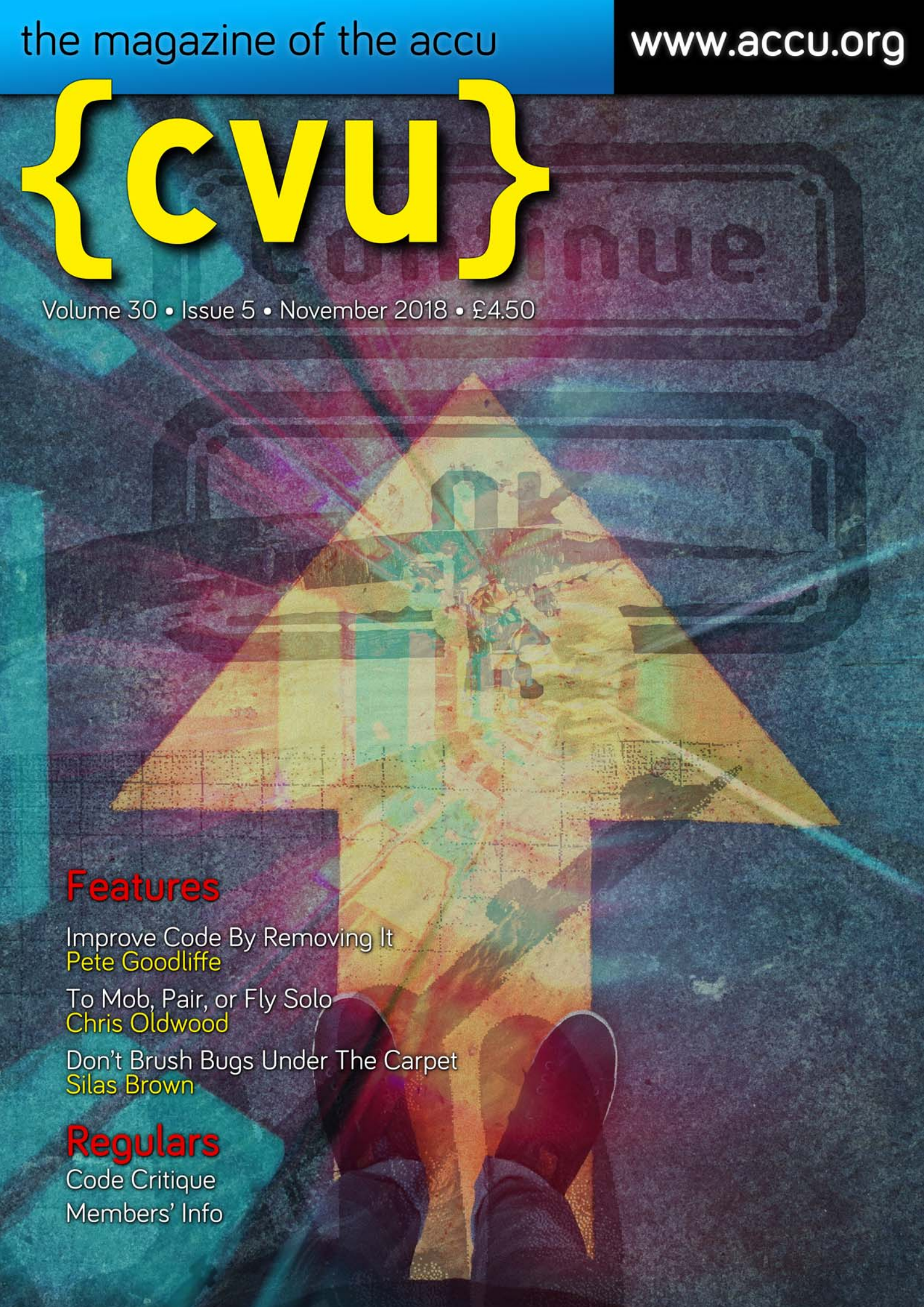
the magazine of the accu

www.accu.org

{cvu}

Volume 30 • Issue 5 • November 2018 • £4.50



Features

Regulars

{cvu}
Volume 30 Issue 5
November 2018
ISSN 1354-3164
www.accu.org

Precious Metals

In days of old, when money was more than just numbers on computers, it was common for people receiving coins in exchange for other services or goods to assess the quality of the coins before deciding their worth. Coin clipping was common, since the materials used were routinely silver and gold, so a sliver of a gold coin might have been enough 'small change', but the purity of the coin's composition is harder to determine than by just looking at it.

In mediaeval England, precious metals were analyzed in small earthenware pots (usually by melting). This would determine if a sample had been debased, i.e. mixed with some material with a much lower value. This practice of *assaying* was commonplace, but more interesting here is that the pot itself was called a 'test'.

It would certainly not be a method used for assessing coins on a trade-by-trade basis, but instead on a sample. The modern-day English verb 'test' is derived from this usage. I'm struck by the parallel with software testing; precious metals were effectively put into a sort of crucible to determine their actual value. Only if the material passed 'the test' would a merchant accept it, based on an assayer's assessment.

My analogy breaks down in the face of currency standardisation and fiat money, which is the practice of deliberately making coins and other 'money' out of materials which have no intrinsic value. The monetary value is ascribed by a promise – usually made by a government – and the widespread belief that the promise is good. Part of the reason that fiat money became so popular in England was that currency debasement was so widely practised by the government in Tudor times.

I'm pretty sure there's another analogy hiding in there, but un-picking it will have to wait for another time!

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 30.6: 1st December 2018
C Vu 31.1: 1st February 2019

Overload 148: 1st January 2019
Overload 149: 1st March 2019

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

COPYRIGHTS AND TRADE MARKS

Improve Code by Removing It

Pete Goodliffe takes a scalpel to unnecessary code.

We ascribe beauty to that which is simple;
which has no superfluous parts;
which exactly answers its end...
~ Ralph Waldo Emerson

Less is more. It's a trite maxim, but sometimes it really is true.

Some of the most exciting improvements I remember making to code involved *removing* vast chunks of it. Let me tell you, it's a good feeling.

A war story: As an Agile software development team, we'd been following the hallowed eXtreme Programming tenets, including *YAGNI*. That is, *You Aren't Gonna Need It*: a caution to not write unnecessary code – even code you *think* is going to be needed in future versions. Don't write it now if you don't need it now. Wait until you do have a genuine need.

This sounds like eminently sensible advice. And we'd all bought in to it.

But human nature being what it is, we fell short in a few places. At one point, I observed that the product was taking too long to execute certain tasks – simple tasks that should have been near instantaneous. This was because they had been over-implemented, festooned with extra bells and whistles that were not required, and littered with hooks for later extension. None of these things were being used, but at the time they each had seemed sensible additions.

So I simplified the code, improved the product performance, and reduced the level of global code entropy by simply removing all of the offending 'features' from the codebase. Helpfully, my unit tests told me that I hadn't broken anything else during the operation. A simple and thoroughly satisfying experience.

> You can improve a system by adding new code. You can also improve a system by removing code.

Code indulgence

So why did all that unnecessary code get written? Why did one programmer feel the need to write extra code, and how did it get past review or the pairing process?

It was almost certainly the programmers' indulging their own personal vices. Something like:

- It was a fun bit of extra code, and the programmer wanted to write it. *(Hint: Write code because it adds value, not because it amuses you, or you'd enjoy trying to write it.)*
- Someone thought it was a feature that would be needed in the future, so decided to code it now, whilst they thought about it. *(Hint: That isn't YAGNI. If you don't need it right now, don't write it right now.)*
- But it was only a small thing; not a massive 'extra' feature. It was easier to just implement it now, rather than go back to the customer to see whether it was really required. *(Hint: It always takes longer to write and to maintain extra code. And the customer is actually quite approachable. A small extra bit of code snowballs over time to a large piece of work that needs maintenance.)*
- The programmer invented extra requirements that were not documented in the story that justified the extra feature. The requirement was actually bogus. *(Hint: Programmers do not set system requirements; the customer does.)*

Now, we had a well-understood lean development process, very good developers, and procedural checks in place to avoid this kind of thing.

And unnecessary extra code still snuck in.

That's quite a surprise, isn't it?

It's not bad, it's inevitable

Even if you can avoid adding unnecessary *new* features, dead pieces of code will still spring up naturally during your software development. Don't be embarrassed about it! They come from a number of unavoidable accidental sources, including:

- Features are removed from an application's user interface, but the backend support code is left in. It's never called again. Instant code necrosis. Often it's not removed "because we might need it in the future, and leaving it there isn't going to hurt anyone".
- Data types or classes that are no longer being used tend to stay put in the project. It's not easy to tell that you're removing the last reference to a class when working in a separate part of the project. You can also render *parts* of a class obsolete: for example, reworking methods so a member variable is no longer needed.
- Legacy product features are *rarely* removed. Even if your users no longer want them and will never use them again, removing product features never looks good. It would put a dent in the awesome list of tick-box features. So we incur perpetual product testing overhead for features that will never be used again.
- The maintenance of code over its lifetime causes sections of a function to not be executable. Loops may never iterate because code added above them negates an invariant, or conditional code blocks are never entered. The older a codebase gets, the more of this we see. C helpfully provides the preprocessor as a rich mechanism for writing non-executable spaghetti.
- Wizard-generated UI code inserts hooks that are frequently never used. If a developer accidentally double-clicks on a control, the wizard adds backend code, but the programmer never goes anywhere near the implementation. It's more work to remove these kinds of autogenerated code blocks than to simply ignore them and pretend that they don't exist.
- Many function return values are never used. We all know that it's morally reprehensible to ignore a function's error code, and we would *never* do that, would we? But many functions are written to *do something* and return a result that someone *might* find useful. Or might not. It's not an error code, just a small factoid. Why go through extra effort to calculate the return value, and write tests for it, if no one ever uses it?
- Much 'debug' code is necrotic. A lot of support code is not needed once the initial implementation has been completed. It is unsightly scaffolding that hides the beautiful architecture underneath. It's not unusual to see reams of inactive diagnostic printouts and invariant checks, testing hook points, and the like, that will never be used again. They clutter up the code and make maintenance harder.

