the magazine of the accu

www.accu.org

{cvu}

Volume 31 • Issue 1 • March 2019 • £4.50

Features

Regulars

TICKETS ON SALE NOW!

ACCU 2019

2019-04-10 to 2019-04-13

MARRIOTT CITY CENTRE HOTEL, BRISTOL

ACCU is the conference for all those interested in native languages and professionalism in programming. Bringing together some of the best minds and game changers in the industry, ACCU is excited to return to Bristol for 2019.
Annually attracting a host of renowned international speakers, 2019 will take an interesting look at the way technology is evolving.

Day Tickets: From £200.00 | **Full 4 Day Conference Tickets:** From £670.00

PRE-CONFERENCE FULL DAY WORKSHOPS 2019-04-09

From £200.00 (all ticket prices exclude VAT)

Dinner entertainment from

I am ECHOBORG

4 DAYS
400+ ATTENDEES
60+ SPEAKERS
5 PARALLEL STREAMS

KEYNOTE SPEAKERS

HERB SUTTER

KATE GREGORY

PAUL GRENYER

M ANGELA SASSE

TICKETS >> conference.accu.org

@ACCUConf #ACCUConf

{cvu}
Volume 31 Issue 1
March 2019
ISSN 1354-3164
www.accu.org

Nobody Does It Better

At some point over the course of our lives as software developers, we all – I suspect – succumb to the idea that we, as individuals, are the crucial component of the team on a project, and that the (continued) success of that project is entirely dependent on us.

It's a tempting idea, because to be indispensable might mean job security, or merely kudos among our peers, but it's an attractive thought.

There is a popular image of programmers as lone-hackers surrounded by pizza, coffee and doughnuts, face green-lit by monitor glare. It might be misrepresentation by segments of the media, but I wonder how many of us secretly want it to be true.

The thing about being indispensable is, of course, that it's a myth, at least for most of us. A good thing too, probably: if you're going to take all the credit when things go well, you have to accept all the responsibility when they don't! It's much better to share both things amongst a whole team, because it is in this way that important lessons are not just learned, they're passed on to the larger community much more effectively, too.

I've certainly had my own personal moments of believing (or at least, wishing) myself to be the Hero Programmer, but I've learned that being able on the one hand to depend on a team of people, and on the other to be able to share what I have learned might make me *dispensable*, but overall it makes me better at my job, and opens up all sorts of opportunities I would have missed were I that lone hacker, alone in a darkened room.

Despite being a fairly young industry, there's already quite a lot of collective knowledge about how we go about our business. Being able to tap into that knowledge is crucial for us if we, collectively, are to continue to grow and learn about our craft. Sure, it's moving all the time, and we *are* still learning as we go, but it's my belief that we're getting better at it. I am, anyway.

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 31.2: 1st April 2019
C Vu 31.3: 1st June 2019

Overload 150: 1st May 2019
Overload 151: 1st July 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

The Simple Life

Pete Goodliffe urges us to keep code simple.

Simplicity is the ultimate sophistication.
~ Leonardo da Vinci

You've heard the advice before: 'KISS.' *Keep it simple, stupid*. Exactly how stupid do you have to be to get *that* wrong? Simplicity is an undoubtedly excellent objective; you should certainly strive for it in your code. No programmer yearns to work with overly complex code. Simple code is transparent; its structure is clear, it does not hide bugs, it is easy to learn, and easy to work with.

So why isn't all code like that?

In the developer world, there are two kinds of simplicity: the *wrong* sort and the *right* sort. The 'simplicity' we are looking for specifically does *not* mean: write your code the easiest way you can, cut corners, ignore all the nasty complicated stuff (brush it all under the rug and hope it goes away), and generally be a programming simpleton.

Oh, if only it were that easy. Too many programmers in the real world *do* write 'simple' code like this. Their brain does not engage. Some of them don't even realise that they're doing anything wrong; they just don't think enough about the code they're writing, and fail to appreciate all of the inherent subtle complexities.

Such a mindless approach leads not to simple code, but to *simplistic* code. Simplistic code is incorrect code. Because it is ill-thought through, it doesn't perform exactly as required – often it only covers the obvious 'main case', ignoring error conditions, or it does not correctly handle the less likely inputs. For this reason, simplistic code harbours faults. These are cracks that (in the typical simplistic-coder way) tend to get papered over with more ill-applied simplistic code. These fixes begin to pile on top of one another until the code becomes a monstrous lumpy mess; the very opposite of well-structured, simple code.

Simplicity is never an excuse for incorrect code.

> Simple code takes effort to design. It is not the same thing as overly simplistic code.

Instead of this wrong simple-minded 'simplicity', we must strive to write the *simplest* code possible. This is very different from disengaging your brain and writing stupid, simplistic code. It is a very brain intensive pursuit – ironically, it's hard to write something simple.

Simple designs

There is one sure sign of a simple design: the fact that it can be quickly and clearly described, and easily understood. You can summarise it in a simple sentence, or in one clear diagram. Simple designs are easy to conceptualise.

Simple designs have a number of notable properties. Let's take a look.

Simple to use

A simple design is, by definition, simple to use. It has a low cognitive overhead.

It is easy to pick up because there is not too much to learn at first. You can start working with the most basic facilities, and as you need to adopt the more advanced capabilities, they gradually open up like a well-crafted narrative.

as you need to adopt the more advanced capabilities, they gradually open up like a well-crafted narrative

Prevents misuse

A simple design is hard to misuse and hard to abuse. It reduces the burden on the code's clients by keeping interfaces clean and not putting an unnecessary burden on the user. For example, a "simple" interface design will not return dynamically allocated objects that the user has to manually delete. The user will forget. The code will leak or fail.

The secret is to place the complexity in the right places: generally hidden away behind a simple API.

Simple designs aim to prevent misuse. They may involve extra internal complexity in order to present a simpler API.

Size matters

Simple code minimises the number of components in the design. Big projects with many moving parts may justifiably require a large number of components; it is possible to have many flying parts and be ``as simple as possible.'

> Simple designs are as small as possible. And no smaller.

Shorter code paths

Remember the famous programmers' maxim: *every problem can be solved by adding an extra level of indirection*? Many complex problems can be subtly masked, and even caused by, unnecessary levels of indirection hiding the problem. If you have to follow a long chain of function calls, or trace indirected data access through many levels of ``getter' functions, forwarding mechanisms, and abstraction layers, you will soon lose the will to live. It's inhuman. It's unnecessarily complex.

Simple designs reduce indirection, and ensure that functionality and data are close to where it's needed.

They also avoid unnecessary inheritance, polymorphism, or dynamic binding. These techniques are all good things, when used at the right times. But when applied blindly, they bring unnecessary complexity.

Stability

The sure sign of a simple design is that it can be enhanced and extended without massive amounts of rewriting. If you continually end up reworking a section of code as your project matures, then you either have a ludicrously volatile set of requirements (which *does* happen, but is a very different problem) or you have an indication that the design was not simple enough in the first place.

Simple interfaces tend to be stable, and don't change much. You may extend them with new services, but do not need to rework the entire API. However, this should not be a straightjacket: interfaces need not be set in stone. Don't make your code unnecessarily rigid – this itself is not simple.

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

Simple lines of code

Simple code is easy to read, and easy to understand. It is therefore easy to work with.

Personal preference and familiarity tends to determine what makes individual lines of code look simple. Some people find that certain layout idioms help clarify their code. Others find those same idioms a huge hindrance. More than anything else, *consistency* leads to simple code. Code with widely varying styles, naming conventions, design approaches, and file formats is needlessly obfuscated.

> Consistency leads to clarity.

Do not write needless obscure code for any reason: not for job security (we joke about this, but some people truly do), not to impress your colleagues by your coding prowess, and not to try out a new language feature. If you can write an acceptable implementation in mundane, but clear, coding style then do so. The maintenance programmers will thank you for that.

Keeping it simple, not stupid

When you encounter a bug, there are often two ways to address it:

- Take the easiest route to solve the problem. Hey, you're keeping things *simple*, right? Fix the superficial problem – that is, apply a sticking plaster – but don't worry about solving any deeper underlying issues if it will be too much work. This is the least effort for you now, but will likely lead to the kind of simplistic code mess we saw earlier.

  This doesn't make things simpler; it makes things more complex. You've added a new wart and not addressed the underlying problem.

- Or, you can rework the code so that it accommodates a fix, and remains simple. You may have to adjust APIs to be more appropriate, refactor some logic to create the correct seam for the bugfix, or even perform serious rework because you spot code assumptions that do not hold.

This latter option is the goal. It does require more effort, but boiling the code down to its simplest form pays off in the long run.

Apply bugfixes to the root cause, not where symptoms manifest. Sticking plaster symptom-fixes do not lead to simple code.

Assumptions can reduce simplicity

Invalid 'simplifying' assumptions are easy to make when you're coding and, whilst they can reduce the complexity in your head, they tend to build into twisted logic.

Simple code does not make unnecessary assumptions, either about the requirements or problem domain, about the reader, about the runtime environment, or about the toolchain used. Assumptions can reduce simplicity, as you implicitly require the reader to know extra information to make sense of the code.

