book was written using Python libraries; built-in data types; and, where required, common external libraries such as NumPy and SciPy it would be a very good introduction to programming in Python. As it is this book is a good introduction to several programming topics in the authors non-standard version of Python, after which the student must invest

more time and effort to learn the actual Python programming language.

Reviewed by Jim Segrave

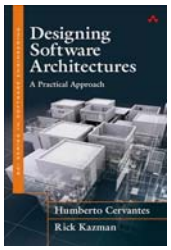
The title is a bit misleading – this is not a book for learning to program in Python, it is on overview of the sorts of subjects university computer science courses will cover. It does teach some things about Python programming as part of showing how problems can be solved, but it is hardly complete in that area; for example list comprehensions are never mentioned, dictionaries appear only in passing in the last chapters of the book.

As might be expected given the authors, there is a heavy emphasis on algorithms and the use of recursion in particular. It is assumed the reader is a science or engineering student, the vast majority of the exercises deal with mathematical concepts that occur in such an environment. However it is not a text on design and analysis of algorithms, it does not delve deeply into the mathematics behind analysing time and memory usage.

The book does put a heavy emphasis on good programming practices – tasks should be broken down into functions which have single purpose, programmers should pay attention to and measure how programs perform as a function of input size, advice on how to choose an API when building modules as part of a larger program, etc. There are a huge number of interesting challenges in the exercises for each chapter which can keep the reader busy for quite some time.

In summary, this is not a book you will keep as a reference on your bookshelf, you won't be going back to it over and over again. From a learning Python perspective, this is not what you will want. From a comp-sci perspective, I found too much of the treatment to be insufficient in depth. It's a good overview of what there is to learn in the field, but doesn't treat any of the topics sufficiently on its own.

Designing Software Architecture. A practical approach



Reviewed by Marco Dinacci

Designing Software Architecture eschews formal architectural models such as the IEEE standards or the "4+1" view model and provides instead a more practical (according to the authors) view of architecting a software system. The authors refer to the design process they adopted as ADD Attribute-Driven-Design (ADD).

The book is essentially split into four main sections.

The first part is a definition of architectural design and is mostly aimed in my opinion to new software architects. The second part goes into the description of the ADD architectural design process. The third, probably the most useful part of the book, focuses on use cases. This is where one can actually see what ADD involves in practice. The last part are appendixes on design useful for reference.

The first part of ADD consists on capturing functional and non functional requirements using what the authors call Use Case Model, Quality Attribute Scenarios and Constraints.

There is a refreshing focus on iterations in this methodology, which is something usually missing from other architecture methodologies but still enough rigour so that an inexperienced architect won't feel lost during the process.

The first part of the design process is about capturing the overall software architecture in text and diagrams and recording the design decisions. The last bit of the first iteration is about cross-referencing the use case model with the chosen architecture to make sure all use cases are addressed.

The second iteration is about selecting technologies and explaining the rational for selecting one or another. The following iterations goes more in detail into aspects specific to the chosen architecture.

Overall I recommend this book to all software architects. ADD may not be the most suitable methodology for every project but it's another tool at our disposal. The use cases are well described and serve as a reference while attempting to use this system in the real world.

Test-Driving JavaScript Applications

By Venkat Subramaniam, ISBN-13: 978-1680501742

Reviewed by Paul Grenyer

I wanted to start this review simply with "Wow! Just wow!", but that's not really going to cut it. It's true to say that when I first learned that there was going to be a book published called "Test-Driving JavaScript Applications" I was sure it was going to be the book I had been waiting for since at least late 2007 when I was forced to write JavaScript in production for the first time. It's publication date was pushed back and back, so it really felt like I was being made to wait. However, I wasn't disappointed and this book was everything I hoped it would be and more.

We all know JavaScript is evil, right? Why is it evil? It's the lack of a decent type system, the forgiving nature of the compilers and an inability to write meaningful unit tests, especially for the UI (User Interface). It's difficult to do a huge amount about the first two points, but now JavaScript can be meaningfully unit tested, even in the UI context, with Karma, Mocha and Chai. Test coverage can be measured with Istanbul and System Tests (referred to by Subramanian as Integration Tests - this is my one bugbear with the book) written with Protractor. All of this is described in Test-Driving Java Applications.