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

Don't Brush Bugs Under The Carpet

Silas S.Brown presents an allegorical lesson on bug reports.

Once upon a time, there was a doctor who was so wonderfully good at curing the sick that he couldn't be replaced. Unfortunately, he also had a psychopathic irrational hatred for red clothes. If any patient came in wearing red clothes, he would kill the patient instead of curing them.

Question: what should be done about this?

- Option A: get rid of the doctor and try to find a replacement, no matter how long that takes and how many patients go without treatment in the meantime.
- Option B: try to fix the doctor. Get a competent analyst to help him figure out why he goes crazy at red clothes and address it so he doesn't do it again.
- Option C: post a guard outside the doctor's office. The guard will not let anyone in if they're wearing red clothing.
- Option D: post a warning sign outside the doctor's office, hoping that wearers of red clothing will read the sign and change their clothes before entering.
- Option E: do nothing. I personally will not wear red clothes, and I'm the only patient that matters, so problem solved.

Unfortunately, I have twice now encountered European software users who seem to think the answer is E. In both cases, the issue was that they can cause a program to crash (only on the Microsoft Windows platform) by giving it input that involves an accented letter. Once they realised that it was the accented letter causing the crash, they simply said "Oh yes, we shouldn't expect English software to cope with our accents, we'll just change our input and you don't have to fix anything." They weren't even willing to show me what the original input was (and I couldn't reproduce the bug by typing accented letters myself, so I needed a copy of their input for investigation and I wasn't going to get it).

Mapping the doctor analogy onto software:

- **Option A** (replace the doctor) might be best if there is a good alternative implementation available, or if you have reason to believe that you really can do better with a rewrite, but remember that a rewrite can introduce new bugs, and if it's you who wrote the first version (and not so long ago) then how do you know you've improved enough not to make the same mistakes a second time?
- **Option B** (fix the doctor) is the best option if you really can fix it. But it's hard to analyse what's going on if all you are getting is vague reports from users who are not willing to share their data with you. My second user was on the point of sharing his data, but he didn't want to share the original, so he tried to create a 'cleaned-up, simplified-down' version, and in the process he discovered that it was a problem with an accented letter, so he said: "Don't worry, I've solved it" and I haven't heard any more from him. Frustrating!
- **Option C** (guard the doctor) is probably the best option if you cannot fix your code: simply add guard code around it that restricts the input. Unfortunately it's a bit of a compromise solution if the guard code has to 'over-block': if I have a bug that only manifests itself in some very specific situation that involves a particular accented letter, then I could say 'insist on ASCII only', but that would be putting extra restrictions on my users that are not entirely necessary. I already know my code works perfectly well with many non-ASCII situations; it just goes wrong in some particular case that people are refusing to tell me.
- **Option D** (warn about the doctor) is a very poor option. People won't read the sign. It's like putting a comment on a function that says 'may crash if X' and nobody ever reads the comment. Perhaps a warning would be better than nothing, but not very much better than nothing.
- **Option E** (do nothing) might be OK for a personal private doctor (some quick program you wrote just to use yourself on your own computer), as saying 'but I would never give it that input anyway' could be true if you are the only user. But even then I'd still be worried about my future self and at least write a simple guard, even if it's nothing more than an assert or two, just to get a good 'fail: you need to take a look at this' message if I ever forget it had that limitation. And I certainly don't think production code that is used in cancer-research labs around the world should take Option E, and I'm worried to see my European users thinking otherwise.

I'm probably 'preaching to the converted', telling this to developers. It is our users who need to learn the value of being willing to give us all the information we need to properly diagnose a bug, instead of giving up as soon as they personally have a workaround. I can understand settling for a personal workaround if the developer is not interested in fixing it properly, but I wouldn't want to do that when dealing with a developer who IS interested. Perhaps they're just not used to having developers take an interest in them.

Anyway, I hope that this 'doctor story' can at least help somebody explain to a user the need to get things properly fixed, at least if that user is fluent enough in your language to be able to listen to your explanation. ■

SILAS S. BROWN

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

Improve Code by Removing It (continued)

So what?

Does this really matter? Surely we should just accept that dead code is inevitable, and not worry about it too much if the project still works. What's the cost of unnecessary code?

- It is undeniable that unnecessary code, like any other code, requires maintenance over time. It costs time and money.
- Extra code also makes it harder to learn the project, and requires extra understanding and navigating.
- Classes with one million methods that may, or may not, be used are impenetrable and only encourage sloppy use rather than careful programming.
- Even if you buy the fastest machine money can buy, and the best compiler toolchain, dead code will slow down your builds, making you less productive.
- It is harder to refactor, simplify, or optimise your program when it is bogged down by zombie code.

Dead code won't kill you, but it will make your life harder than it needs to be. It gets in the way and slows you down. ■

To Mob, Pair, or Fly Solo

Chris Oldwood compares different collaboration practices.

...If asked, most programmers would probably say they preferred to work alone in a place where they wouldn't be disturbed by other people. The ideas expressed in the preceding paragraph are possibly the most formidable barrier to improved programming that we shall encounter.

~ Gerry Weinberg, *The Psychology of Computer Programming*, 1971

I first started hearing about pair programming around the mid 2000s and became more aware of it after joining the ACCU around the same time. However it was not something that immediately made sense to me, after all, by then I had been programming professionally for well over a decade and unprofessionally for another decade before that. By then I felt I could probably describe myself as a fairly experienced programmer, not necessarily an expert [1], but definitely no longer a junior.

As I sat in the bar very late one night (or more likely one early morning) at the Barcelo Hotel in Oxford (where the ACCU Conference was being held back then) I quizzed my fellow ACCU colleagues what use they thought pair programming had for the more experienced programmer. I could see that the practice would have benefits at the more junior end of the spectrum but I couldn't really grasp what was in it for those of us with a few more miles on the clock.

A decade later, I'm finally beginning to formulate an answer and, probably unsurprisingly, it's not at all what I thought it was about…

Heavy scepticism

Many people, when they hear about any kind of new practice that they don't really understand, fill in the gaps by making assumptions and then often come to the wrong conclusions and therefore potentially dismiss it out of hand. Even if they're willing to give it a jolly good go, unless they really know what problems it aims to solve they can evaluate it poorly and then still dismiss it as useless (to them).

Certainly, for the first couple of years after first hearing about pairing, I fell into this trap. I generally worked in environments where the other team members were all very experienced, the systems were generally pretty mature and the culture was (as I perceived) focused on individual performance rather than what the team as a whole was producing. Any 'pairing' that was done was of the more traditional sense, such as talking through ideas on a whiteboard, jumping in to help out when a production incident occurred or helping a teammate debug an issue. In the latter case, the 'pairing' usually ended once the issue was identified; there was no continued collaboration to see the entire task right through to the end together or to go beyond the immediate need for learning. (In retrospect the goal seemed to be to minimise disruption of the more experience programmers rather than maximise the transfer of knowledge to the less experienced ones.)

Naturally my perception, based on very little knowledge, was really a misconception as I assumed the point was to be more productive, where the term 'productive' could be interpreted very loosely as writing more lines of code per day. The mental model I was working with was that of a Symmetric Multiprocessor Architecture (SMP) – still somewhat expensive even back then – where adding each additional person was like adding another CPU. Everyone is probably aware of the old adage 'two heads are better than one' and if you equate programming to a largely cerebral activity I don't think this is an altogether unexpected hypothesis to derive from the smattering of (dis)information floating around at the time.

Consequently, knowing that an extra CPU in a SMP architecture generally only adds another 60% of a full CPU's performance (a rule of thumb back then due to communication overhead) I was not convinced that the 'whole' could ever be greater than the sum of its parts, at least not for a pair of seasoned programmers. I was definitely open to the idea that for more junior programmers, there was a lot to be gained because you have so much to learn when you're starting out about the tools, processes and problem domain.

And what about mob programming – the idea that if two heads are better than one, why shouldn't three, ten or a hundred be even better? Now things are starting to get crazy and are verging on the faux-science peddled by homeopathists! Applying the multi-processor analogy once again, there are clearly diminishing returns here and there is simply no way ten people working on ten problems sequentially can be as effective as ten people working on ten different problems concurrently?

Something you might have noticed when you do work together with other people while tackling a problem where there is a time pressure element, such as a production incident, you tend to pick up every little mistake the person doing the typing, talking, drawing, etc. makes. It's almost as if every single keystroke matters and one of the benefits you can provide is the ability to catch any mistakes before they've even hit the Enter key. But surely it's not just about saving keystrokes either?