> Avoid implicit assumptions in your code.

Assumptions can, though, *increase* simplicity. The trick is to make it clear exactly what assumptions are being made; for example, the constraints and context that the code is designed for.

Avoid premature optimisation

Optimisation is the antithesis of simplicity. Knuth, in his 1974 Turing Award lecture 'Computer Programming as an Art, famously said: "Premature optimisation is the root of all evil (or at least most of it) in programming."

The act of code optimisation is generally that of taking a straightforward, readable, algorithmic implementation and butchering it: pulling the algorithm out of shape so that it executes faster on a given machine under particular conditions. This inevitably alters the shape to be less clear and, therefore, less simple.

Write clear code first. Make it complex only when needed.

Employ a simple, standard sort, until you know you need to make it cleverer. Write the most straightforward implementation of an algorithm and then measure to see if you need to make it faster. Again, beware of making assumptions: many programmers optimise the parts they *think* will be slow. The bottlenecks are often elsewhere.

When working at the design and architecture level, you can make decisions that have real impact on how your system performs. You might consider the decisions made here a form of optimisation. When you get to the code level, these architectural concerns are not easily reversible, and you can try to enhance code performance with low-level tweaks. These tweaks tend to reduce clarity, and obfuscate the code logic in order to eke out a few CPU cycles.

Sufficiently simple

Simplicity is allied with *sufficiency*. This works in a few directions:

- You should work in the simplest way possible and write the simplest code possible. But keep it sufficiently simple. If you oversimplify, you will not solve the actual problem. Our 'simple' solutions *must* be 'sufficient' or they are *not* solutions.
- Only write as much code as is required to solve your problem. Don't write reams of code that you *think* will be useful. Code that is not in use is just baggage. It's an extra burden. It's complexity you don't need. Write the sufficient amount of code. The less code you write, the fewer bugs you'll create.
- Don't overcomplicate solutions; excitable developers find this a very real temptation. Solve only the issue at hand. Don't invent a needlessly general solution for a whole class of problems that are not relevant. Work until you reach a splendid sufficiency.

> Only write as much code as is needed. Anything extra is complexity that will become a burden.

A simple conclusion

We all know that beautifully simple code is better than needlessly complex code. And we've all seen our fair share of foul, ugly, complex code. Few people aim to write code like that. The road to complexity is usually trodden with hurried changes and slipping standards. Just one slack change. Just one sticking plaster fix. Just one code review skipped. Just one "I don't have time to refactor." After enough of these, the code is an unholy mess, and it's hard to figure a way to restore sanity.

Sadly, simplicity is pretty hard work.

Simplicity is a banner born out by many popular developer maxims: YAGNI – *you aren't going to need it* – speaks to the theme of sufficiency. DRY – *don't repeat yourself* – speaks to the theme of code size. Our preference for high cohesion and low coupling speaks to simplicity in design. ■

Questions

- What is the simplest code you've seen recently? What is the most complex code you have seen? How did they differ?
- What sort of unnecessary assumptions can a coder make about their code that will render it too complex? What assumptions are valid to make?
- Is it possible to optimise code but maintain its simplicity?
- Does the 'simplicity' of a section of code depend on the capabilities of the programmer reading it? How should an experienced coder work in order to ensure their code is of high quality, but appears 'simple' to a less experienced maintenance coder?

C++ On Sea 2019 Trip Report

Arne Mertz and Frances Buontempo share their experiences of a new C++ conference.

Arne Mertz

From February 3rd through February 6th I have been in Folkestone, UK, to visit the first *C++ On Sea* conference. There must be something in the water on that island that enables them to organize fantastic conferences like *ACCUConf* [1] and, since this year, *C++ On Sea* [2].

C++ On Sea is definitely the best conference I have ever been to, and here's a little glimpse why I think so.

To put my experience in context, here is what my main reasons are for going to conferences:

I arrived in Folkestone on Sunday evening. It is in Kent, UK, directly where the channel tunnel emerges. Leas Cliff Hall, where the conference took place, is situated directly on the cliffs and has a great view that sometimes lets you see the French coast. If it's not windy, rainy and foggy as it was most of the time during the conference.

Conferences don't start with the opening speech or first keynote. They start on the day before, when people meet to chat and socialize. In this case, I had dinner with one of several groups of people from the #include<C++> community [3].

Monday, February 4th

On Monday morning, the actual conference programme started with an opening speech by the conference organizer Phil Nash.

C++ was not only in the conference name, but also in the details: The opener was titled "Hello, World", there was a "main()" plenary hall and session rooms titled "const west", "east const", and "unsigned". The latter was the smallest of the session rooms and had an overflow problem a few times, but luckily that did not lead to undefined behavior, because C++ conference attendees seem to be very nice people in general.

The list of C++ puns goes on, as you can see from the floor plan [4] and schedule [5].

The first keynote of the conference was 'Oh, the humanity!' by Kate Gregory [6]. She showed how the cliché of the logical, unemotional programmer is wrong. We can see lots of emotions in the code we write, and we better check whether it's the emotions we want to show towards our fellow programmers.

One of the takeaways from the talk was that single letter variable names are bad and a sign of laziness. I myself am guilty of this, and throughout the conference, lots of speakers commented on their own slides along the lines of "yeah, since Kate's keynote I know I should fix those".

After the keynote, normal sessions started in four tracks. The first talk I went to was 'Postmodern immutable data structures' by Juan Pedro Bolivar Puente [7]. Juanpe showed some great examples of very efficient, partially copy-on-write, vector-like data structures.

The next talk I visited was by Adi Shavit and is called 'What I Talk about When I Talk about Cross Platform Development' [8]. Adi showed how the 'Salami tactic' enables us to write as little platform-specific code as possible by building thin layers of platform-independent C and C++ APIs around a pure C++ domain core.

Subsequently, I went to the `main()` track to see Jason Turner talk about 'Practical Performance Practices Revisited' where he showed how seemingly innocent changes can enable the compiler to optimize our code. The talk contained a fair amount of 'life-godbolting' and showed how optimizer technology is advancing: Jason himself sometimes seemed to be surprised that newer versions of Clang reduced his examples to a no-op or a constant return statement.

After that followed the most amazing talk I have ever seen at a conference. The keynote 'Deconstructing Privilege' by Patricia Aas [9] does not seem to have anything to do with C++, but it has to do a lot with the people who use it.

In her talk, Patricia shows how even though we might have hardship in our lives, being spared discrimination due to race, gender, sexual preferences and/or other factors can give us a head start in life. If you look again at the list in my Tweet above, this talk is a great example of #2: I have never before seen a large room filled with so many people in complete stunned silence.

After the keynote, there was a session of 11 interesting, enlightening, and entertaining lightning talks. The day concluded with a gala dinner where speakers and attendants met to eat and chat.

Tuesday, February 5th

After a short kick off, day two of the programme started with another round of sessions. The first one I visited was 'Programming with Contracts in C++20' by Björn Fahller [10]. Contracts let the programmer define preconditions and postconditions for their functions that can, depending on compilation flags, be checked at run time.

While this sounds like a great thing to have, the current state of the standard draft looks rather disappointing to me. This is also due to some properties of the language that implementers cannot always work around. However, while I am writing this, a standing committee meeting takes

ARNE MERTZ

Arne Mertz is a C++ programmer living in Hamburg. He runs a blog called 'Simplify C++' at https://arne-mertz.de/.

FRANCES BUONTEMPO

Frances has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com

Arne's report was first published on *Simplify C++* on 20 February 2019 at https://arne-mertz.de/2019/02/cpp-on-sea-2019-trip-report/

place at Kona, where no less than seven papers regarding contracts are scheduled for discussion.

The second talk was quite something different in many regards. Hana Dusíková talked about 'Compile Time Regular Expressions', but not in person: Due to a cancelled flight, she had to give the presentation remotely. And, of course, Murphy's law provided some technical difficulties so she had to rely on Matt Godbolt as an avatar to handle her slides and, lacking a working camera, tell her about reactions from the audience.

This manner of talk delivery was quite an experience and had an element of fun, at least for the audience. But the talk itself was just amazing. Hana presented the intricacies of parsing regular expressions at compile time in a casual manner that makes constexpr template wizardry seem like a stroll in the park. In addition, there even was a working regex parser written in JavaScript *in her slides* (that Matt had to type the input for) to demonstrate the tree her library builds at compile time. Quite a few minds were blown by this talk.

Then it was time to give my own talk about 'Learning (and Teaching) Modern C++ – Challenges and Resources' [11]. In the presentation, I talked about how the quick evolution of our language makes it hard for teachers and authors to keep their knowledge and teaching materials up to date, and about the benefits and drawbacks of the different resources available to us.

As the next talk to watch I chose 'Sailing from 4 to 7 Cs: just keep swimming' by John Shearer [12], because it mentioned one of my blog posts [13] in the abstract and the talk: John presented how his team started out with the '4C' development environment (Clang, CMake, CLion, and Conan) and added three more 'C's and quite a few other letters to their tool stack.