I think it's important to read all of part 1, Creating Automated Tests. The chapters cover everything you need to know to get started writing unit tests for both server side code and UI code, how to test asynchronous code (very important in JavaScript) and how to replace dependencies with test doubles such as fakes, stubs and spies. It's all demonstrated with a completely test first approach with excellent commentary about how this leads to good design.

I cherry picked from part 2, Real-World Automation Testing. I was only really interested in how to write automated tests for the DOM and JQuery and how to write 'Integration' tests. Other chapters included how to write tests for Node.js, Express and two versions of AngularJS. The DOM and JQuery chapter was excellent showing me exactly how to take advantage of test doubles to write fully tested JavaScript without having to fire up a browser, resulting in something I can make immediate use of.

The Integrate and Test End-to-End chapter, which describes how to use Protractor, was almost enough to encourage me to abandon Java (Selenium) for System Tests and move to JavaScript. However, while looking at the latest version of Selenium, there are some other things I want to investigate first.

The final chapter, Test-Drive Your Apps is the equivalent of Pink Floyd playing Run Like Hell at the end after Comfortably Numb. It's still good, but is really there to help you wind down from the climax and could just as easily have been omitted, but it would feel a bit odd if it was.

If there was one more thing I could get from this book it would be how to send test and coverage results to SonarQube.

If you want to use JavaScript, intend to use JavaScript or are forced to use JavaScript, get this book and automated the testing of your JavaScript.

View from the Chair

Bob Schmidt
chair@accu.org

Today is Thanksgiving in the U.S. That means a four-day holiday for some of us. But there's no rest for the weary, as I have to get this article written, and then I have to finish my Hour of Code [1] presentation. I committed to talking to approximately 50 ten- to 13-year-old students (5th through 8th grade) on December 5th, which is just fantastic because I have absolutely no recent experience talking to any large group, and less experience talking to children. If I survive the experience, I'll provide you with an update in the next *CVu*.

Diversity statement

The Diversity Statement was given final approval during our bi-monthly committee meeting held on November 19th, and has been published on the ACCU website [2]. Thank you to all who participated in its drafting and refinement.

ACCU Conference 2017

The 2017 ACCU conference is scheduled for Wednesday, April 26th through Saturday, April 29th, with pre-conference tutorials on Tuesday the 25th [3]. The schedule for the conference will be announced on January 20th. The conference once again will be held at the Marriott City Centre in Bristol, UK. Members of ACCU get a discount on the conference – if you are not already a member, please join the organization and take advantage of the benefits of membership. Go to https://accu.org/index.php/joining to get started.

Just announced (well, 'just' being while this was being written) – Herb Sutter will be giving one of the keynote presentations at ACCU 2017.

Code of conduct

The conference committee has published a Code of Conduct on the ACCU website [4]. This Code of Conduct will apply to all events run under the aegis of ACCU and all its local groups, and will evolve as necessary. If you are running an ACCU event, presenting at an event, or planning to attend an event, please be aware of your rights and responsibilities under the code.

Election of officers

The ACCU Annual General Meeting will be held in conjunction with the conference, on Saturday April 29th. An important part of the AGM is the election of officers to serve on the committee for the coming year [5]. Serving on the committee is a great way to give back to the organization. To find out more about being a committee member, or if you are interested in serving on the committee for the 2017–2018 term, please contact me or one of the other committee members.

Conference web site improvements

The new conference session proposal submission system has been up and running since mid-November. After a bit of a shaky start, the major problems were sorted and the system made available to prospective speakers. Conference Chair Russel Winder reports the system is not pretty, but is functional. He is hoping that this time next year it will be a lot smoother, and prettier.