Collaboration benefits

While possibly useful, correcting typing mistakes is not our modus operandi or a major selling point for pairing or mobbing – we generally have other tools, like compilers and IDEs, which are far more reliable at that sort of thing. No, while there are some short term benefits, the emphasis is playing the long game which most software development falls into.

Shared understanding

Those earlier opinions of mine, I now realise, are the thoughts of a brain addled by too many years working in the enterprise where a fundamental assumption is that overall efficiency is high if every single person is working to capacity. The flaw in the assumption is that what really matters is how much work each person does per day. One thing this naïve view fails to take into account is the lost opportunities due to delivering everything at the end instead of when ready. But it also misses the lost productivity due to the extra overhead of handovers and context switches as everyone is too busy to help each other out [2]. (This is one area where my multi-processor analogy does appear to hold some water.)

While these are very real concerns, what I began to realise was that the real benefits of multi-programmer collaboration are not about short-term wins but about long-term sustained delivery. What slows a team down over time is the rise in complexity as the product gains more and more functionality. A single programmer can write and maintain an implementation of Fizz Buzz, but once we're talking in the thousands of lines of code, the complexity becomes much harder to manage. Over time as people, tools and concepts come-and-go the landscape changes and without a common understanding of where we were, where we are now and where we're heading, that rate of divergence accelerates and becomes ever harder to correct. Before long the code-base

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 and can be easily distracted via gort@cix.co.uk or @chrisoldwood

resembles Frankenstein's monster and the cognitive load required to implement even small changes safely becomes insurmountable.

This isn't just an issue for someone new to a code-base either, although this is definitely one area where the practice of pairing shines because it allows the new joiner to make contributions to the team from day one. Instead of waiting a few weeks (or months) for someone to get up to speed with the code-base and keeping their work quarantined while it's reviewed and merged, they can use their existing knowledge and skills of the technical domain from the get-go to suggest solutions and improvements while their longer-serving partner(s) provide the conceptual integrity [3]. The knowledge can then be drip-fed real-time and prioritised based on what you actually need to get the job done day-to-day.

At the more trivial end of the spectrum are the small things that we generally shouldn't sweat [4], like coding guidelines. What really matter are the design principles that may not always be quite so obvious from just browsing through the code. For example, parsing the response from an HTTP endpoint may be done more laboriously than is otherwise necessary because certain types of errors trigger slightly different error recovery behaviours. Similarly certain conventions may be used because they enable legacy code to be more easily tested such as the use of `internal`, instead of `private` in C#.

Obviously, documentation and tests can provide additional context but they are rarely used to express the kinds of decisions just described. Architecture Decision Records [5] are a great way to cover big picture stuff but smaller patterns or idioms, e.g. `Value Object` and `Optional<T>`, would probably be considered out of scope and may just assumed to be common knowledge. (In my experience these are anything but common.) With nobody and nothing to guide you, you'll have to use a spot of software archaeology [6] to work out where on the timeline you are and what you should be refactoring towards, which all takes up time and does not provide a definitive answer.

Shared ownership

It isn't just the understanding of the products' architecture and design that needs sharing but also the ownership of the source code itself and therefore the right for anyone to change it. In the past, when single people worked on specific features, the code they wrote to implement it would, whether consciously or not, be associated with that person. They naturally became the expert and therefore the gatekeeper of that module and therefore future changes to it would likely be funnelled through that person too – it's the most efficient approach, right?

Conway's Law teaches us that the structure of our code will reflect the structure of the communication paths in the organisation and so if the paths are blocked we will route around them which in code terms might mean duplication for example or inappropriate use of inheritance as a way of side-stepping the 'problem'. The Open/Closed Principle was intended as a technique for extensibility of a design, not a way to get around bug fixes needed in the existing code.

When every line of code stems from the contributions of more than one person, any notion of single ownership vanishes as it's clearly nonsensical to attribute the code to the person who happens to be on the check-in or the one at the keyboard at the time. When everyone gets involved in pretty much everything the chances of 'silos' developing are minimal. (Eric Raymond [7] opines that a non-territorial approach to software development, which Weinberg much earlier termed 'egoless programming', is likely a significant factor in the speed and quality of open source projects.)

Learning

For those already largely au fait with the organisation's product and processes there is still a lot to be gained from collaborating with both their peers and juniors alike. Although we may try really hard not to utter those infamous words "we've always done it this way", there is nothing like a mind unhindered by decades of baggage to challenge why you're not doing something in a different way.

There is simply too much going on in the world of software development to keep up with all the new languages, tools, products, etc. Much as I try to flick through the release notes for the latest update to my favourite editor I quickly forget some of the little shortcuts and I need to be reminded to get it into my muscle memory. Just watching someone else navigate around the code-base can be enlightening both about the tool you're using and the code-base itself. I once remember watching some teammates continually build the entire solution in Visual Studio 7 instead of installing the Fast Solution Build plugin that ensured you only built what was changed, thereby saving a fair amount of time. Just telling people these tips via email or IM isn't enough, especially if it's in the middle of something important; they likely already have enough yaks to shave.

Learning, of course, means more than just what keys to press or tools to use, useful though they are. We also need earlier feedback on the way we express ourselves in code. There is a humorous take on code reviews which regularly does the rounds on Twitter that goes:

> Code reviews: 10 lines of code = 10 issues, 500 lines of code = 'looks fine'.

By the time the code is written, there is already a strong desire to push it through and get it out there rather than hold it up until it's corrected. Like it or not, we often have an emotional investment which makes us reluctant to change too much – there are sunk costs. The best time to provide feedback is the moment it happens as you're associating it with the trigger which makes the chances of the more desirable behaviour occurring much stronger in future. It also means spending less time on formal reviewing and rework and therefore faster delivery in the long run (for the same level of quality).

Wisdom of crowds

One of the quotes (from Émile Chartier) that I was introduced to by Kevlin Henney is:

> Nothing is more dangerous than an idea, when it's the only one we have.

While I would like to think that I am capable of having multiple ideas for every problem, it simply isn't true. Even at the lowest level there are many little decisions we make every day about the names of classes, interfaces, methods and variables. It's not just their names either; even the decision of how to partition the logic – inline vs free functions vs classes etc. – is something which different programmers would take a view on.

Hal Abelson famously said that programs must be written for people to read and only incidentally for machines to execute. What you or I might consider readable, however, may not be so obvious to a future maintainer. It's not just the names of things here, the layout and structure matter too, and I don't mean tabs or spaces, I mean how the code is organised within namespaces and assemblies and then how those are arranged on disk within the solution.

A good example of the wisdom of crowds was brought home to me once during a mobbing session. We were doing some refactoring after adding a new feature and a bunch of tests started failing. It wasn't immediately obvious what was going on even during a quick debug of one of the broken tests. I had a good idea where the problem was but one of my colleagues suggested we go looking somewhere completely differently. The crowd agreed with them instead of me, which is lucky, because we got to the root cause pretty quickly. If we had gone down my route we could easily have lost a couple of hours searching in the wrong place. Even the intuition of very experienced programmers can be wildly wrong.

It's fun!

Five years ago I had done virtually no 'production' pairing at all. I knew I was going to enjoy it though after attending a meet-up hosted by Jon Jagger [8]. Writing code in a group where we didn't know each other and had different backgrounds and skills was both a scary and exhilarating

experience. Taking it to such an extreme meant that the diversity aspect of the group really shone through along with the social nature. (You could argue that I did a fair bit of pairing in my teens when I went round my mate's house as naturally we only had one computer back then, but I wouldn't classify that period as one of 'egoless programming' [9].)

Not long after, I ended my current contract and jumped ship to a company where pairing was often the norm, even the main interview was an exercise that involved pairing on a simple kata (which makes a lot of sense). Since then the majority of my professional time has been spent programming in groups of two or more, mostly as a pair, and now when I do find myself working on my own I really miss having that input from other people. Much as I enjoy listening to music I find a spot of banter when working can be equally enjoyable and often lead to tangential conversations that are actually relevant.

In a recent 'Afterwood' [10], I questioned why there weren't more programming partnerships along the lines we traditionally see in writing partnerships such as for sitcoms. Over my 25 years as a professional programmer, there are a handful or so of people that I've worked closely with for a significant amount of time that I'd work with again without question because I felt we complemented each other in ways that meant we did great work together. We don't necessarily share the same taste in music, films, books, food, text editor, brace placement, etc. which I'd suggest is A Good Thing as it ensures there is always a topic up for 'a healthy debate' when the code itself is lacking contention. Maybe we're just not ready yet to settle down and pair with just one partner…

by working together, you are naturally reviewing and reworking as you go

This continual need to justify what we write is what I believe makes it a fun and productive technique. The aim is produce the simplest solution that can solve the problem at hand which means that nearly everything becomes a negotiation – you can't do something just because you feel like it. This causes us to continually reflect on what we're doing and therefore helps to ensure our decisions are conscious ones. The outcome is invariably more satisfying because the journey has been more adventurous.

One colleague of mine has suggested making a set of 'Programming Luminary' style Top Trump cards for those moments when you need to lighten the mood and back up your argument with 'a big hitter'. For example you might decide your side of the argument needs a quote from 'Martin Fowler' whereas your partner, nay opponent, might choose to counter with something from 'Uncle Bob'. The point being that there are many notable people whose ideas shape our vocabulary and thinking but we need to avoid falling into the trap of programming dogma.

How many cooks?