The last keynote of the conference was by Matt Godbolt: 'What Everyone Should Know About How Amazing Compilers Are' [14], in which he showed us how good optimizers are at guessing what developers want to do. In a series of live compiler explorer examples, he presented how modern optimizers can sometimes melt multiple lines of code into a single assembly instruction.

The key takeaway from this keynote: Trust your optimizer, don't manually optimize in 'smart' ways without checking the output, because you might actually inhibit the optimizer from recognizing common patterns.

Conclusion

For me, the conference was a blast. I met a lot of people I had known from other conferences, quite a few that I had only met online before at #include<C++>, and made a handful of acquaintances I would not have made otherwise. Full marks for #1 of my conference checklist.

There were also quite a few talks that were fun and/or engaging to attend, for me and others. A check on #2 of the list, and also for #3, as I find it always rewarding to share thoughts and experiences with a room full of people, even though the prospect is extremely scary every time I submit a talk proposal.

If you have not been there, there are two things you should do: Try to get a ticket for the next instalment (hopefully 2020) and watch the videos on YouTube [15]. (At the time of writing, new videos are being uploaded every day.)

Frances Buontempo

Phil Nash organised a new conference, CppOnSea [2], this year. I was lucky enough to be accepted to speak, so attended the two conference days, but not the workshops.

Figure 1

There were three tracks, along with a beginners track, run by Tristan Brindle, who organises the C++ London Uni, which is a great resource for people who want to learn C++ [16]. I'll do a brief write-up of the talks I attended.

The opening keynote was by Kate Gregory, called 'Oh The Humanity!' She made us think about the words we use. For example, Foo and Bar trace back to military usage, hinting at people putting their lives on the line. Perhaps we need better names for our variable and functions. We are not fighting a war. What about one letter variable names? 'k. Nuff said. What about `errorMessage`? If you call the `helpMessage` instead, how does that affect your thinking?

Kate was also talking about trying to keep the code base friendly, to increase confidence (Figure 1).

I went to see Kevlin Henney next. I have no idea how to summarise this. He covered so many thing. What does structured programming really mean? By looking back to various uses and abuses of `goto`, by highlighting the structure in various code snippets, he was emphasising some styles make the structure and intent easier to see.

I saw Andreas Fertig next. Inspired by Matthew CompilerExplorer's Godbolt [17], he has created https://cppinsights.io/. Try it out. It unwraps some of the syntactic sugar, so you can see what the compiler has created for, say, a range-based `for` loop. This can remind you where you might be creating temporaries or have references instead of copies or vice versa, without dropping down into assembly. Do you know the full horror of what might be going on inside a Singleton? (See Figure 2.)

This led to an aside about statics and the double checked locking pattern. His headline point was the spirit of C++ is 'you pay only for what you use', so be clear about what you are using. The point isn't that the new language features are expensive. They are often cheaper than old skool

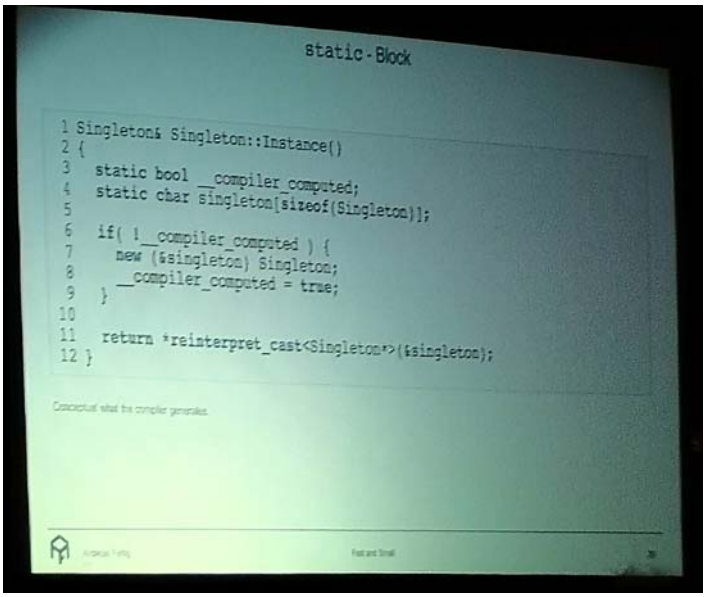


Figure 2

ways of going things. Just try to be clear about what's going on under the hood. Play with the insights tool.

Next up, I saw Barney Dellar, talking about strong types in C++. His slides are probably clear enough by themselves, since they have thorough speaker notes [18]. My main note to myself says 'MIB: mishap investigation board', which amused me. He was talking about the trouble that can happen if you have doubles, or similar, for all your types, like mass and force:

```
double CalculateForce(double mass);
```

It's really easy to use the wrong units or send things in the wrong order. By creating different types, known as strong types, you can get the compiler to stop you making mistakes. Use a template with tags, you can write clear code avoiding these mistakes.

Next up was a plenary talk by Patricia Aas on 'Deconstructing Privilege'. She's given the talk before, so you'll be able to find the slides [9] or previous versions on YouTube. Her take is that privilege is about things that haven't happened to you. Many people get defensive if you say they have been privileged, but this way of framing the issue gives a great perspective. Loads of people turned up and listened. Maybe surprising for a serious geek C++ conference, but the presence of https://www.includecpp.org/ ensured there were many like minded people around. If you are privileged, listen and try to help. And be careful asking intrusive questions if you meet someone different to you.

After quite a heavy, but great talk, I was 'in charge' of the lightning talks. Eleven people got slots. More volunteered, but there wasn't time for every one:

- Simon Brand – C++ Catastrophes: A Poem.
- Odin Holmes – volatile none of the things
- Paul Williams – std::pmr
- Heiko Frederik Bloch – the finer points of parameter packs
- Barney Dellar – imposter syndrome or mob programming
- Matt Godbolt – 'words of power'
- Kevlin Henney – list
- Neils Dekker – noexcept considered harmful???
- Patricia Aas – C++ is like JavaScript
- Louise Brown – The Research Software Engineer: A Growing Career Path in Academia
- Denis Yaroshevskiy – A good usecase for if constexpr

My heartfelt thanks to Jim from http://digital-medium.co.uk and Kevlin 'obi wan kenobi' Henney for helping me switch between PowerPoint, PowerPoint in presenter mode and the PDFs, and getting them to show on the main screen and my laptop. No body knows what was happening with the screen on the stage for the speaker. If you ever attend a conference, do volunteer to give a lightning talk. Sorry to the people we didn't have time for.

Day one done. Day two begun. First up, for me, after missing my own talk pitch, was Nico Josuttis. Don't forget his leanpub C++17 book [19]. It's still growing. He talked about a variety of C++17 features. The standout point for me was the mess you can get into with initialisation. He's using `{}` everywhere, near enough. Like

```
for (int i{0}; i<n; ++i)
{
}
```

Adding an equals can end up doing horrible things.

Much as I wanted to go see Simon Brand, Vittorio Romeo and Hana Dusikova (with slide progression by Matt Godbolt) next, I had a talk to do myself. I managed to diffuse my way out of a paper bag, while reminding us why C's rand is terrible, how useful property based testing can be, using some very simple mathematics: adding up and multiplying. This was based on a chapter of my book, and you can download the source code from that page if you want, even if you don't buy the book [20]. I used the SFML to draw the diffusing green blobs. Sorry for not putting up a list of resources near the end.

Fran's report was first published on *BuontempoConsulting* on 6 February 2019 at https://buontempoconsulting.blogspot.com/2019/02/cpponsea.html

I attempted to go to Guy Davidson's 'Linear algebra' talk next, but the room was packed and I was a bit late. I heard great things about this. In particular, how important it is to design a good interface if you are making libraries.

My final choice was Clare Macrae's 'Quickly testing legacy code' [21]. This was my unexpected hidden gem of the conference. She talked about approval testing [22]. This compares a generated file to a gold standard file and bolts straight into googletest or Catch. It's available for several other languages. It generates the file on your first run, allowing you to get almost anything, provided it writes out a file you can compare, under test. Which then means you can start writing unit tests, if you need to change the code a bit. Changing legacy code to get it under test, without a safety hardness is dangerous. This keeps you safer. Her world involves Qt and chemical molecules visualisations. These can be saved as pngs, so she can check she hasn't broken anything. She showed how you can bolt in custom comparators, so it doesn't complain about different generated dates and does a closer than the human eye could notice RGB difference. Her code samples from the talk are available [23]. I've not seen this as a formal framework before. Her slides were really clear and she explained what she was up to step by step. Subsequently twitter has been talking about this a fair bit, including adding support for Python3.

Matt 'Compiler Explorer' Godbolt gave the closing keynote. Apparently, the first person Phil Nash has met whose first conference talk was a keynote. He also had some AV issues (Figure 3).

If you've not encountered the compiler explorer before, try it out [17]. You can chose which compiler you want to point your C++ code at and see what it generates. You need a little knowledge of the 'poetry' it generates. More lines doesn't mean slower code. His tl;dr; message was many people spread rumours about what's slow, for example virtual functions. Look and see what your compiler actually does, rather than stating things that were true years ago. Speculative de-virtualization is a thing. Your compiler might decide you only really have one likely virtual function you'll call so checks the address and does not then have the 'overhead' it used to years ago. He also demonstrated what happened to various bit counting algos – most got immediately squashed down to one instruction, no matter how clever they looked. How many times have you been asked to count bits at interview. Spin up Godbolt and explore.This really shows you need to keep up to date with your knowledge. Something

Figure 3

GitHub's Crazy Contribution-Graph Game

Silas S. Brown does a one-year streak.