Committee spotlight

Matt Jones has been a member of ACCU since 2005, and our Membership Secretary since 2014 (taking over from Mick Brooks after his seven year term). Matt started programming with BBC Basic, and then learnt C on a summer job between school and VI form. He did a four year mixed electronics and software degree at Southampton, but took all the software options. He's been programming professionally for over 20 years, becoming freelance 2 years ago. He works mainly in C++ but occasionally reverts to pure C when required, and sometimes Python. Most of his work has been on large real time and/or embedded systems. When bit bashing he misses the freedom of working at high level, but when the tables are turned, he eventually gets bored with the abstracted nature of high level code and yearns to get dirty with hardware again. At the moment he is writing GUI code in C++11 and Qt 5.7 – about as advanced as he's ever been!

The duties of the membership secretary are varied. Usually the signup/pay/renew cycle is handled electronically by the web site, and automatic credit card payments are handled via Worldpay. Corporate members who require invoices, and members who pay by standing order or cheque, have to be dealt with by hand which includes poring over the bank statement once a month to tie payments back to members. Once a month, the address lists for the magazine have to be prepared and sent to the company who print and distribute them for us. Once the paper copies are out, the excess from the print run is delivered to Matt's house, where he stores them for up to a year. This is in order to supply back issues on request, for members who have missed copies, and marketing hand outs (e.g. for local groups, and at the conference). Postal delivery problems (i.e. 'Return to sender') also end up in his post box, at which point the member is contacted and the problem resolved. Matt knows his local post office quite well. Official duties include preparing a yearly report for the AGM, regular reports to the committee, and being a signatory for cheques.

Call for volunteers

I have been 'advertising' for an auditor (to replace me) for the past several issues, without success. It has been suggested that perhaps the duties of the auditor were unknown and mysterious – a description of the position does not exist on our website or in the constitution – and this could be the reason no one has stepped forward. Well, I can rectify that.

The role of an ACCU auditor is to conduct a high-level review of the accounts which have been prepared by a professional firm of Accountants using information supplied by our Treasurer. The auditor is expected to question any unusual transactions, ask for evidence to support transactions, verify assets, and ensure that the accounts provide an accurate statement of ACCU's financial situation. A detailed investigation into every single transaction is not usually required, nor is the auditor expected to duplicate the work of the Accountants.

The work is not particularly difficult; it's a bit like balancing a cheque register. Familiarity with, and access to, Microsoft Excel is helpful as that is how the records are kept.

There should be two auditors at all times, serving two-year stints, with one auditor being replaced each year. By design, auditors are separate from, and independent of, the committee. Auditors report their findings to the committee. If all is in order the Chair signs the audited financial statement on behalf of the organization.

We still have several other open positions:

- The ACCU web site uses Xaraya, a PHP framework that has been moribund for the last 4 years at least, and a replacement is overdue.
- We are hoping to recruit someone to assist Martin Moene with the web site sys admin duties.

Please contact me if you are interested in any of these positions.

Local groups

Nigel Lester, our Local Groups Coordinator, reports that local groups Meetup membership has grown by five percent in the past two months, and by approximately 50% since the beginning of the year. To those of you who have joined a local Meetup group this year – welcome. If you have not already done so please consider becoming members. Membership entitles you to print copies of our journals, discounts to the annual conference, and the ability to participate in the direction the organization takes through voting and committee activities [6].

Finally, with magazine production schedules being what they are, by the time you read this our calendars will have rolled over to 2017. On

behalf of your ACCU committee I wish you a happy, safe, healthy and prosperous New Year.

References

[1] Hour of Code https://hourofcode.com

[2] ACCU Diversity Statement https://accu.org/index.php/aboutus/diversity_statement

[3] ACCU 2017 https://conference.accu.org/site/index.html

[4] ACCU Code of Conduct https://conference.accu.org/site/stories/coc_code_of_conduct.html

[5] I know that we just had a Special General Meeting and election. The realities of *CVu*'s publishing schedule means that this issue may be a little early to be thinking about the AGM election, but the next issue may be leaving it too late.

[6] Yes, I am aware that this magazine is distributed and available online to members only; however, copies are made available to local groups, so I am hopeful that this message will be read by prospective new members.

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner.

To find out more about Intel products please contact us:

020 8733 7101 | **enquiries@qbssoftware.com**
www.qbssoftware.com/parallelstudio

qbs SOFTWARE