With all that said, not every task requires a legion of programmers working on it and quite frankly it can be an intense experience. If you've ever done any full day workshops you'll know how draining it can be (assuming it was one you enjoyed and actively participated in). There are calls out there to do more formal research into where the tipping points are but the sections below give my entirely unscientific approach to when I've found each of these approaches beneficial.

Mobbing

The (somewhat unfortunate) term 'mob programming' is a relatively new one and it's something which is gaining more ground as working in pairs may just not be valuable enough in some circumstances. I've heard about teams that work entirely in mobs, some even as a 'free flowing mob' where people dip in and out of the mob during the day. Where this has happened to me by accident I've not noticed any significant side-effects from the coming and going.

Personally I've only worked in a large mob a dozen or so times, mostly where the whole team was engaged on a single problem. (We generally commandeered a meeting room for the entire day so we could all comfortably share one laptop projected onto a large screen.) The driver for forming the mob has always been 'shared understanding'. In one instance it was setting up the build pipeline for a new project where we all had slightly different skills but wanted to formulate the pipeline together as a group. We needed to choose some tools and approaches and rather than keep stopping and starting to discuss options we decided to just work on the task together and trash out the initial skeleton over a number of days. This helped set the scene for the eventual pairing afterwards as we all had a better idea of how we wanted to work.

Other times when programming as a team has proved useful has been when you need to tackle a new behaviour within a system which introduces a fundamental design principle. For example when working with a document oriented database that lacks atomicity, except at the document level, you need to be careful how you handle multi-document changes. It is essential that all developers in the team understand these core concepts if changes to the system in the persistence area are to be done safely in the future as proving correctness or testing for race conditions is hard. Security is another aspect of system design that needs to be permeated across the entire team and doing it as a group has proved well worthwhile in establishing common patterns that are easy to follow.

Naturally there are some problems with programming in a large group, although I will hasten to add that the problems I've experienced are usually outside the team. The long term benefits of such an approach are still under scrutiny and seeing the entire team working on a single problem can make some managers very nervous. Whenever we did it the board only showed a single item of work in progress and someone would always ask: "When do you think we'll be able to start working in parallel again?" Explaining the consequences of everyone not being on the same page for critical design decisions certainly goes a long way, but you may still be left repeating yourself as you make the case for the apparent (short-term) drop in productivity.

Pairing

While the whole team working together on a new epic is useful for setting the tone I've found it less useful once you start rattling through the individual features. (Oren Eini uses this notion of Concepts & Features [11] where concepts are the 'framework' which underpin the individual features.) For example when you start adding authentication and authorisation to an API you might pick an initial endpoint and work as a mob on to iron out the basic design and then switch to pairs to apply it across the rest of the API. The first part is pure design work which cannot be parallelised whereas the implementation likely can be because most of the difficult questions should have already been addressed.

While you can go all out and spread the load right across the team you are only delaying delivery of that work by storing up the code reviews and rework for later, and forcing each other to context switch to pick it up. It's not uncommon to see a 'review' column added to the task board with an inevitable WIP limit added to slow the team down and ensure that they are critiquing each other's work. Pairing largely takes that burden away, as by working together, you are naturally reviewing and reworking as you go along thereby keeping complexity under control and avoiding sunken costs.

It's all too easy when working alone to get carried away, either with a spot of refactoring or scope creep, and so having that Jiminy Cricket working with you keeps you honest all the time. The converse is also true though and they are there to ensure that the work is carried through to completion – that the TDD states of red and green really are followed by any necessary refactoring.

Just to be perfectly clear they are not there to be your nanny or lackey but an equal partner in the delivery of a story no matter how much or little experience either of you has.

Going solo

Given what's been said about the usefulness of working with one or more people it's questionable whether there is every really a time when working by yourself is ever appropriate? I think there is.

There are some tasks which really don't demand the attention to detail that product features do because the risks are so low or they are less valuable and being done for other reasons. For example updating background documentation or applying a simple fix to a script that came up during some investigation can easily be done by oneself. Working as a group can be intense and so one way to take a break and grab a little space is to go and do something else alone. (Of course if you work in an open plan office you're rarely entirely alone.)

We all have a variety of administrative duties to perform during the week as well as catching up on any emails, background conversations on IM, timesheets to complete, meeting rooms to book, etc. All these tasks can be done in isolation and fitted in around the group sessions. In fact I'd say that one further advantage of group working is that you get fewer interruptions as it's easy to avoid these kinds of distractions and people generally don't bother disrupting an entire group unless it really is urgent.

One definite downside of paring or mobbing is that course corrections can often happen so quickly because of the fast feedback loop that you can easily miss the deeper learning opportunities. While it may not feel like it at the time but being stuck on a problem and then finding a solution all on your own is a deeply rewarding experience. This gets watered down a lot when you're in a group and replaced by a different kind of satisfaction, one of collective achievement. Sometimes the group might take the 'safest' route and you're still not convinced or quite sure and so a little spike on your own can be a good way to close the loop. We shouldn't feel as though we're giving up our own freedom to explore by working with other people as we are still individuals with our own thoughts and opinions.

Ebbs and flows

Like everything in life one size never fits all – we never spend our entire time collaborating in only one way. In my experience the team works together (or apart) depending on the nature of the current workload. A common pattern is that the team works as a mob on any feature where there is a clear need for shared understanding, such as when key design decisions need to be made. Once the groundwork is done the team might then be equally comfortable working in smaller groups, nominally pairs, for the majority of the feature work. Finally when there are trivial tasks or other quests which are intentionally personal then the team members act more autonomously, although still with any necessary checks and balances.

Consequently the team will likely ebb and flow over time, coming together as one for a while, then working in smaller groups with occasional pockets of time being individuals and then back around the loop again. During this natural flow there may also be more spontaneous larger collaborations as support issues arise or something unexpected turns up that needs a realignment of the team's understanding.

Aside from all the other benefits of breaking the work down into small units, it makes it much easier for a team to continually move around and work with different people. Even being able to mob on a problem for a few hours can really help share the knowledge around. The morning huddle or stand-up has proved to be incredibly useful here as not only do you have an opportunity to reprioritise the backlog but it gives a natural point for team members to decide on who to pair or mob with. If a large part of the team has been mobbing on a single problem, perhaps for a few days, this also provides a natural checkpoint to question whether a 'mob' is still the right approach or whether the returns are beginning to diminish. This really aids transparency and shows that the team is behaving diligently.

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?

Epilogue

It is interesting how often the argument that programming is a solitary act and most programmers are introverts comes up on the social networks. It's also interesting to see the stats about how many projects fail because they didn't deliver what the customer really wanted. One has to wonder if those two are in any way correlated...

In my experience there are very few true introverts, often they are the product of working in an environment that moulded them that way. But with the right attitude from the team and the organisation they soon realise that working collaboratively does not mean having someone nit picking at every little mistake they make but having another pair shoulders to stand on so that they can see further and ultimately achieve more. ■

References

[1] The Downs and Ups of Being an ACCU Member, *C Vu* 25-1, Chris Oldwood http://www.chrisoldwood.com/articles/the-downs-and-ups.html

[2] Be Available, Not Busy, *C Vu* 29-1, Chris Oldwood, http://www.chrisoldwood.com/articles/be-available-not-busy.html

[3] Conceptual Integrity, C2, http://wiki.c2.com/?ConceptualIntegrity

[4] Chapter 0, Don't sweat the small stuff, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Andrei Alexandrescu & Herb Sutter, ISBN: 0-321-11358-6

[5] Documenting Architecture Decisions, Michael Nygard, http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions

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

[7] *The Cathedral and the Bazaar*, Eric S. Raymond, 1999, ISBN 1-565-92724-9.

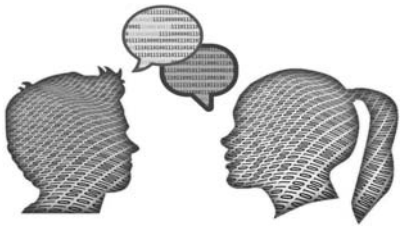
[8] ACCU London, September 2010, Jon Jagger's Coding Dojo, reviewed by Chris Oldwood, http://www.chrisoldwood.com/articles/accu-london-september-2010.html

[9] *The Psychology of Computer Programming*, Gerald M. Weinberg, 1971, ISBN 0-442-29264-3

[10] Afterwood, *Overload* 135, Chris Oldwood, https://accu.org/index.php/articles/2298

[11] Application structure: Concepts & Features, Oren Eini, https://ayende.com/blog/3895/application-structure-concepts-features

Code Critique Competition 114



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

This is a very short code sample but nonetheless an interesting problem.

The writer is using a variadic template to pass multiple arguments to the **least** function, which uses **std::sort** to find the smallest input number. They report the code used to work in 32-bit on a couple of compilers, but it's not reliable when used in more modern 64-bit projects.

Can you identify the problem(s), and suggest any fixes (or alternative approaches)?

The code is in Listing 1.

Listing 1

```
#include <algorithm>
#include <iostream>

// find the least value in the arguments
template <typename T, typename... Ts>
T least(T first, Ts... rest)
{
  std::sort(&first,
    &first + 1 + sizeof...(rest));
  return first;
}
int main()
{
  std::cout << "least(1): "
    << least(1) << std::endl;
  std::cout << "least(1,2,3,4): "
    << least(1,2,3,4) << std::endl;
  std::cout << "least(10,9,8,7,6,5,4,3,2,1): "
    << least(10,9,8,7,6,5,4,3,2,1)
    << std::endl;
}
```

Critiques

Gennaro Prota <gennaro.prota@gmail.com>

Of course, the problem with the code is that it treats distinct variables (function parameters) as if it were accessing elements of the same array. It simply invokes undefined behaviour.