For years I had been happy to post my public code on my personal home page, which is mostly hand-coded in HTML and has been in my `public_html` directory since before the turn of the century (although the external URL had to change a couple of times, which makes things harder for Internet Archive's Wayback Machine). Google seems to have no problem finding my pages.

But nowadays some people don't even know what a personal home page is, because they're so used to individuals' Internet presences being confined to large recognised social-media platforms, like Facebook (reputedly for people who like to mess around and air family feuds in public, and I wouldn't dare take out an account there and subject my creations to the whims of whatever crazy algorithm they come up with next, but that's just my opinion), Twitter (reputedly not as bad as Facebook but sometimes I wonder), and LinkedIn, which is supposedly 'for work' – it's not blocked in China and they have a Chinese name 'LingYing' which means 'leading heroes' – but I for one haven't yet seen any real benefits from being on it, apart from occasionally being able to get back in touch with people whose email addresses have changed; I opened a private account on it to write a recommendation for someone who asked for one, and since then I've accepted connection requests from people I know (unless they got lost in the spam), but any actually-interesting news they post tends to be drowned in spammy 'work is great' articles generated by the platform itself, which feels like it's trying to control my life instead of help.

Give me my `public_html` over that any day, apart from the small detail of some people not even knowing what I'm talking about when I say something is on my home page. I was once asked if 'home page' means 'blog', and I said 'sort-of', although the closest I've come to running it as a blog is to write an ugly shell script that automatically lists all page titles and makes an RSS feed; I only did that as a convenience for when I needed to access my notes quickly from a mobile device, and it doesn't track when pages are corrected. And the only way to comment is to email me. Very old-school. (An old classmate predicted I'll eventually be waving a walking-stick shouting "young people don't know how to write HTML!")

SILAS S. BROWN

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

C++ On Sea 2019 Trip Report (continued)

that was true ten years ago may not longer hold with new compiler versions. Measure, explore, think.

There was a lovely supportive atmosphere and a variety of speakers. People were brave enough to ask questions, and only a few people were showing off they thought they knew something the speaker didn't.

Thanks to Phil for arranging this conference. ■

References

[1] Arne Mertz (2018) 'ACCUConf 2018 Trip Report', published on *Simplify C++* on 25 April 2018: https://arne-mertz.de/2018/04/accuconf-2018-trip-report/

[2] *C++ On Sea*: https://cpponsea.uk/

[3] #include <C++>: https://www.includecpp.org/

[4] *C++ On Sea* 2019 floorplan: https://cpponsea.uk/rooms/

[5] *C++ On Sea* 2019 schedule: https://cpponsea.uk/schedule/

[6] Kate Gregory (2019) 'Oh The Humanity!', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=SzoquBerhUc

[7] Juan Pedro Bolivar Puente (2019) 'Postmodern Immutable Data Structures', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=y_m0ce1rzRI

[8] Adi Shavit (2019) 'What I Talk About When I Talk About Cross Platform Development', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=dU3IgHrAmFA

[9] Patricia Aas (2019) 'Deconstructing Privilege', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=ZYvyO27uMCU

[10] Björn Fahller (2019) 'Programming with Contracts in C++20', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=Dzk1frUXq10

[11] Arne Mertz (2019) 'Learning and Teaching Modern C++: Challenges and Resources', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=fKCwDg0vd18

[12] John Shearer (2019) 'Sailing from 4 to 7 Cs', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=1Eh-uEzpikw

[13] Arne Mertz (2016) 'The '4C' Dev Environment', published on *Simplify C++* on 10 August 2016: https://arne-mertz.de/2016/08/the-4c-development-environment/

[14] Matt Godbolt (2019) 'What Everyone Should Know About How Amazing Compliers Are', presented at *C++ On Sea* 2019: https://www.youtube.com/watch?v=w0sz5WbS5AM

[15] Presentations from *C++ On Sea* 2019: https://www.youtube.com/channel/UCAczr0j6ZuiVaiGFZ4qxApw/videos

[16] Start Learning C++: https://www.cpplondonuni.com/

[17] Godbolt compiler explorer: https://godbolt.org/

[18] Barney Dellar (2019) 'Strong Types in C++', presented at *C++ On Sea* 2019: https://docs.google.com/presentation/d/17WRh90KWWqobRX1miUFCnRwaJJXxqvlcu-t3DvFVHOA/edit#slide=id.p

[19] Nicolai Josuttis (2017) *C++17: The Complete Guide*, available from http://www.cppstd17.com/

[20] Frances Buontempo (2019) Genetic Algorithms and Machine Learning for Programmers, available from https://pragprog.com/book/fbmach/genetic-algorithms-and-machine-learning-for-programmers

[21] Clare Macrae (2019) 'Quickly Testing Legacy Code', presented at *C++ On Sea* 2019: https://www.slideshare.net/ClareMacrae/quickly-testing-legacy-code

[22] Approval testing: https://github.com/approvals

[23] Clare Macrae (2019), code samples from 'Quickly Testing Legacy Code', presented at *C++ On Sea* 2019: https://github.com/claremacrae/cpponsea2019

I did, however, start checking things in to a SourceForge account when someone made me a subdirectory in his project and invited me to check in my stuff there, saying it would be convenient for people to see what I've changed before they update. I had previously thought "people can just check my home page to see if I've changed the version number since they last downloaded, and if they want a more detailed comparison they can run diff themselves", but I supposed a repository does give added convenience even when there's only one contributor, plus having my code in version control makes me less frightened of deleting old parts instead of commenting them out. I did not, however, trust the SourceForge platform enough to be willing to 'live' there: my home page stayed where it was, with the repository being very much a secondary 'mirror' of my code without the HTML explanations that accompany it. I have an automatic script to fetch the latest code from my home page and update the repository, although unfortunately that means most of my commit summary messages just say 'update', which is not very informative. (See cartoon at the top of the page. It is from the *xkcd* comic, which can be found at https://xkcd.com.)

GitHub for the boardroom?

After SourceForge descended into an advertisement-ridden mess (fake Download buttons became particularly obnoxious), most projects switched to Git and GitHub, along with its later competitors GitLab and BitBucket (now owned by Atlassian). There are of course other Git hosting providers, like Assembla and Beanstalk, but they don't tend to have free tiers in their pricing models and are therefore less popular in the 'open source' world. I had already used Git on some closed-source projects (mostly if others I was working with wanted to use Git, or as a convenience to track old versions locally), but I hadn't bothered hosting anything on GitHub.

Then in late 2017 I was working part-time for a startup (it had been spun off from someone's university research and I ended up getting involved), and someone there said he was trying to pitch the company to investors and said those investors wanted to see employees' personal projects on their GitHub profiles. It seemed he thought the company's work projects might not be sufficiently impressive so better show off the personal projects of employees as well, and it must be GitHub because that's the only platform the investors had heard of. So I obligingly ported my Subversion directory into a bunch of GitHub projects and expanded my 'update' scripts to mirror to both. (Some projects are only on Git as I didn't bother adding them in to the Subversion script, but anything I previously had on Subversion continues to be updated there as well.)

The company sold soon after, but I kept going with GitHub in case it was convenient to anyone, and later added GitLab and BitBucket 'just in case' after GitHub was acquired by Microsoft and the general atmosphere of apprehension set in. Git lets you add multiple origins to a repository so that a single 'git push' will update multiple remote servers (see Listing 1) so I'm able to keep the proverbial foot in all three camps for now.

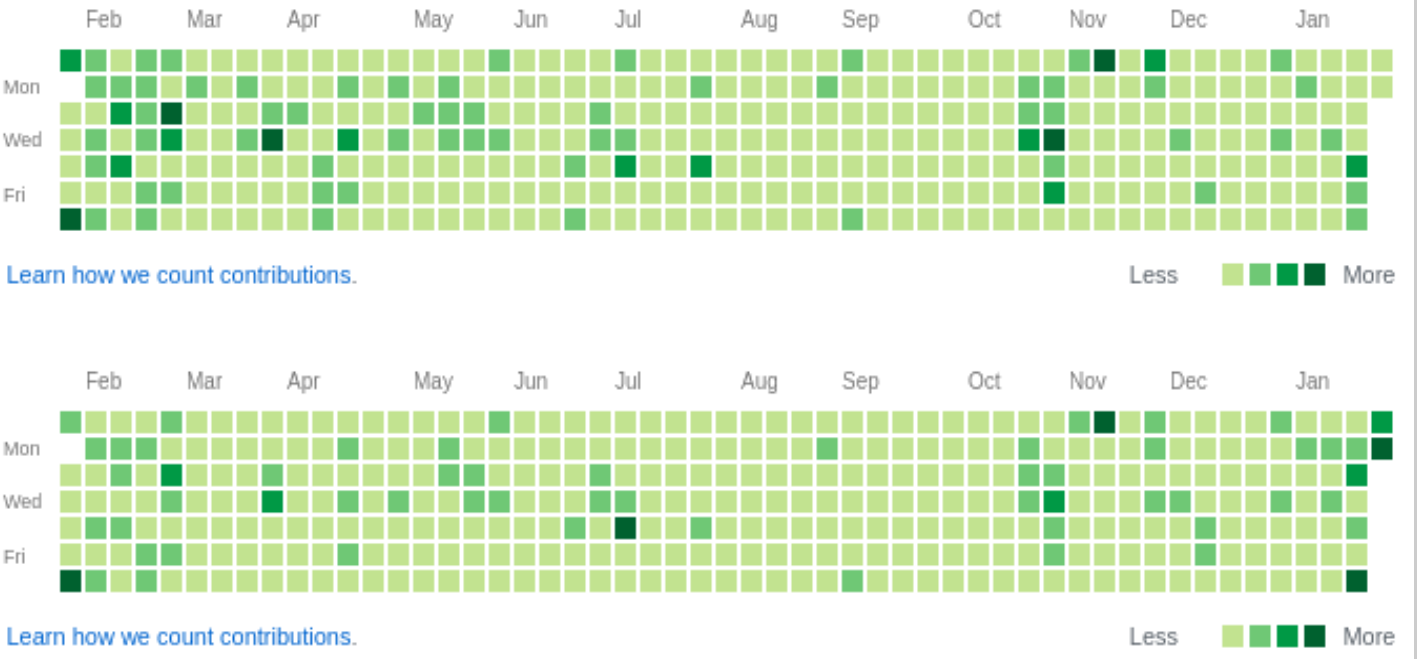
However, I couldn't help feeling a tiny little bit worried about what he'd said – the thought of those investors, who don't know what a home page is, trying to judge an individual's contributions on GitHub. If they don't know what a home page is, would they really be able to look at code and see if it's any good? Or are they just looking at numbers, metrics like Figure 1?

Figure 1 shows my GitHub contribution graph at the time of writing. It can be found at http://github.com/ssb22 when the site is set to Desktop mode (they don't show these graphs in Mobile mode). The top graph is with GitHub set to show public commits only; the bottom is with GitHub set to show private commits as well. If you look carefully, you will notice that some squares which are shaded darkly on the 'public commits only' graph are actually shaded less darkly on the graph with private commits as well: it seems the scale of the shading is adjusted to reflect the maximum number of commits in any one day over the period shown, so a high number of commits on one particular day can make the rest of the

Listing 1

```
git remote set-url origin --push --delete .
git remote set-url origin --push git@github.com:$USER/$REPO.git
git remote set-url origin --push --add git@gitlab.com:$USER/$REPO.git
git remote set-url origin --push --add git@bitbucket.org:$USER/$(echo $REPO|tr A-Z a-z).git
```

Figure 1



graph look flatter for a year. It's a pity they didn't take more of an average when calibrating this scale; I would have coded 'the median of all non-zero days' if it had been my job (although I would probably have opposed the implementation of the graph in the first place).

Thankfully, I have so far not been desperate enough to need to 'prove' myself to some ghastly metric-oriented boardroom who don't know the faintest thing about code quality and only look at graphs like this, but, just on the off-chance it ever comes to that, I have felt a certain amount of pressure to commit code on 'as many different days as possible', which is the thing rewarded by GitHub's contribution-graph game. Never mind the size of each commit, and certainly don't mind its quality, just make sure your commits are spread over as many different calendar days as possible.

Playing the graph game

As you can see, it has now been a full year since I last missed a day. Some of that was due to my having multiple commits ready to go but choosing to delay them so as to have 'something for tomorrow' and 'something for the next day' as well. Perhaps 'delayed commits' are an unintended consequence of this game. At one point I went on a 5-day trip and had separate commits lined up for each of those 5 days, which I executed via an SSH client on a mobile phone. I haven't yet stooped to scheduling unattended commits in a cron job: what if I die in an accident and somebody notices I seem to be committing from beyond the grave? I care about people too much to risk playing with their emotions like that.

GitHub's graph can be tricked by fiddling with the dates in your commit log, and there are scripts out there to create fake repository histories whose commit dates cause pictures to appear in the GitHub graph (although the nature of GitHub's shading calibration means you'd better not combine such a picture with too many other commits in the same year). The newer company GitLab makes fake histories harder to achieve, because the only timestamps that matter to GitLab are the ones made by GitLab's servers when accepting your push requests. So your script would have to run over many months of real time instead of back-dating hundreds of commits in one sitting. But I don't know if the decision-makers who are naive enough to judge a coder by these graphs will know the difference between GitHub and GitLab anyway. They might one day learn it, depending on how the reaction to Microsoft's acquisition of GitHub pans out in the coming months. (BitBucket does not have a graph, at least not yet.)

Figure 2 shows my GitLab contribution graph, which has a different (fixed?) shading scale; the large space on the left is because GitLab does not graph contributions until you push them to GitLab servers (which I didn't do before the Microsoft acquisition of GitHub), and the smaller gaps are due to 'UTC versus British Summer Time' differences, on days when I pushed to both GitHub and GitLab shortly after midnight BST without realising that GitLab was working on UTC.

Many of my commits have been real coding or bug-fixing, but some were admittedly refactoring, minor corrections to comments or help text, or (especially) updates to the dictionary I use for Chinese text parsing, which is pushing the meaning of 'coding' somewhat. I thought I was going to have to break the streak in October when my wife and I went on a 9-day trip to see friends in Berlin, but our host had Wi-Fi and I still fiddled with that dictionary on my phone while I was waiting for people to get themselves ready and so on. (Being around native speakers does increase the probability that I'll be exposed to something that prompts me to edit that dictionary.) As I write this, we are about to go on a 1-month trip to see in-laws in Hong Kong and Taiwan, so it is almost certain I will have to break the streak this time, but if by the time this article is printed you should see my graph does indeed show commits throughout February, then it probably means I've been fiddling on my phone again. (That doesn't sound good does it.)

It would, of course, be possible to keep a streak going for a month or more entirely automatically, if I were happy to resort to scheduling unattended commits, and if I were happy to artificially 'shred' some contribution into dozens of small commits to stretch it out over a month. But although I have previously delayed a commit or two for (at most) a handful of days, I'm not dishonest enough to take it to that extreme, plus I don't like to think of holding back finished work too long, and I certainly don't want to schedule automatic commits that do something completely meaningless like add a random number to a comment. I do, however, point out the possibility so as to draw further attention to the uselessness of this metric.

Private repositories and web hosting

GitHub private repositories became free for small teams in 2019, like the non-academic version of BitBucket (BitBucket users with university email addresses can have unlimited private collaborators, just like all GitLab users, but BitBucket non-academic and the new GitHub plan limit the team size of free private repositories). GitHub is so far the only platform to introduce the option of counting private commits on your public graph. I did start using GitHub's free private repositories as a convenience to access personal private projects (things I can't distribute because they include other people's copyrighted material like song lyrics); third-party hosting always carries some small risk that your files will be stolen in a break-in, but provided there is nothing TOO sensitive in there, having them on GitHub or similar can be useful: I can more easily access them from multiple systems and it's also an off-site backup that's slightly less fiddly than putting a USB stick in my pocket, as long as the project I want to host fits in the limits (1G max and 100M max per file, unless you install Large File Storage extensions). There is now a script in my 'bits-and-bobs' repository called `gitify` which can help to 'git'-ify any files you have not currently under version control: it creates one commit for each file and dates each commit to the timestamp of the file, which is probably the only reasonable way to port timestamps to Git when they might be significant. But I carried on making at least one commit per day to a public repository as well, in case I ever have to delete the private repositories (which will revert the squares on the graph), or in case I ever have to use GitLab's graph instead of GitHib's.

All three platforms also offer to host static HTML files for website serving, although GitHub's free tier allows this only from public repositories. GitLab and BitBucket allow it from private repositories too, even on the free tier; serving pages from a private repository means someone needs to know a page's URL before they can retrieve it, and also makes it a bit less easy to fork your site. All platforms default to enforcing HTTPS although GitLab lets you turn this off. The simplest setup is to create a repository called username.github.io, username.gitlab.io or username.bitbucket.io (substituting username as appropriate); GitLab also requires a file called `.gitlab-ci.yml` (see Listing 2 for a simple example).

If you do choose to create multiple mirrors of your website then you would be advised to insert markup like **`<link rel="canonical" href="http://...">`** to indicate each page's 'canonical' address so that search engines are less 'confused' by the duplication.