IMHO, the easiest thing to do is to leverage the **min_element()** algorithm and use **std::initializer_list<>**:

```
#include <algorithm>
#include <initializer_list>
#include <iostream>

// find the least value
template <typename T>
T least(std::initializer_list<T> list)
{
  // assume list is not empty; it would be nice
  // to be able to do the following:
  // static_assert(list.size() != 0, "");
  return *std::min_element(list.begin(),
  list.end());
}
int main()
{
  std::cout << "least({1}): "
    << least({1}) << std::endl;
  std::cout << "least({1,2,3,4}): "
    << least({1,2,3,4}) << std::endl;
  std::cout << "least({10,9,8,7,6,5,4,3,2,1}): "
    << least({10,9,8,7,6,5,4,3,2,1})
    << std::endl;
}
```

Stewart Becker <stewart@sibecker.co.uk>

So, first off – kudos for wanting to use an **<algorithm>** rather than rolling your own. This is good practice. Unfortunately, that's about all that can be said in praise of this code. There are three main issues with it:

1. Implementation-defined behaviour
2. Undefined behaviour
3. Choice of algorithm

The key approach used by the function is using the addresses of function arguments as iterators to a standard algorithm. While pointers can be used as iterators, we do need to make sure they form a valid iterator range. The assumption seems to be that the function arguments will be contiguous in memory so as to behave like an array, meaning that we can use pointer arithmetic to determine the 'end', and incrementing the address of the first argument will give the addresses the others in order. Indeed, it seems all the major compilers and calling conventions involve pushing arguments onto the stack in reverse order, so this effect is to be expected. However, this is a convention and implementation-defined. Furthermore, if a function is inlined then all bets are off regarding the relative memory locations of the arguments.

What I suspect is happening is that in the 32-bit world, an **int** is the same width as a stack element so the pointer arithmetic trick worked. However, in the 64-bit world **int**s are only half as wide, so alignment issues come into play. The pointer increment will move forward only in multiples of 32 bits rather than to the next value which is 64 bits away. Trying to force the pointer arithmetic to work in 64-bit might make the trick work again, but I wouldn't want to count on it. It would be better if we could program to the standard alone, rather than relying on implementation details.

As written, the function exhibits undefined behaviour when passed different types. As written, the **Ts...** pack can expand to any list of types. We could, for example, pass in an **int**, a **double** and an object type, e.g. **std::string**. The result would still compile, but at runtime it would perform pointer arithmetic based only on the first type, and dereference

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

that memory location as type `T`, not the `Ts...` types. Even if we know the types are of the same size this is still undefined behaviour. If we are lucky, `sizeof(first)` will be no greater than the average of `sizeof(rest)...` . We then 'only' have to worry about the issues of pointers being dereferenced as the wrong type, and that some values passed lie beyond the end iterator. We're still deep in UB territory, but we should just return a wrong result, at least if we're working with native types like `int`. With object type then we could very easily get an object in an invalid state, which will cause very hard to debug problems later on, but that's a secondary effect. The worst case scenario is if `sizeof(first)` is large compared to the other arguments. We would then be asking `std::sort` to operate on memory locations we don't actually have the right to use. Given that `std::sort` is a modifying algorithm this means we are overwriting arbitrary memory – a disaster waiting to happen!

We can make a start on both issues at once by changing the function signature. If we want a contiguous memory layout with all arguments of the same type, we can do so by using a collection type that does just that. This is the definition of an array, so let's start by using one – in the guise of `std::array` because we want to be modern. Also, we can guarantee check at compile-time when it is empty, which other collections (in particular the usual first choice `std::vector`) cannot detect. One thing the old code did not have to worry about was finding the least of an empty collection. We'll also use actual begin and end iterators instead of pointers:

```
template<typename T, size_t N>
T least(std::array<T, N> array)
{
  static_assert(N > 0, "Empty array");
  std::sort(array.begin(), array.end());
  return *(array.begin());
}
```

Oh dear! Now our calls don't compile. This is because they are passing many arguments as integers, instead of one array. We'll need to modify them to pass as one argument. Instead of `least(1,2,3,4)` we'll add curly braces round the arguments and call `least({1,2,3,4})`.

Nope, that still doesn't compile. The problem is that `{1,2,3,4}` isn't a typed C++ value, therefore the compiler can't deduce its type. However, there C++ is one type that will accept such syntax: `std::initializer_list`. But now we need a runtime check for emptiness. Again we'll be modern and use ADL to lookup the begin and end iterators, just in case we change the type again (hint: we will):

```
template<typename T>
T least(std::initializer_list<T> list)
{
  using std::begin;
  using std::end;
  auto first = begin(list);
  auto last = end(list);
  assert(first != last);
  std::sort(first, last);
  return *first;
}
```

WHAT NOW, COMPILER?! We experience the joy of compile errors longer than our codebase that are the signature of programming with templates. It seems that `std::sort` won't work with `std::initializer_list`. The key error message is `error: assignment of read-only location '* __first'` (gcc). In English, it means that `std::sort` is trying to manipulate the values inside the list, but `std::initializer_list` only provides read-only iterators, even when we passed it by (non-const) value to the least function. Surely it must be possible to get the smallest element from a list without needing to re-order it?

This brings me to the final issue – choice of algorithm. The use of `std::sort` has two problems. As we have just seen, it is a modifying algorithm, so we cannot use it on ranges that are read-only. Furthermore, it is overkill for finding just the least element as it is takes O(n log n). We really should be able to achieve our goal in a single, read-only pass. Indeed we can, by using the algorithm `std::min_element`. Since we're now using a read-only algorithm, we'll take the list by const reference too:

```
template<typename T>
T least(const std::initializer_list<T>& list)
{
  ...
  return *std::min_element(first, last);
}
```

Hooray! It compiles, runs and what's more actually works without relying on implementation details. However it only works on containers of type `std::initializer_list`, which aren't particularly common. It really would be better if we could pass any container, especially the ubiquitous `std::vector` and our first attempt, `std::array`. Also, what it we want to find the greatest element? One of the benefits of `std::min_element` is that we can pass a customised comparison object, including a lambda. We should template the whole range type:

```
template<typename Range>
auto least(Range range)
/* code unchanged */
```

Oh no – our examples don't compile again! The issue is that because brace expressions aren't actually C++ values, they can't be type deduced as `std::initializer_list` objects. We'll need to add an overload with the original signature. Since the code in both will be identical. we'll do this by writing an implementation function and delegating to it in the overloads.

Also, we'll add a customisable but defaulted comparator:

```
template<typename Range, typename Comp>
auto least_impl(Range range, Comp comp)
{
  /* ... only the last line changes */
  return *std::min_element(first, last, comp);
}
template<typename Range,
typename Comp=std::less<>>
auto least(Range range, Comp comp={})
{
  return least_impl(range, comp);
}
template<typename T, typename Comp=std::less<>>
T least(const std::initializer_list<T>& list,
Comp comp={})
{
  return least_impl(list, comp);
}
```

Hmm, that visible `least_impl` function is a bit ugly. If this were a class, we'd probably make it private. One trick used is to move it into a separate namespace, often named 'detail' or 'impl'. But the class approach appeals to me. As for why, when I started this, I forgot about `std::min_element` and tried to roll my own as `std::accumulate(first, last, *first, std::min)`, which wouldn't compile either! The problem is that because `std::min` is a function overload, the compiler couldn't work out which one I meant. Our overloads of `least` mean that trying build algorithm up by passing it as a parameter will suffer the same problem. But if we make it a functor then we can just reference a single unambiguous object. All that changes is how and where we declare the functions, and one final object declaration:

```
class Least {
private:
  template<typename Range,
  typename Comp = std::less<>>
  auto least_impl(Range range, Comp comp) const;
public:
  template<typename Range,
  typename Comp=std::less<>>
  auto operator()(Range range,
  Comp comp={}) const;
  template<typename T,
  typename Comp=std::less<>>
```

```
  T operator()(
  const std::initializer_list<T>& list,
  Comp comp={}) const;
};
static constexpr Least least = {};
```

Joe Wood <joew60@yahoo.com>

By sheer chance I am currently using a 32bit Ubuntu 18.04 distribution. Sometimes it works as expected, sometimes it produces an impossible result, e.g. outside the parameter range, and sometimes it reports a stack smash, so there is obviously something wrong.

Let's start by trying GCC's sanitizers, e.g.

```
g++ -g -O -fsanitize=address \
  -fsanitize=undefined code.cpp
```

Well, running the compiled code produced some interesting information. (The output is a little too long to reproduce here.) The source of the problem is in the call to **std::sort**, and the actual bug is in reading and comparing two iterators deep inside **std::sort**. The offending line is

```
{ return *__it1 < *__it2; }
```

where **__it1** and **__it2** are both iterators.

Why does this fail? We know that sorting an STL conforming container with random access iterators works. Hmm, sorting a valid STL container relies on the data been contiguous [1]. Is our data contiguous? Unfortunately not; to understand why, we have to turn our attention to how parameters are passed to functions (the calling convention) and the Application Binary Interface (ABI). Firstly, both C and C++ appear to put all parameters on the stack starting at the right hand side and moving leftwards. This is a must for functions like **printf**, so it knows where to find the format string and hence process all the arguments. Secondly, I said appears to put the arguments on the stack. As an optimisation, a small number of variables may be passed in the CPU registers. The exact details vary by processor, compiler and ABI.