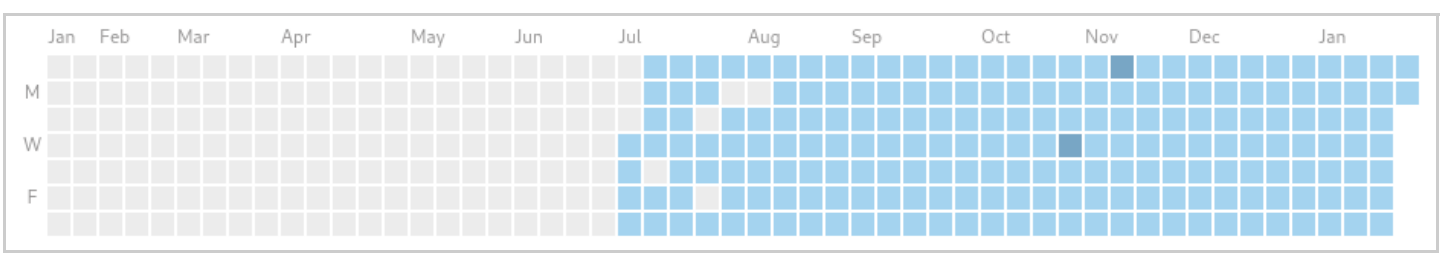
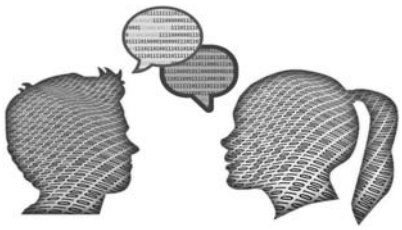


Figure 2

Code Critique Competition 116



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

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

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

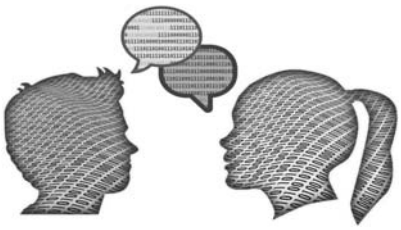
Last issue's code

I am trying to write a generic 'field' that holds a typed value, optionally with a 'dimension' such as 'kg'. One use case is to be able to hold attributes of various different types in a standard collection. I want to be able to compare two attributes to see if they're the same (and maybe later to sort them?) – two simple fields are only comparable if they are the same underlying type and value and two 'dimensioned' fields need to be using the same dimension too. I've put something together, but I'm not quite sure why in my little test program I get true for comparing house number and height:

```
Does weight == height? false
Does weight == house number? false
Does house number == height? true
Re-reading attributes
Unchanged
Reading new attributes
some values were updated
```

Can you help with the problem reported, and point out some potential flaws in the direction being taken and/or alternative ways to approach this problem?

The code is in Listing 1 (`field.h`) and Listing 2 (`field test.cpp`).



Listing 1

```
#pragma once
#include <memory>
#include <string>
// Base class of the 'field' hierarchy
class field
{
protected:
  virtual bool
  equality(field const &rhs) const = 0;
public:
  // convert any field value to a string
  virtual std::string to_string() const = 0;

  // try to compare any pair of fields
  friend bool operator==(field const &lhs,
    field const &rhs)
  {
    return lhs.equality(rhs);
  }
};
// Holds a polymorphic field value
struct field_holder : std::unique_ptr<field>
{
  using std::unique_ptr<field>::unique_ptr;
  bool
  operator==(field_holder const &rhs) const
  {
    return **this == *rhs;
  }
};
```


ROGER ORR

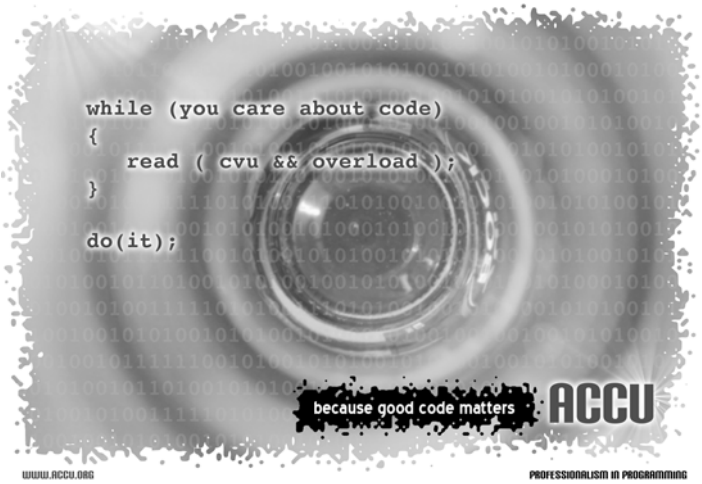
Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

GitHub's Crazy Contribution-Graph Game (continued)

Actual results?

A few months ago I was emailed, out of the blue, a job offer from a startup who said they liked my GitHub activity. But they wanted to pay me equity instead of salary; I was not sufficiently convinced of that company's potential, so I told them I saw it as too much of a risk. I was polite enough to include a couple of suggestions about their idea in my reply, but they didn't write a second time.

But I really hope I won't ever have to use a silly GitHub metric to impress a potential employer. I'd much rather leave them with a printed copy of *CVu* or *Overload* with one of my own articles in it. I'm happy to say this led to the last company becoming a corporate ACCU member themselves, although I was recently disappointed to discover that someone cancelled the ACCU subscription during the process of the company's being acquired by Oracle (and I don't understand the Oracle corporate structure enough to know what to do about this) but at least the subscription did some good while it lasted. ■



Listing 1 (cont'd)

```
// Implementation in terms of an underlying
// representation
template <typename Rep>
class field_t : public field
{
public:
  field_t(Rep const &t) : value_(t) {}
  std::string to_string() const override
  {
    return std::to_string(value_);
  }
protected:
  bool equality(field const &f) const override
  {
    // compare iff of the same type
    auto p =
      dynamic_cast<field_t const *>(&f);
    if (p)
      return value_ == p->value_;
    else
      return false;
  }
private:
  Rep value_;
};
// Add a dimension
#include <typeinfo>
template <typename dimension,
  typename Rep = int>
class typed_field : public field_t<Rep>
{
public:
  using field_t<Rep>::field_t;
  std::string to_string() const override
  {
    return field_t<Rep>::to_string()
      + dimension::name();
  }
protected:
  bool equality(field const &f) const override
  {
    // compare iff the _same_ dimension
    return typeid(*this) == typeid(f) &&
      field_t<Rep>::equality(f);
  }
};
```

Listing 2

```
#include "field.h"
#include <iostream>
#include <map>
// Example use
struct tag_kg{ static const char *name()
  { return "kg"; } };
struct tag_m{ static const char *name()
  { return "m"; } };
using kg = typed_field<tag_kg>;
using metre = typed_field<tag_m>;
using attribute_map = std::map<std::string,
  field_holder>;
// Dummy method for this test program
attribute_map read_attributes(int value)
{
  attribute_map result;
  result["weight"]
    = std::make_unique<kg>(value);
  result["height"]
    = std::make_unique<metre>(value);
  result["house number"]
    = std::make_unique<field_t<int>>(value);
  return result;
}
int main()
{
  auto old_attributes = read_attributes(130);
  // height and weight aren't comparable
  std::cout << "Does weight == height? "
    << std::boolalpha <<
     (old_attributes["weight"] ==
      old_attributes["height"])
    << '\n';
  // weight and house number aren't comparable
  std::cout << "Does weight == house number? "
    << std::boolalpha <<
     (old_attributes["weight"] ==
      old_attributes["house number"])
    << '\n';
  // house number and height aren't comparable
  std::cout << "Does house number == height? "
    << std::boolalpha <<
     (old_attributes["house number"] ==
      old_attributes["height"])
    << '\n';
  std::cout << "Re-reading attributes\n";
  auto new_attributes = read_attributes(130);
  if (old_attributes == new_attributes)
  {
    std::cout << "Unchanged\n";
  }
  else
  {
    // update ...
    std::cout << "Some values were updated\n";
  }

  std::cout << "Reading new attributes\n";
  new_attributes = read_attributes(140);
  if (old_attributes == new_attributes)
  {
    std::cout << "Unchanged\n";
  }
  else
  {
    // update ...
    std::cout << "some values were updated\n";
  }
}
```

Critique

Balog Pál < pasa@lib.hu >

"I am trying to write a generic 'field'..." Oh, my. A picture forms in my head, that OP shows this to his deskmate for some pre-review advice. And he gets something like:

> Seriously? You want to run it through Pál? Well, bring a jetpack, as he'll throw you outta window. Also I see you went ahead and wrote code, didn't you hear enough times to ask for a design review on ideas before you coding, especially for a new class hierarchy? Please just wait a moment while I fetch some more audience and a bucket of popcorn.

And right he is: 'properties', 'attributes' and 'generic fields' are for me like muleta to a bull. Probably because in my working experience with writing applications I mostly saw it creating mess and more mess. While it stays a popular demand, and is 'backed up' by claims that 'other popular languages' have those with native support therefore it must be good. I wish the world worked that way.

Certainly exiling people and asking questions later is not my thing, I more like flood them with questions until they maybe decide to exile themselves... Unfortunately this format is not interactive, and building a strawman just to burn it down is not really fun. Or preaching. So instead

I'll try to create some conversation that will supposedly reveal my general concerns.