But what exactly is being passed to **std::sort**? Well let's look at the original code

```
std::sort(&first,
  &first + 1 + sizeof...(rest));
```

- **&first** is clearly the address of the first parameter.
- **&first + 1 + sizeof...(rest)** is the address of the end of the region to be sorted. This assumes that the region is contiguous starting at **&first**. However, due to C++ calling conventions, this may or may not be true.

Hence, the straight forward reason for differing behaviour of least, is that the parameters may not be in contiguous memory, hence **std::sort** may experience a memory access error.

Before, looking at possible solutions lets get another problem into the open. The student wants to find the smallest number of a supplied set by sorting the set and taking the first element of the sorted set. This at best is going to be of order O(NlogN), and we have absolutely no interest in the rest of the sorted set beyond the smallest. Using **std::min** to find the smallest element is of order O(N) and we do not need to carry any additional information around.

To recap, we need to find the smallest element of a set of numbers supplied as parameters to a variadic function, without any memory 'gaps' due to the calling convention. There are two possible approaches, viz.: recursion on the rest parameter or use **std::initializer_list**. **initializer_list** guarantees that its memory is contiguous.

Let's start with the recursive approach, we will call the procedure **least_r**. As with all recursive processes, we need a base case, which for this problem must be one item, as in the sample code. That is just

```
template <typename T>
inline T least_r(T first) {
  // single input, so just return it
  return first;
}
```

That is easy. Now for the recursive case

```
template <typename T, typename... Ts>
T least_r(T first, Ts... rest) {
  // to prevent non-contiguous memory access,
  // we peel the arguments off in turn
  return std::min(first, least_r(rest...));
}
```

The second approach, using **std::initializer_list** is just as simple. We will call this procedure **least_il** for **initializer_list**.

```
template <typename T, typename... Ts>
T least_il(T first, Ts... rest) {
  // to prevent non-contiguous memory access,
  // we must create an initializer list
  const std::initializer_list<T> il {first,
    rest...};
  return std::min(il);
}
```

Both methods work, with the sample inputs and a set of 100 elements, and are acceptable to the **-fsanitize** test above. For added reassurance, both pass Clang's scan-build utility.

There is one more consideration, namely, efficiency. Whilst it is true that to use the **initializer_list** we have to make a complete copy of the input set, it runs quicker than the recursive version, because it does not have to keep calling a smaller version of itself.

Note

[1] Strictly speaking, this is not an explicit requirement, rather the exact requirement is that random access to the container happens in constant time. In practice all containers satisfying the random access criteria are also contiguous. [Ed: this is incorrect – for instance **std::deque** is a counter-example] Being able to 'split' memory on updating an iterator would be a challenge.

James Holland <James.Holland@babcockinternational.com>

I can see that the student is trying to use **std::sort()** to iterate through the parameters of **least()** in order to find the one with the lowest value. Unfortunately, the code relies on the parameters being stored contiguously starting with first. This may not be the case and is not guaranteed by the standard. One way around this problem is to copy the parameters into a container from which **std::sort()** can operate, as shown below.

```
#include <algorithm>
#include <array>
#include <iostream>
template <typename T, typename... Ts>
T least(T first, Ts... rest)
{
  std::array<T, sizeof...(rest) + 1> a{first,
    rest...};
  std::sort(a.begin(), a.end());
  return *a.begin();
}
```

The container I have chosen is of type **std::array** and is named **a**. In order to declare **a**, the type of its elements needs to be specified. This is conveniently provided by the template parameter **T**. The declaration of **a** also needs to specify the length of the array. The array has to accommodate however many parameters there are in the parameter pack rest plus the parameter **first**, and is simply calculated by the expression **sizeof...(rest) + 1**.

Finding the smallest value of the array does not require **std::sort()**. The algorithm **std::min_element()** will be faster and, in this case, slightly easier to use.

```
template <typename T, typename... Ts>
T least(T first, Ts... rest)
{
  std::array<T, sizeof...(rest) + 1> a{first,
    rest...};
```

```
  return *std::min_element(a.begin(),
    a.end());
}
```

There is also a recursive approach that can be used to find the minimum value as shown below.

```
#include <algorithm>
#include <iostream>
template <typename T>
T least(T first)
{
  return first;
}
template <typename T, typename... Ts>
T least(T first, Ts... rest)
{
  return std::min(first, least(rest...));
}
```

When `least()` is called with more than one parameter, the version with the parameter pack is invoked. The returned value is the minimum of its first parameter and the minimum of all the other parameters. The minimum value of all the other parameters is obtained by calling `least()` again but this time not including the first parameter. This recursive arrangement keeps going until `least()` is called with just one parameter, at which point the version without the parameter pack is invoked. This version simply returns its parameter (the minimum of just one value is the value itself). Eventually, when all the invocations of `least()` have returned, the minimum value of the entire sequence is obtained.

It is interesting to note that although I have described this technique as being recursive, it is only recursive from a source code point of view. The compiler generates a series of `least()` functions each with a different number of parameters ranging from 1 to *n* where *n* is the number of elements in the sequence. Which one is called depends on the number of parameters provided. In this way, when `least()` 'recurses' it is a different function that is called each time.

It is possible to do away with the recursion and let `std::min()` operate on the whole sequence in one go as shown in my final version of `least()` below.

```
template <typename... Ts>
auto least(Ts... all)
{
  return std::min({all...});
}
```

Here, the parameter pack is expanded and used to create an unnamed object of type `std::initializer_list` that is passed to `std::min()`. The return type of `least()` is automatically deduced.

Jason Spencer <contact+pih@jasonspencer.org>

The headline bug of the unreliability of this code is down to function calling conventions for different platforms/languages/architectures.

Specifically, the greater number of registers available on 64-bit CPUs may mean the compiler passes function arguments in registers rather than on the stack, so we can't use pointer magic to calculate the end iterator.

The issue can be seen by entering the following program in to Matt Godbolt's Compiler Explorer [1]:

```
A0> int fn (int a, int b, int c, int d, int e,
    int f, int g) {
A1>
A2>     return (a+b+c+d+e+f+g);
A3> }
A4> int main() {
A5>      return fn(1,2,3,4,5,6,7);
A6> }
```

With the x86-64 gcc 8.1 compiler and no switches (therefore defaulting to amd64) the call looks like:

```
B0> push    7
B1> mov     r9d, 6
B2> mov     r8d, 5
B3> mov     ecx, 4
B4> mov     edx, 3
B5> mov     esi, 2
B6> mov     edi, 1
B7> call fn(int, int, int, int, int, int, int)
B8> add     rsp, 8
B9> nop
```

The first six arguments are copied to registers (lines B1 to B6), but the seventh, the literal 7, is pushed to the stack (line B0).

When compiled with the same compiler and the `-m32` switch (telling the compiler to generate a binary for x86) the call looks like this:

```
C0> sub     esp, 4
C1> push    7
C2> push    6
C3> push    5
C4> push    4
C5> push    3
C6> push    2
C7> push    1
C8> call fn(int, int, int, int, int, int, int)
C9> add     esp, 32
CA> nop
```

The subtraction in line C0 is to keep the stack pointer correctly aligned (on x86 the `esp` value has to be aligned to a 16 byte boundary at the function call).

The return value in both cases is returned in register EAX.

Lines B8 and C9 put the stack pointer to where it was before, effectively wiping that part of the stack used to transfer arguments.

C++ templates are instantiated at compile time, and the same template, but with a different number of arguments, is effectively a different function (which is why some people complain that templates cause "code bloat").

As such the instantiation, aka function, with four arguments may have all arguments reside in registers, on both x86 and amd64, but the function with 10 arguments will have some on the stack and some in registers.

The bit that's missing from the above snippets is what happens to the passed arguments in the *called* function. They can be pushed from the registers on to the stack, which is what actually happened in the case above (but not shown here), but that is not always the case.

So why do we care about how it's passed? Well, in the student's `least` templated function we take the address of the first argument and we use that as the start iterator passed to `std::sort`, and from that we do some arithmetic to find the end iterator (the one after the last element). But because of the calling conventions, and because we are dealing with raw pointers, the elements to be sorted may not be contiguous in memory and they may also be in registers. The arguments could also be pushed on the stack in reverse order (but that's not what happens above since the stack grows 'downwards' in memory, hence the add in C9 and B8 actually unwinds the stack). If you attempt to take the address of one of the arguments, ie `&first`, and it is in a register, then the compiler will copy it first to the stack – but the other elements it may not.

Don't take this is as a hard rule, though – the way arguments are passed and whether the calling or the called function does the cleanup depends on a number of things, not just the CPU architecture, and can change between OSes, the language, the compiler, and even compiler version – see [2], [3], [4], [5], [6] and [7] for more details. You can also add attributes to functions to change the calling convention on a per-function basis, for example [8]. It is these differences that causes problems for the student.

Also, since we're dealing with literal values, when optimisation is enabled most compilers will calculate the least value at compile time, rather than at runtime anyway (and magically work!).

Put simply, you should never ever make such assumptions about the calling conventions.