First let's take a detour for that missed design review. OP came to me with the idea before writing the code, but already though up those classes and templates. Obviously the discussion starts with the motivation, the use cases, the real problem we want to solve (really too bad not a word on that is present in the entry), but let's stash that for a moment, and jump to technicals. My opening suggestion would be like:

> So you need a 'tuple' of { type, value, dimension }. Where type could be a `string` or `enum`, dimension a `string`, empty covering the no-dimension case. value is more tricky but you can start with `string`. Or `int`. Later a variant of candidates. What else we need? standard collection: check. Equality? if it's an actual `std::tuple`, isn't the factory version just do what you need? And maybe even just {value, dimension } is enough, you're somewhat moot on the 'generic' part, IME just the type of storage unit is not enough, your dimension may cover that information .

I look up and notice the expression on OP's face. It radiates:

> WHAT? You can't be serious! I imagined up a full framework, gadgets, templates, a hierarchy. Using a tuple or a simple struct instead of all those almost makes me obsolete. I definitely can't brag about that to my girlfriend… A puny `std::pair` and full stop? No, this just can't be right. I need to shred this blasphemy^Widea to pieces right away. This just can't possibly be a solution, not even a candidate for initial brainstorming.

Before going on, I suggest the reader take a step back and actually consider this. Yes, it has a plenty of shortcomings both for OO theory and some imagined use practice, but does it solve what was revealed as a problem? Would it pass the test cases? How much opportunity it leaves for bugs in that area? I honestly think it would work okay-ish, so it leaves us balancing its potential problems in other parts of the use case against the UB (sorry for the spoiler) and broken `==` in the presented alternative let alone its complexity. And it also open for improvements. Back to the conversation:

OP: That's not exactly what I had in mind. How about making the field an interface and create subclasses.

Me: Sure, that can be done, but looks plenty of work off the bat just for the hierarchy alone. And you looked for a standard collection for it, I think we still didn't include polymorphic ones. I have some in my support lib, but that does not qualify as solution. You need to rack up some benefits to offset the costs.

OP: I think we're wasting a lot of memory. In my way the dimension is not stored, but part of the class behavior. So that's an extra string per instance. And if we use variant that will be heavier than just one actual type. And while at it, the user could just assign a different dimension or switch the variant to a different type unlike with subclasses.

Me: The second part is granted for the most primitive implementation. Looks trivial to cover if we use an actual class with those members private and function to change the value, while keep the dimension immutable after construction. Still way less complexity than a subclass.

Me: If dimension is a COW string, `const char *` or `enum`, that is not much storage. The variant depends on the data types you will need to disclose at some point, till then let's assume string, `long long` and `double` is enough for all practical uses. that is like a dozen extra bytes over a plain `int`. While in the alternative we have a vtable pointer and a heap block. I'd call it a tie at best. But let's pretend we have a few hundreds of bytes waste. Is that really a significant factor?

OP: Of course it is. Say my users have a million instances, that would amount to hundreds of MBs.

Me: Hmm, we really can't postpone the intended use really. What I pictured is not compatible with this claim. So in what part of the program you want those generic fields and for what?

OP: Why, everywhere! Isn't that cool and generic? A `Field<int>` is more sophisticated than a plain `int`, so why not replace our plain `int`s? Also we read a database in the program, it looks natural to fetch the record into such fields. (*Here OP notices a murmur from the audience and some looks at the window…*)

Me: (Despite other rumors I just do not launch nukes before going through the 5 whys, so must let down the audience's expectations... let's dig a little deeper.) Was that really the original idea behind this?

OP: Well, not really, I just wanted an easy way to write user interface. Kinda like in the Visual Studio 'Properties', that is just a 2-column listview with editable value. I could use that instead of creating a custom dialog for many of our classes. Also much easier to maintain if we change the data. I can just map data members to those fields and the UI could edit the values independently of the client, that can then get it back and consolidate. But is we have this component created why not use it for many other things in our model? (*A few people head for the door.*)

Me: Okay, I appreciate the enthusiasm but let's focus on the real task. Sleep on this other part and we can come back to it at a later time unless you happen to change your mind. So how many of those fields you intend to have in the interface at any one time?

OP: Maybe a dozen. But... suppose I also want to present a 'browse' for database tables like I saw in the museum there was in dBase clones? If the table has a million records, we're back to the waste.

Me: The browse only shows a few dozen lines that fit the screen vertically. If for that you read up all the million-to-unlimited amount of records without consideration, field or native, I'm sure someone will call that idea FU^H^Hsuboptimal.

OP: I concede the waste. But how about extensibility? For my solution the user can create fields for new types. All independently of everyone else. Especially not touching a central component. Do you want everyone just add extra elements to the variant if we go with that?

Me: See, that is a good valid point. With no stock answer. Say we did not include `float`. And 15 client classes would want to use that. Is it better to all of them creating their own or just one adding it to central. A really problematic case could be if a client has its specific type that is not available on the kernel level (or where the central facility lives.) If that is expected subclasses have an edge. However alternatives still exist, à la 'strategy' pattern. I think just adding a `std::function` into the variant could solve that problem. But I'm not sure I would happy with that. For a sensible verdict we need real data, real information to work, generics and speculation will NOT result is good software. You need to lay out the actual aimed use cases, and we will focus on that, selecting the solution that fits that case.

OP: But what if we create a solution for the presented specific problem and next day someone adds a class that does not fit? I think we should think ahead and work a future-proof solution.

Me: Over my dead body. ☺ No, Sir, while I'm in charge here that is just not happening. But feel free to apply for my position. And put your own ass in the fire explaining that we missed several deadlines struggling to solve for some made-up problems from an imagined future instead of what is needed by real clients here and now. And carry the burden of all the excess complexity. My good record of deploying working software in the field is mostly tied to strictly following KISS. Also having seen a lot of cases of the speculative programming. Thrown away as soon as actual change requests arrived, that almost always had some nuance not expected. I admit it may be confirmation bias, but enough cases happened.

And after this detour that possibly could prevented the whole accident, let's look at the written code and figure out the problems. I'm sure some other entries will use the compiler's help, I keep just reading the code and hope to understand it. Some conclusions may be inaccurate, if so I expect Roger to insert a warning.

A fast look through the code makes me say it is interesting and looks much better than many first swings at making and collecting a polymorphic type. Especially old times before smart pointers. But there are some fishy points. Before anything else, the elephant in the room:

We have abstract class field and store its subclasses with `unique_ptr`. A sound idea, that would be one of my first picks is working from scratch.

But the class lacks a virtual dtor. It's a major goof on OP's part, almost on the level of accidental deletion of the related line after the last compilation and before submission. The guideline of starting a class hierarchy with virtual dtor is almost as old as C++ itself, and I believe most compilers warn about the deviation for maybe a decade. This should not be missed or dismissed. There are some special cases when you can get away without but those are better left for pure trivia. (And the only acceptable motivation to not have virtual dtor at the root is when it has no virtual functions either...) Worth mentioning, that if `shared_ptr` was used here, it would actually work even with this situation, as it can capture the correct deleter at the creation if the proper incantation is used. But `unique_ptr<field>` will always issue plain delete on `field*` static type, that in our program always have a different dynamic type and no virtual dtor, that is UB. Until the fix we should not treat any output or other observation seriously. The fix is to either add it with `=default` directly or follow my practice to inherit from an empty base class `VirtualDtor` that has just that and the other spec functions defaulted – because that defaulted dtor still counts as user-declared and kills our move ctor and assignment. And if we declare those with default that kills the copy ctor and assignment. Not nice. As field is empty at the moment it may look redundant but I rather consistently replaced all my dtors with that workaround and sleep well.

After steering back to defined behavior, the tests still fail. But I suggest another small detour to improve that test first. We're testing a function that returns `bool`. So it's got a 50% chance of looking right with a broken implementation. Even if we forgot the proxy implementation that just returned `false` no matter what. As a matter of fact, having that the actually present tests would just pass with flying colors. Oh, Oh. I'd add a function that at least covers what it can, taking (a, b, expected) looks that `a == a` and `b == b` are `true` and `a == b` and `b == a` result `expected`. My estimate is that the last line would produce different results with the arguments switched. Then we could use some more cases with a different base type.

Now finally let's look at those `op ==`s. We see two of those, a nonmember-friend in field and a member in holder. I'm somewhat baffled on this inconsistency, and ask OP to explain. Beyond that the implementation looks correct. In field it defers to virtual function. Important thing that we have no `op ==` defined for subclasses that could probably ruin the show. In holder defers to the held type. With C++20 we could add a precondition contract to be nonempty and expecting the user to comply. Some designers would object that and instead require check for `null` and `report ==` accordingly. With the latter splitting to two camps, those returning `false` for `null` on either side and those returning `true` if both `null`s. All three variants make sense for certain philosophy, that I leave at that till our examples have no empty holders. But OP must make a choice and document it.

Following up in `equality()`. That is pure in field (good), and defined for `field_t` and `typed_field`. With more curious discrepancy: using `dynamic_cast` and `typeid`. Again OP to explain why the different approach while verbally seeking the same semantics.

Vanilla uses `dynamic_cast` on the rhs to its own type. if it succeeds, then comparing the value otherwise going `false` for incompatibility. I say that matches the aim.