A few other things stand out straight away in the student's code:

- The code unnecessarily uses sort to get the lowest value. `std::sort` is guaranteed to be O(N ln N) in time complexity, but we don't need all of the values sorted, we just need the lowest value. To do this we can simply scan through the values keeping a record of which is the lowest value. This can be done in O(N) since each value has to be checked only once.
- The use of pass by copy – this requires the type to be copyable – fine if it's a POD (plain-old-datatype), not so much if it's a UDT (user-defined-type). Prefer to pass by const reference or const copy (the latter *may* lead to copy elision, in some circumstances).
- There is a potential `reinterpret_cast` of arguments: `std::sort` deduces the type of the elements being sorted from the iterators being passed to it (they must both have the same type). With a variadic template, the type of each argument can be different. So if the first argument is a `char` (say, one byte in size) and the remaining three arguments are `long long` (say, 8 bytes), then the type to be sorted is deduced as `char` but the end iterator points to the `char` four (1+sizeof...(rest) => 1+3) **bytes** later, which is not even the whole way through the first `long long` argument. And the bytes extracted from the first `long long` argument are effectively nonsense anyway. The precise values are dependant on the endianess of the architecture. This wasn't seen here because the literal digits were assumed to be of type `int`, but adding a `float` in the middle will easily corrupt the result.
- And the elephant in the room is that the least function is already part of the STL since C++14: `template< class T > constexpr T min( std::initializer_list<T> ilist );` Bear in mind, though, that `initializer_lists` pass by copy, so `T` has to be again copyable.

As to an alternative implementation… I don't know… it all depends on what you really want from it.

To go completely the other way from a pass-by-copy function, and if you're particularly weird, you could pass by universal reference everywhere and return such a reference, so you could write a function that will let you overwrite the left-most lowest value:

```
template <typename T> T&& least(T&& first) {
  return (std::forward<T>(first));
}
template <typename T, typename... Ts> T&&
  least(T&& first, Ts&&... rest)
{
  auto && l =
    least(std::forward<Ts>(rest)...);
  return std::forward<T>(((first <= l ) ?
    first : l));
}
```

The template here starts from the last value and works forward, so make the ternary test `(first < l )` if you want a reference to the right-most lowest value. And obviously `>=` if you want your `least` function to perform a left-most `most` search.

So now that we're dealing with references we can do weird things like this:

```
int main() {
int a = 3, b = 1, c = 2, d = 0;
std::cout << "Before : " << a << ',' << b <<
  ',' << c << ',' << d << '\n';
least(a,b,c,-1,d) = 42;
std::cout << "TP1 : " << a << ',' << b <<
  ',' << c << ',' << d <<'\n';
least(a,b,c,d) = 42;
std::cout << "TP2 : " << a << ',' << b <<
  ',' << c << ',' << d <<'\n';
std::cout << "TP3 : " << least(2,3,4,5,6,78,9)
  << '\n';
}
```

Where the output is:

```
$ ./a.out
Before : 3,1,2,0
TP1 : 3,1,2,0
TP2 : 3,1,2,42
TP3 : 2
$
```

So now `least` returns a reference to the lowest value and we can overwrite it. I think this is iffy for a few reasons, the least of which is that it might be confusing as to what the function really does.

Another issue is that the type of each argument can be different and the values may undergo a narrowing conversion (by implicit cast) to different types with loss of precision in the less than comparison. For example, `least ( 0.9, 2, 3, 4 )` will return 0.9, but `least ( 1, 2, 3, 0.9 )` will return 0.

My preferred approach, however, is to decompose the problem – we're reducing/folding according to some operator. So why not implement a fold/reduce template, pass the reduction operation as a parameter and let the compiler do the heavy lifting. Of course C++17 also has fold expressions.. but they only work for a handful of operators (+, -, * etc) and we can forsake the syntactic sugar of fold expressions and go further with a customisable reduction.

So let's create a template that'll do our reduction and which takes the reducing functor as a parameter:

```
template < typename OP> struct lfold {
OP op;
template <typename T, typename... Ts>
constexpr const T operator()(const T & first,
  const T & second, const Ts&... rest)
{
  if constexpr ( sizeof...(rest) == 0 )
    return op(first, second);
  else
    return operator()(op(first, second),
      rest...);
}
};
```

We're returning by copy deliberately – because `op` may not be returning a reference, and may be returning a temporary.

We're also using an `if constexpr` so we only need one template. Note also we're taking two arguments at a time, and the adjacent arguments must have the same type, so they effectively all have to be the same type. Passing a series of `int`s and a `double` will cause a compile error, rather than a silent implicit conversion.

We could now write a functor that returns the lower of two values and be done with it… instead let's write a functor that selects one argument according to a comparator that is provided as a parameter:

```
template < typename CMP > struct selector {
CMP cmp;
template <typename T> constexpr T &
  operator()( T & first, T & second )
{
  return (cmp(first,second)?first:second);
}
};
```

Ok, so now we can do something like this:

```
using least =
  lfold < selector<std::less<void>> >;
using most =
  lfold < selector<std::greater<void>> >;
```

Note the use of `void` as the template parameter – since C++14, this means that the type deduction is deferred, so it means we don't have to provide an explicit type in the `typedef`.

Now that we have a general-purpose reduction template we can also do things like:

```
using sumer = lfold < std::plus<void> >;
using producter =
  lfold < std::multiplies<void> >;
```

And it gets crazier – **std::plus** works on anything that has an **operator+** overload, so we could even concatenate strings:

```
using namespace std::string_literals;
std::cout << sumer{}("a"s,"b"s,"c"s,"d"s)
  << '\n';
```

outputs **abcd**.

Note, though, that the order of summing is significant when the type is a string… which is why our reduction template is called **lfold** (left fold). A right fold would look like:

```
template < typename OP> struct rfold {
OP op;
template <typename T, typename... Ts>
  constexpr const T operator()(
    const T & first, const T & second,
    const Ts&... rest)
{
  if constexpr ( sizeof...(rest) == 0 )
    return op(second, first);
  else
    return op(operator()(second, rest...),
      first);
}
};
```

And with:

```
using rsumer = rfold < std::plus<void> >;
std::cout << rsumer{}("a"s,"b"s,"c"s,"d"s)
  << '\n';
```

would output **dcba**.

If the creation of a least, most, rsumer, sumer, etc. temporary (through the use of **{}** initialisation) is not acceptable then there may be mileage in calling the **operator()** from a variadic template constructor which takes the actual arguments, and the temporary object returns the result by a conversion operator. I prefer the approach above, however, because we can initialise the reduction operator and we can do things like this:

```
struct ewma_op {
  double ewma;
  template <typename T> constexpr T
    operator()(const T & data1,
      const T & data2) {
    return (((1.0-ewma)*data1) +
     (ewma*data2));
    }
};
using ewma = lfold < ewma_op >;
std::cout << ewma{0.1}(0.1,0.2,0.3,0.4)
  << '\n';
```

The last line calculates the exponential weighted moving average of the arguments with a smoothing co-efficient of 0.1.

The **{0.1}** is used to initialise the **lfold::op**… which is also a struct that supports brace initialisation… so that value is used to set **ewma_op::ewma**… and we can magically set the smoothing co-efficient.

And the coolest thing about this (if you supply literals, and depending on the types used) is that it's **constexpr** and the compiler will optimise out much of it, even finding the entire result at compile time and there'll be no runtime cost.

You could also use **lfold** with a **hash_combine_op** that is initialised with a seed, then takes a list of arguments for reduction, calculates the hash (with a hash function specified as a template parameter), combines the hash value (see **boost:hash_combine**) with the seed and continues (thanks to **lfold**) through the list of arguments.

But for now, I think that solves the original problem.

References

[1] https://gcc.godbolt.org/
[2] https://en.wikipedia.org/wiki/Application_binary_interface
[3] https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
[4] https://www.agner.org/optimize/calling_conventions.pdf
[5] https://software.intel.com/en-us/node/682402
[6] https://en.wikipedia.org/wiki/X86_calling_conventions
[7] https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions
[8] https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html

Commentary

The code in this critique relies on being able to sort a set of function arguments as if they were a range. As all our entries pointed out to greater or lesser extent, this assumption is flawed in a number of ways.

Firstly, the standard requires that a range refers to a single C++ object, so irrespective of the specifics of the implementation treating the *separate* objects as a range is **undefined** behaviour.

Secondly, the code is assuming that the arguments are contiguous in memory with increasing addresses. While this may be true in some cases – for example on some platforms with un-optimized 32bit calls using the default calling convention – it is not in general true. some calling conventions push pass the arguments left-to-right, other push them right-to-left, and in many calling conventions the first few arguments are passed in registers and others are passed on the stack; if the address of one of the former arguments is taken then the register value will be copied into a memory location and its address will be used. This address may have no obvious relationship to the addresses of the arguments passed on the stack!

Thirdly, the code is assuming that the arguments are the same size as the amount of stack that they consume. So, even on a 32-bit platform, the original code was highly likely to fail if called with char arguments.

As everyone noticed, the code did too much work in that the whole range was sorted although only the single least value was required. Over the years I have noticed a number of places that used **std::sort** when only a small part of the sorted output was being used; there are other algorithms that can be used such as **std::partial_sort**.

One open question is what the code should do with arguments of different type. The code provided would *accept* them but do the wrong thing. As is common with these critiques, and is also the case for many of us in our working life, there's no formal specification.

Postscripts to recent Code Critique competitions

Jason Spencer has sent us some reflections and updates to his entries.