For dimensioned ,I'd expect the same approach really. I see no good reason to drag in `typeinfo`, and it is a thing I consider a code smell in its own right. I had to look up how it is supposed to work, turns out it can be invoked for the reference and fetches the record for the most derived type. and `==` is supposed to work too. Is this different from the `dynamic_cast` version? As long as we have just the presented classes I think not. But `typed_field` is not marked `final`, and if we subclassed that we could observe a difference. What is probably a bad thing, say I subclass to have a more decorated version of `to_string()` and nothing else, I'd expect the same values for `==`, but would get `false` instead. But for the current state I say this is correct too.

So where we are? We fixed the UB, have correct implementations of ==, and yet the test fails. What did we leave unchecked? I guess the wimps just single-step in the debugger till reach some unexpected state. I just have to make predictions: for the 3rd line we eventually get to `field_t::equality` with lhs being vanilla and rhs being a metre. `dynamic_cast` gives us its base class `field_t<int>` and finds the same value, returning the `true` we see. (With lhs and rhs switched we'd be in `derived` and looking at different `typeid`s and a `false` result...)

So, are we ready to declare the verdict that `dynamic_cast` sucks and switch to `typeid` here too? I hope not, before thinking again, especially as we passed this function as correct recently. Indeed, `dynamic_cast` is only guilty in working by its specs instead of our wishes. Just as those pesky computers keep following the instructions (unless we drive to UB-land that is). For the case `dynamic_cast` works fine for derived-to-public-base-class. And that is the relation we defined when we said

```
class typed_field : public field_t<Rep>
```

In other words, we said that `typed_field` IS-A `field`. That it stands fine in any place vanilla would be fine. That is a clear contradiction with the equality-test where the substitution (semantically, not technically) slices the dimension and puts us on the incorrect true path. `typed_field` is IS-A vanilla field for value-holding purpose but NOT for equality.

This is very similar to the basic OO problem of 'is Rectangle a Square or is Square a Rectangle or neither in OO design' that surprised may novices (probably quite close to all!). OO and hierarchy design is riddled with these kind of traps. The simple-looking ancient guideline 'make sure all your public inheritance means is-a' is not trivial to follow.

How to fix the situation? Well, I rather leave it to the OP, after all he still has the jetpack unused, and next time he can add this lesson to the for-hierarchy arguments and how much easier are those to handle for the clients. They may be. Or may open one more subtle opportunity to naively making a subclass and breaking some part of is-a-ness. (Or the other way around, as mentioned at the `typeid`-using version).

OP mentioned he will want sort too eventually. That will need `op <`, may turn out even trickier. Before someone gets the wrong conclusion that I'm pushing for the flat version and imply it is a piece of cake just like that: it's not the case. But it definitely dodges lots of bullets. And once we really know what we are chasing, and is still (!) on the table – in a contest on less bugs initially and over lifetime my bets would go there. And if your experience is different, consider to make the world a favor by teaching good and working OO and hierarchy-design practices. (Also please write an article or just PS reaction...)

I could nit-pick on the rest of the code, but will not do so, as I believe these are the really important lessons and better not water them down.

Commentary

This issue's critique was somewhat long: while the questions raised by the code did seem to be interesting I think that my inability to shrink the code very much resulted in readers feeling a bit overwhelmed. (This issue's critique is much shorter...)

Pál does a good job of covering some of the problems with the tricky question of equality and class hierarchies. The problem is that it is easy to break the (mathematical) definition of equality.

The first property is *substitution*: if a == b then f(a) == f(b). This maps nicely into Liskov Substituability Principle ("LSP") and the IS-A principle.

The next set of properties are the *reflexive*, *symmetric* and *transitive* properties. These are:

a == a

if a == b then b == a

if a == b and b == c then a == c

These are much harder to manage in a class hierarchy, as this example demonstrates.

Implementations usually adhere to the *reflexive* property. This critique fails with the *symmetric* property:

If a is a `field_t` and b is a `typed_field` then

`a == b` calls the equality method in `field_t` that checks that b IS-A `field_t` (and then compares the values) whereas

`b == a` checks that the types of a and b are the **same**.

The safest change is to make the implementations consistently verify their arguments have the same type: but this, as Pál noted, means that substitutability is broken since you now *cannot* substitute a `typed_field` object for a `field_t` one. This is a case where the question must be pushed back to the original writer of the code as they need to decide which is the design best matching their use cases. Alternatively, the use of `operator==` and the method `equality` should be changed to something without the same implicitly assumed set of properties.

The test code in the code example is quite minimal: the first set of tests checks various objects all of which have the same value (130) – but does not do an exhaustive set of tests – and subsequent tests simply test equality and inequality of a set of objects.

It's hard to write good test cases; thought need to be given to what possible failure and success modes there might be and therefore what test data should be used. For example, in the case of an `operator==` we should ensure some tests are written for each of the four main properties listed earlier in the commentary.

The winner of CC 115

There was only one critique this time, but I liked Pál's approach in concentrating on the higher level issues so I am happy to award him the prize for this issue's critique.

Code Critique 166

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

I'm trying to extract the count of letters in a file and while the program itself seems to work OK, I'm not getting the debugging output I expected. I was expecting to see a debug line of each letter being checked during the final loop, but I only get some of the letters shown.

For example, using `sample.txt` that consists of "This is a some sample letter input" I get:

```
letter sample.txt
Seen T
Seen h
Seen n
Seen o
Seen r
Seen u
Commonest letter: e (4)
Rarest letter: T (1)
```

Why are only some of the letters being shown as `Seen`?

Can you solve the problem – and then give some further advice?

Listing 3

```
#include <iostream>

namespace
{
  class null_stream_buf
  : public std::streambuf
  {
  } nullbuf;
  std::ostream null(&nullbuf);
}

#ifdef NDEBUG
#define WRITE_IF(X) if (false) null
#else
#define WRITE_IF(X) ((X == true) ? std::cout
  : null)
#endif
```

Listing 4

```
#include "logging.h"

#include <fstream>
#include <map>

// get the most and least popular chars
// in a file
int main(int argc, char **argv)
{
  WRITE_IF(argc < 2)
    << "No arguments provided";
  if (argc > 1)
  {
    std::ifstream ifs(argv[1]);
    std::map<char, int> letters;
    char ch;
    while (ifs >> ch)
      ++letters[ch];

    std::pair<char, int> most =
      *letters.begin();
    std::pair<char, int> least =
      *letters.begin();
    for (auto pair : letters)
    {
      WRITE_IF(pair.second) << "Seen "
        << pair.first << std::endl;
      if (pair.second > most.second)
      {
        most = pair;
      }
      else if (pair.second < least.second)
      {
        least = pair;
      }
    }
    std::cout << "Commonest letter: "
      << most.first << " ("
      << most.second << ")\n";
    if (most != least)
      std::cout << "Rarest letter: "
        << least.first << " ("
        << least.second << ")\n";
  }
}
```

Listing 3 contains `logging.h` and Listing 4 is `letter.cpp`.

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.

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?

View from the Chair

Bob Schmidt
chair@accu.org

ACCU 31st Annual General Meeting

ACCU's 2019 Annual General Meeting will be held Saturday 13th April 2019 at 13:30, in the Bristol Marriott City Centre Hotel, in conjunction with ACCU's annual conference [1]. Although the agenda has not been set as this is being written, the table on the right contains the remaining critical dates leading up to the AGM.

These dates have been posted to the web site, which will be updated with the agenda as it becomes available.

Important dates for the ACCU AGM 2019

Event	Date	Countdown
Agenda freeze	16 March 2019	AGM - 28
Voting opens	23 March 2019	AGM - 21
AGM	13 April 2019	

BSI Standards Awards

As reported last issue, Francis Glassborow was nominated for a 2018 BSI Standards Award in the category of International Standards-Maker. From the Nominees announcement:

> This award is to recognize an exceptional contribution to representing, championing and safe-guarding a UK committee's position in European and/or international standards-making and recipients will have demonstrated how their contribution has specifically supported UK interests/positions in the development of a standard(s) which has a significant impact on industry, the economy and/or the public good.

This category had the most nominees, by far – with 19 nominations, it was a very crowded field. In the end, 8 nominees were selected for the award; unfortunately, Francis was not among them. A complete list of BSI Standards Awards winners and their biographies is available on the BSI web site [2].

Congratulations once again to Francis for his well-deserved nomination.

Beginners welcome

CVu and *Overload* have a reputation for presenting advanced and cutting-edge material. Programmers at the beginning of their careers may find these articles intimidating, and wrongly think that ACCU is an organization only for the experienced. In order to reach out to those who are just starting their professional careers, and hopefully encourage them to become members, we are soliciting articles that deal with subjects that those with more experience take for granted.

Authors, please submit your ideas and articles to Fran [3] or Steve [4] – whether for beginners or not. If you have never before written for ACCU, they make the process easy and enjoyable.

References

[1] ACCU 2019: https://conference.accu.org/
[2] BSI Standards Awards: https://www.bsigroup.com/globalassets/localfiles/en-gb/about-bsi/nsb/standards-awards/2018/bsi-standards-awards-2018-winners.pdf
[3] Frances Buontempo, *Overload* editor: overload@accu.org
[4] Steve Love, *CVu* editor: cvu@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