Postscript to Code Critique 111

In an attack of involuntary stupidity, not only did I not comment on the sequence point issue in Code Critique 111 but – even worse – I made the mistake myself! I spotted it in the first pass when looking at the Code Critique, but then I context switched to other languages while doing the write-up. It's down to an annoyance between many C-like languages – they look similar, but they don't always act similar. Java, C# and Javascript would interpret:

```
a = 42;
a = ++a + a++;
```

as:

```
a = 43 + a++;
a = 43 + 43; // but a is 44 (because of the post-
             // increment) before it is assigned
             // the value of the summation
a = 86;      // and a is now 86, overwriting the
             // 44 value in a
```

That's because they guarantee (to the best of my knowledge) to *evaluate* expressions from left to right once operator precedence has been taken into

consideration. Because the assignment operator has a low precedence and right-to-left associativity the expression on the right of the operator will be computed first.

C and C++ do not guarantee the order in which expressions are evaluated.

The good news is that g++ flags this as undefined behaviour when invoked with the `-Wall` or `-Wsequence-point` switch, but clang++ and VS don't complain, even with `-Wall` /`Wall`.

In practice, it looks like this:

VC++	-	87
VC++	-O3	87
g++	-	87
g++	-O3	87
clang++	-	86
clang++	-O3	86
ICC	-	86
ICC	-O3	86

Alas, I don't have the compiler versions to hand, but you get the point – different compilers use different ordering of evaluation as it's not guaranteed by the standard. Check out godbolt.org to try the code snippet above.

Mistakes like this are a good reason to spread out code and program in a more defensive and explicit way, with the increment on a line of its own.

Postscript to Code Critique 109

While working towards a `zipit::operator*` implementation, I mimicked `std::vector<bool>::reference` by implementing a proxy object which acts like a reference.

Throwing caution to the wind, in a spasm of flippancy, I suggested an implementation of `operator->` technically could (but it's really ugly and it shouldn't) return a (smart) pointer to a dynamically allocated proxy object.

`vector<bool>` does not do this, nor Boost.ZipIterator, and for good reason – it's not only hideous, it's also against the specs.

It has come to my attention through a discussion between Niels Dekker and Anthony Williams on the accu-general mailing list that in fact the C++ specification requires `operator->` to return a raw pointer, which would forbid the use of a proxy object. So the idea of returning a smart pointer to a proxy object is even more wrong.

Right, now I need to go and boil my head.

The winner of Code Critique 113

The critiques all identified the problem and proposed alternative approaches without the troublesome behaviour. I found Stewart's phrasing of the difference between the size of an `int` and the size of a stack word to explain why the code doesn't work on 64bits particularly clear,

I like Joe Wood's mention of using sanitizers to help debug the problem: there are a lot of good tools out there to help with identifying problematic code in an automated fashion.

Jason Spencer provided some very nice examples of where you might take this technique further (although the universal reference version of least may behave unexpectedly if the returned reference is to a temporary.) The suggestion of using godbolt is a good idea to help see the generated output in a readable format.

I liked James Holland's final version of his solution to the problem, which simply provides a forward to the standard `min` function taking an `initializer_list`. This solution makes great use of pre-existing standard facilities.

Stewart provides a good dialogue as he works towards his solution, and I think many programmers would find this approach would not only help them to understand the solution, but also *how* one might reach such a solution.

After some deliberation on the strengths of the various entries, I have awarded this issue's prize to Stewart.

Code Critique 114

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

> I am trying to write a simple test for some C++17 code, but I am getting some unexpected output. I have simplified the code quite a bit, and in the resultant code below I was expecting to see the output "A,B" but on the latest version of both gcc and MSVC I get just "AB". Where has my comma gone?! Even more oddly, if I use an older compiler I get different output: "65,66", which gives me back the comma but loses the letters. Has C++17 broken something?

The code 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 <iostream>
#include <string>
#include <vector>
template <typename T>
using coll = std::vector<T>;
// Start the collection
template <typename T>
void start(coll<T> &up)
{
  up.clear();
}
// End the collection
template <typename T>
void end(coll<T> &up)
{
  up.push_back({});
}
// Extract the non-zero data as a string
template <typename T>
std::string data(coll<T> const &up)
{
  std::string result;
  for (auto v : up)
  {
    if (v)
    {
      result += std::to_string(v);
      result += ",";
    }
  }
  result.pop_back();
  return result;
}
void test_char()
{
  auto i2 = coll<char>();
  start(i2);
  i2.push_back('A');
  i2.push_back('B');
  end(i2);
  // actual test code elided
  std::cout << data(i2) << '\n';
}
int main()
{
  test_char();
}
```

View from the Chair

Bob Schmidt
chair@accu.org

ACCU 2019

ACCU's 2019 Conference will be held from the 10th to 13th of April, 2019, with a day of pre-conference tutorials on 9th April [1]. Our conference chair, Russel Winder, has announced the four keynote speakers: Angela Sasse, Kate Gregory, Paul Grenyer, and Herb Sutter. Make sure to mark that week in your calendar! I hope to see you there.

Web site crash!

Jim Hague, ACCU's web master, reported to the committee that our web site crashed in August due to extremely high activity. It turns out that the *Overload* 146 article 'Cache-line Aware Data Structures' hit the high rankings on HackerNews [2], went viral, and our server collapsed under the weight of all the download requests. Our web host, Bytemark, added resources to the server, and the site was back up and running in short order. Congratulations to authors Wesley Maness and Richard Reich for (briefly) crashing accu.org!

GDPR

There have been some minor changes to ACCU's Privacy Policy [3]. Nigel Lester continues to refine the policy in response to questions and comments. Thank you to everyone who has taken the time to read the policy and provide feedback.

Code of Conduct and Diversity Statement

I'd like to take this opportunity to remind everyone that ACCU has a Code of Conduct (CoC) and a Diversity Statement [4] that articulate our core values. The CoC applies to all ACCU activities, conferences, local group meetings, and interactions on ACCU's mailing lists. Our conference has its own, expanded Code of Conduct [5].

ACCU's mailing lists have charters that describe and proscribe the topics that may be discussed. In keeping with the CoC, all ACCU mailing lists are moderated; each list has its own group of moderators. In addition, the ACCU committee bears ultimate responsibility for ensuring that all of ACCU's spaces remain friendly and open to all our members and guests. Please keep the charters and CoC in mind when using ACCU's mailing lists.

ACCU committee

It may seem a bit early to be talking about an election and an Annual General Meeting that won't be held until next April, but we start running into constitutionally mandated deadlines for the Annual General Meeting and elections starting in January (which is only one *CVu* issue away). We are going to have several critical openings on the committee, and they must be filled in order for ACCU to continue running smoothly.

Treasurer

Rob Pauer has been ACCU's treasurer for many years now. Rob retired from his 'real' job some time ago, and now would like to retire as treasurer as well. He has offered to continue as treasurer until April, and has offered to mentor a new volunteer during a transition period. Unlike most of our committee positions, the role of treasurer requires a person who is located in the UK in order to have physical access to our bank.

It should be obvious – without a treasurer, ACCU will be unable to pay its bills and will be unable to function. ACCU hires and pays for a production editor to lay out our magazines; a printer to print the magazines; and postage to send those magazines around the world. ACCU receives money from new and renewing members, and from the proceeds of our conference. ACCU supports our local groups with financial assistance, and maintains a separate fund for supporting standards activities. Without a treasurer to receive and disburse funds, all of this activity stops. If this activity stops, so does ACCU.

Conference chair

When our current conference chair, Russel Winder, took on the role, he agreed to a four-year ~~sentence~~ term. That term expires as of the end of ACCU 2019, so ACCU needs to find a new conference chair. As with the treasurer position, it would be helpful if the new conference chair shadows Russel as we approach ACCU 2019, in order for there to be a smooth transition. This is one of ACCU's most important positions – our yearly conference is one of the four 'faces' of ACCU (the magazines and web site being the other three). As with Rob Pauer, Russel deserves his well-earned retirement.

Chair

On a personal note, I would like to take this opportunity to announce that I will not be seeking a fourth term as Chair when my current term is up in April. It is time for someone new to inject their ideas and leadership into the organization.

What does this mean for ACCU?

This means that at least two of the four executive committee positions will be open, and need new volunteers, as of the 2019 Annual General Meeting. Vacant executive committee positions put ACCU into caretaker mode, which limits the actions that can be taken by the rest of the committee.

Almost every one of my Views has included a 'Call for Volunteers'. This View is no different, except perhaps in scope. We still have other positions available. At our last committee meeting we discussed eliminating some of the positions, such as Mentored Developers and Study Groups, due to inactivity in those areas and a lack of someone to take the positions. A final decision on disbandment has not been reached, so there is still time to save them if someone steps forward.

Volunteering to help run ACCU is a way to give back to your community, increase your industry profile, and to hone your skills running and building a diverse community. It looks good on your CV, too. Please contact me if you are interested in volunteering for one of these positions.

References

[1] ACCU 2019: https://conference.accu.org/
[2] HackerNews: https://news.ycombinator.com/
[3] ACCU Privacy Policy: https://accu.org/index.php/privacy_policy
[4] ACCU Code of Conduct and Diversity Statement: https://accu.org/index.php/aboutus/diversity_statement
[5] ACCU Conference Code of Conduct: https://conference.accu.org/coc_code_of_conduct.html

JOIN THE ACCU!

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

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

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel code.

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 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE