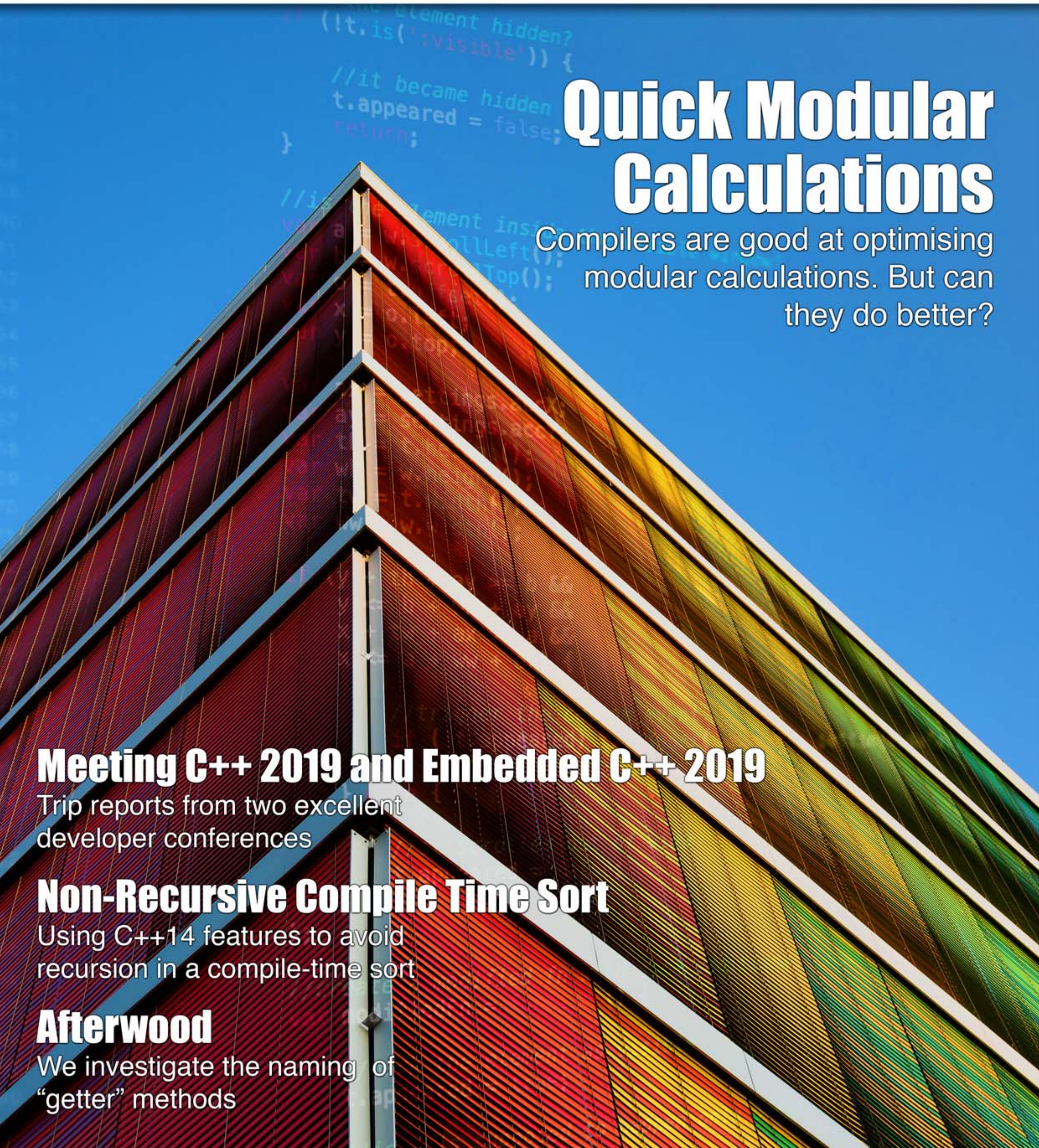
overload 154

DECEMBER 2019 £4.50



Quick Modular Calculations

Compilers are good at optimising modular calculations. But can they do better?

Meeting C++ 2019 and Embedded C++ 2019

Trip reports from two excellent developer conferences

Non-Recursive Compile Time Sort

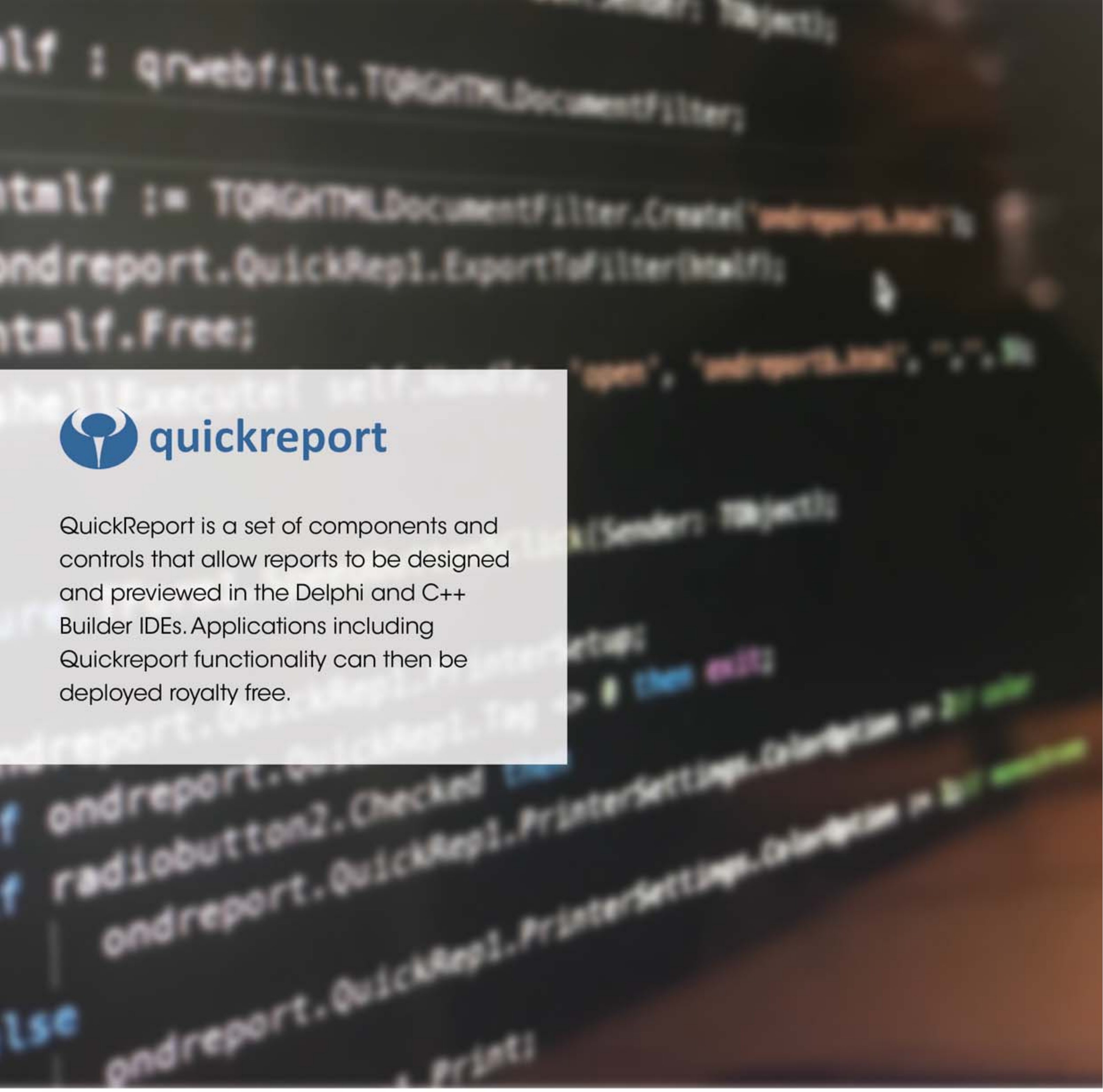
Using C++14 features to avoid recursion in a compile-time sort

Afterwood

We investigate the naming of "getter" methods

A magazine of ACCU

ISSN: 1354-3172



quickreport

QuickReport is a set of components and controls that allow reports to be designed and previewed in the Delphi and C++ Builder IDEs. Applications including Quickreport functionality can then be deployed royalty free.

Key partners include:

Agisoft · ASPOSE File Format APIs · ATLASSIAN · axure · BrowserStack · DevExpress

embarcadero · Hex-Rays · intel Software · JetBrains Platinum Reseller · Microsoft Silver Partner · qbs Publishing

Sketch · SMARTBEAR · SPARX SYSTEMS · Telerik Develop experiences · think-cell · Visual Paradigm

Plus many more on **www.qbssoftware.com/developer**

For your latest software needs, contact our team on:

020 8733 7100 sales@qbs.co.uk

qbs SOFTWARE

OVERLOAD 154

December 2019

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 155 should be submitted by 1st January 2020 and those for Overload 156 by 1st March 2020.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

The ACCU

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

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

Inside-Out

Sometimes things appear to be inside out. Frances Buontempo considers when a shift of perspective can make things seem better.

Cutting to the chase, I haven't written an editorial and I suspect I have run out of excuses now. I have been distracted by reading *Jerusalem* by Alan Moore [Moore16]. This book pulls together a variety of characters, including gods and daemons, and explores ideas related to extra dimensions, directly referencing *Flatland* [Abbott84], which some readers may be familiar with. It tries to imagine what life in higher dimensions would be like, starting with the perspective of two-dimensional beings. Such beings, living on the equivalent of a piece of paper, have a limited view of their surroundings. A three-dimensional being would have a very different view, being able to see inside rooms from above. The flatlanders would have no concept of above. By shifting perspective, the world looks very different.

Alan Moore also wrote the comic *Watchmen* [Moore], which was made into a film and has recently been serialized for the television. From memory, the comic has a long discussion near the beginning of super-hero outfits. Cloaks are called out as a no-no. They are a health and safety nightmare, both being a trip hazard and potentially getting stuck in fire-escapes or other fixtures and fitting as the hero attempts to duck and dive after whoever they pursue, or perhaps run from. I personally have never understood why so many super-heroes wear their underpants on the outside of their trousers. Inside-out, with no possible advantage, to my mind. Finding a suitable outfit is often a headache for anyone. I am sure many readers will join me in bemoaning a lack of pockets, having put valuable possessions in a handbag. Some things are much safer in an inside pocket.

There are many conventions on placement, particularly of clothing, though not limited to attire. Should you tuck your shirt in? Should a scarf go over a coat or inside? What about code documentation? Should this be inside the code as comments or along-side the code in document form? Where do diagrams go? How do you explain your code in diagram form? For physical objects, trying to represent three dimensions on a two-dimensional surface throws up many challenges. At some scales, a plan and elevation will provide enough information to indicate what goes where. However, flat-pack furniture is often shown as so-called 'exploded diagrams', attempting to show how the parts fit together, showing which fixings go where inside. Sometimes a few words along with the diagram help. So much for furniture. How many dimensions does a code-base have? Can you really represent it on a flat surface? Various projections of the earth onto maps exist. Each will distort the landmass in some way. The Mercator projection gives the impression Greenland and Africa are the same area. Getting the areas right can make all the shapes wrong. What does a UML diagram do to your code base? Does code have a surface area? Who knows? I don't.

Maps are often use to plan journeys. Trying to represent a landscape, road network or pathways is a challenge. Unknown areas may just be marked 'Here be dragons'. The same stands for codebases. When planning a walk, it is useful to know just how steep the paths are. Many popular maps on smart phones don't show this. Other types of maps might. The Ordnance Survey used to use physical marks to map out the height above sea-level [OS-1], and does show contour lines joining points of equal height, and therefore indicating the steepness of a path. Some of the markers have subsided somewhat, meaning the data is no longer accurate. They now use GPS and similar to get accurate measurements. If a marker stone is on top of a hill, and a coal-mine underneath collapses, the marker might just end up inside the hill, rather than on top. Edges and boundaries move over time. Furthermore, tides rise and fall all the time, so averages are taken. You should also clarify which sea-level. Newlyn in Cornwall, it turns out [OS-2], has a stone pier sited on granite, so is probably more stable than some other points around the UK. Finding the length of the coastline round a country is difficult too. The result depends on the accuracy with which you measure the edges, ignoring parts that may fall into the sea from time to time. Any distance is always an approximation at a given scale. Tides and edges are not the only things to move over time. Documentation can become out of date or architecture diagrams go stale. Tests might start to fail.

Given we cannot find one best way to provide accurate distances, diagrams, or pathways for many things, what are the options for finding our way round a code base? Start somewhere in the middle, with a break point in a function and see what hits it? Add some logging? Start at `main`, or another boundary, and follow a path through? Draw a new diagram? Attempt to understand existing diagrams? There are many options. There's no one best way. Learning to navigate code bases, geography, meetings or indeed any situation is an important skill.

How do you navigate your way through a multi-track conference or options for studying a course? Do the session titles indicate what's really involved? There may be an abstract somewhere, but without reading through these properly in advance, you might just pick according to titles or speakers. Perhaps that doesn't matter for an hour's talk as much as for a long course. The 'blurb' presented to the outside world may or may not be an accurate representation of what's going on inside. So many companies have a mission statement, taglines or slogans which might be somewhere between meaningless or misinformation. The real thing? Because you're worth it. Just do it. I shouldn't mock; hours of thought have gone into these and the companies concerned are almost certainly making money. ACCU adopted 'Professionalism in programming' a while ago, and that has stuck. Though this means different things to different people, it's a good place to start. How do you describe ACCU when you talk to people about us?

Frances Buontempo 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 has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

I often surprised by non-technical peoples' mental models of how things work. Someone once claimed their smart speaker only started listening in on them *after* they said its name. I was going to ask how it heard its name if it wasn't listening, but suspected I wouldn't get far with that line of enquiry. Mind you, tech people often talk about a fabled hamster not pedalling hard enough if machines slow down, or gremlins larking around, or just say 'Magic' when asked how something works. On one level, we do ourselves no favours, and on another it is very difficult to explain something clearly and precisely to others who have no background knowledge at all. Sometimes we use analogies, but they always fall down in the end. They can be a good starting point for a discussion though.

Do your friends or family know what you do for a living, if you are in tech? Many presume you can fix their home internet connection problems or similar, though not all. If people don't know what a program or an app is, even when they use such things, how do you explain that you create these? Even talking to other tech people working in different domains can be hard. A GUI programmer will use different language to a backend developer. When we say 'agile', what do we mean? When we say 'test', what do we mean? If you claim to do TDD, do you write the tests first? Perhaps this is about the ambiguity of language in addition to disparate experience. Sometimes the best way to understand each other is to do something together, rather than talk about it. If someone wants to know what a website is, build one with them. If someone wants to know what an algorithm is, work through one together on paper with a pencil. If someone wants to know what stochastic calculus is, well, there are ways to give an overview, but it takes a lot of thought to make complicated things simple to 'outsiders'.

Throughout history, there have been many experts who have tried to explain details of their specialist knowledge to the general public. Einstein wrote many academic papers, but also wrote for a wider audience, including his book *Relativity: The Special and General Theory* [Einstein16]. He claimed:

> The present book is intended, as far as possible, to give an exact insight into the theory of Relativity to those readers who, from a general scientific and philosophical point of view, are interested in the theory, but who are not conversant with the mathematical apparatus of theoretical physics.

Feynman also tried to popularize physics with books and talks [Wikipedia-1]. Brian Cox is on a similar trajectory, with articles talking of the Brian Cox effect [Manchester]. By explaining complicated things with enthusiasm, he has inspired others to study physics. Some have claimed relatively recently that people are sick of experts [Gove16], but I don't think that is true. I think most people like to have an idea of how things work or new ways of looking at life, the universe and everything. They may need an Einstein, Feynman or Cox to explain it. Or even Alan Moore to put ideas into story or comic form, at least providing analogies and parables to get the brain ticking.

Some experts are insiders who share secrets. Though this can be a bad thing, for example stealing a company's intellectual property, others are whistleblowers. The UK government website [GOV.UK] describes it like this:

> You're a whistleblower if you're a worker and you report certain types of wrongdoing. This will usually be something you've seen at work – though not always.
>
> The wrongdoing you disclose must be in the public interest. This means it must affect others, for example the general public.

I'm not sure how precise this definition is, but the idea is to protect workers who call out bad behaviour or practice. Wikipedia has a list of whistleblowers [Wikipedia-2]. The earliest entries concern the military and war crimes. Over time, the frequency of entries increases, and the domains expand into spying, the pharmaceutical industry, banking and then IT. Of course, the list is not definitive. Volkswagen isn't on the list [Reuters16]. Other computer systems have had problems too, and though whistleblowing may not be involved, experts may have something to say on the matter. The grounding of the Boeing 737 MAX planes was called for by experts [Wikipedia-3]. Thank goodness for experts.

Some companies willingly share their source code. Free and open source software, FOSS, is a term many of us are familiar with. As AI becomes all the rage, I increasingly see open source code and online platforms offering tools to help people develop solutions quickly. You can pick up a model and tweak the parameters to see if you can improve its performance, even if you don't have a clue what a GAN or CNN is, or what reinforcement learning is. Who needs experts when you can twiddle knobs, change numbers and generally hit things with a hammer until they appear to work? To be fair, some initiatives started as a way of making sure proprietary AI setups don't inadvertently invent some SciFi nightmare like The Terminator's Skynet. For example, OpenAI [OpenAI] has a charter, which starts:

> OpenAI's mission is to ensure that artificial general intelligence (AGI)—by which we mean highly autonomous systems that outperform humans at most economically valuable work—benefits all of humanity.

Other organisations are also looking out for us. The Future of Humanity institute [FHI] claims to 'bring the tools of mathematics, philosophy and social sciences to bear on big-picture questions about humanity and its prospects'. They include a governance of AI section, keeping an eye on where the tech is going, including governments and industry leaders. Since we don't know what AI is really capable of, predicting what might happen is hard, but bringing together experts to discuss matters seems like a good idea. Experts from different areas will have different perspectives on what is happening and what might come to pass.

Despite my lack of editorial, I hope looking in a different direction, getting a new perspective, or listening to experts might help me, or a reader or two. If you happen to be an expert in something, or even just get the hang of it, do write it up for someone else to read. This might fire you with new ideas, better ways of explaining yourself and new things to discover. Happy thinking.

References

[Abbott84] Edwin A Abbot *Flatland: A romance in many dimensions* Seeley & Co 1884.

[Einstein16] Albert Einstein *Relativity: The Special and General Theory* First published Dec 1916.

[FHI] Future of Humanity Institute: https://www.fhi.ox.ac.uk/

[GOV.UK] 'Whistleblowing for employees': https://www.gov.uk/whistleblowing

[Gove16] Michael Gove in an interview with Faisal Islam, Sky News, 3 June 2016: https://en.wikiquote.org/wiki/Michael_Gove

[Manchester] 'The Brian Cox effect' rejuvenates physics in Britain: https://www.physics.manchester.ac.uk/research/impact/the-brian-cox-effect/

[Moore] *Watchmen*, Alan Moore, DC Comics, 1986–1987

[Moore16] Alan Moore (2016) *Jerusalem*, Knockabout, Sept 2016

[OpenAI] OpenAI Charter: https://openai.com/charter/

[OS-1] Ordnance Survey (blog): https://www.ordnancesurvey.co.uk/blog/2018/05/25-years-since-last-benchmark/

[OS-2] Ordnance Survey (blog): https://www.ordnancesurvey.co.uk/blog/2011/08/how-do-you-measure-sea-level/

[Reuters16] 'Fired employee says Volkswagen deleted documents about emissions tests', published in *The Guardian*: https://www.theguardian.com/business/2016/mar/14/volkwagen-whistleblower-deleted-documents-emissions-tests

[Wikipedia-1] Richard Feynman: https://en.wikipedia.org/wiki/Richard_Feynman

[Wikipedia-2] List of whistleblowers: https://en.wikipedia.org/wiki/List_of_whistleblowers

[Wikipedia-3] Boeing 737 groundings: https://en.wikipedia.org/wiki/Boeing_737_MAX_groundings#Experts

Trip Reports: Meeting C++ 2019 and Embedded C++ 2019

Deciding which conferences to attend is difficult, and we can't go to them all. Svitlana Lubenska, Hans Vredeveld and Benedikt Mandelkow give us a flavour of what we may have missed.

From Svitlana Lubenska

It was my 4th Meeting C++ [Meeting C++] conference and I liked it the most. In this report, I will try to explain why and encourage you to watch the videos once they are available and, of course, to go to this conference next year!

As always, the conference takes place in Berlin, in the beautiful Vienna House Andels Hotel, and lasts 3 days.

During these days, I literally did not leave the hotel, because every day was filled with so many interesting things!

The first day had so many interesting speakers so that it was really hard for me to decide where to go. I will list the talks I finally attended, but I am going to watch the recordings [MeetingC++] of others too.

- The opening keynote by Howard Hinnant about <chrono> [chrono] was just great! As I have recently worked with this library, I have now learned better ways of using it and tomorrow am going to refactor my code ☺. Howard not only listed the existing and new (coming in C++20) features but also explained the philosophy of design, making it clear why they implemented certain features. He also gave cool advice to people who want to design their own library (although not necessarily: I think I can apply them even if I write just one class ☺). Howard kindly shared his slides [Hinnant19].
- The first talk was the one I was really looking forward to attending – '10 Techniques to Understand Existing Code' by Jonathan Boccara. Let me first talk about the speaker. I first heard his name while I was on maternity leave. Although it was year when I had a break from my profession so I could focus on my family, I did want to follow what was happening in the C++ world, so I was listening to cppcast [CppCast] where I heard an interview with Jonathan which was super interesting. I started to read his blog fluentcpp.com and also watched videos on youtube. When I got back to work, I continued following his posts, used a lot of his tips while coding and still think his explanation of tag dispatching technique is the best ☺.

  I was also super happy when Jonathan published his book and so, of course, I could not miss his talk at conference!

  The talk was interesting and helpful; I especially liked the tip about using call stacks ☺.
- Daniela Engert talked about the famous modules: once they are there, go to see how to use them ☺.
- I went to listen to Hana Dusíková with her 'Compile Time Regular Expressions with Deterministic Finite Automaton' talk because I had heard so much about her and missed the talk about Hana's library last year. I am a big fan of regular expressions (in Perl mostly, as I am in love with scripting languages), so… what can I say, I was really impressed! You will also be impressed if you check the slides [Dusíková19].
- I could not miss the talk 'C++20: The small things' by Timur Doumler because I just love the examples he usually gives. I was not disappointed ☺.

As always, the first day was finished with the Quiz! It was the first time I participated; of course, failing to solve a lot of tasks… but it was SO MUCH FUN! Please always try it out when you at a conference; the best part is that you are with people you have never worked with but after the quiz it is like you have known them for ages ☺. Last year, I was so inspired that I created a similar event for my colleagues, not as evil but still… 15 minutes that felt like the best team building ever ☺.

The second day was even better! Because:

- It started with an inspiring keynote by Frances Buontempo. You have to watch it! So many questions to think about. I recently read a book by Andreas Weigend [Weigend17] (not exactly about AI) which raised similar ideas in my head, so this talk was really important to me.
- Arne Mertz talked about code smells. I liked that the talk was built on real code examples where he asked the auditorium to find the smells. I think you should show this talk to junior programmers straight away when they join your team. You know, just in case ☺.
- Pavel Novikov talked about asynchronous C++ programming, on a pretty advanced level. I particularly liked his examples on tasks (and puppies ☺).
- Phil Nash is still obsessed with error handling in C++. I recommend watching his previous talk first [Nash18]. Basically, this time he was considering different proposals related to this and showing how the code would look. I personally am quite happy with **`system_error`** even with the problems Phil listed, but it was interesting to see what can be improved and why.
- Guy Davidson gave his talk 'Teaching Analytic Geometry to C++', as always with so much positive energy. Now with a geometry library, I am sure C++ will shine ☺.

The last-but-not-least part was Lightning Talks. There were so many that I cannot list them all but just mention my favourite – by Tina Ulbrich, 'The Life-Changing Magic of Tidying Up' [Ulbrich19]. The idea was so simple

Benedikt Mandelkow is a 23-year-old CS student from Germany. He has a Bc. CS from the RWTH Aachen and is now studying computer graphics and image processing at Uni-Koblenz as a Master CS student. He is interested in a wide range of programming languages and the underlying technologies he uses. You can contact him at benedikt.mandelkow@rwth-aachen.de

Svitlana Lubenska has a Masters degree in Applied Mathematics and is working as a Senior Software Engineer at Brainlab (Munich, Germany). You can contact her at s.lubenska@gmail.com

Hans Vredeveld started working in the software industry 20+ years ago as a system administrator. Via application administration, he soon moved into software development, where he was bitten by the C++ virus. Not wanting to be cured, he is always searching for the next cool thing C++. He can be contacted at accu@closingbrace.nl.

The code presented **varies between crazy and psychopathic**, and stands for a lot of "what the..." and fun. I only hope to **never see such code in production**.

and brilliant so that I could not help laughing. She applied ideas from a home-cleaning philosophy to code writing. I read everything written by Marie Kondo and I try to follow her advice at home, but it would never have occurred to me to use these ideas in code. I think my mindset is different now, and this simple move will bring even more joy to my work ☺.

The third day opened with a talk by Jon Kalb where he talked about 'Modern Template Techniques'. I found policy classes very interesting, not being sure if I had heard of this technique before. So, I learned something again ☺.

- Fabio Fracassi talked about C++20 new features. The talk intersected a bit with Timur's talk but, in this case, the more examples the better. I liked the talk a lot!
- The 'Testing Legacy Code – Fuzzing for Better Input Data' talk by Tina Ulbrich and Niel Waldren was sooo good! I am going to check libFuzzer as soon as I get to my computer!
- There were also secret Lightning Talks, my favourite was the one done by Peter Sommerlad just because I loved the idea of having 'The Rule of DesDeMovA'.
- Jens talked about burnout, which is supposed to be a sad topic… but those beautiful photos… I don't know what to say, you should see them ☺.
- It was time for the closing keynote by Walter E. Brown who actually started the talk in German ☺. He made us laugh but also think about the cost of the mistakes programmers sometimes make. For me, it was also important topic as I write medical software, so…

Yes, these three days were awesome! I still have a lot to catch up (talks I did not attend), but I have already learned so much from people who are so kind to share their knowledge and thoughts.

See you next year!

From Hans Vredeveld

We have a great C++ community that gives a lot, at least to me. Some time ago, I was thinking about how I could give back something. I concluded that writing for *Overload* would be a good way to start. But, what to write about? An easy answer to this question is, write trip reports for events I went to. So I wrote a trip report for *Italian C++* earlier this year [Vredeveld19]. Now it's time for my second trip report. This time about *Meeting Embedded* and *Meeting C++*, the second time I have visited these conferences. It's also the second time *Meeting Embedded* has been held.

Meeting Embedded

Meeting Embedded [MeetingEmbedded] is a one-day single-track conference that is held the day before *Meeting C++*. This year it was on November 13th. The day started with a short welcome by Jens Weller, followed by a one-hour keynote, after which nine 30-minute talks followed. After the keynote and after each third talk, there was a longer break in which we could get some refreshments or have lunch. Between the other talks, there was a 5-minute break to allow for one speaker to leave the stage and the next to set up.

The keynote was delivered by Peter Sommerlad. In his talk he argued that we should stop using C and start using modern C++ for embedded development. Part of the problem is that vendors give examples predominately in C and that these examples often constitute bad code. The development cycle usually consists of quickly writing your code, so that you get to the real thing: debugging. Peter then argued that a lot of the problems found during debugging (or worse: only found in production) can be circumvented by making proper use of modern C++. C++ gives us a type system with deterministic object lifetime, compile time programming, and a standard library, amongst others, that make many of the problems with C go away like snow before the sun. Next, he went into what it means to properly make use of modern C++, how many of the problems that in C can be only found at runtime can be found at compile time in C++. If you want to convince your colleagues to start using C++ instead of C for embedded, watch the keynote on YouTube for some inspiration when it becomes available.

Many of the talks that followed were also on the subject of writing better code. Maciek Gajdzica looked at different design and development methods, also touching on the subject of better languages than C for embedded development. In particular, he went into the C4 model for visualizing software architecture [C4]. Paul Targosz taught us how we could make our code more robust by using **`const`**/**`constexpr`**/ **`consteval`**, by putting configuration in files instead of littering it throughout our code, and by using a policy-based class design. Nikola Jelić told about how he created a library for physical units modelled after the chrono library and how this improved his code. Frank Mertens went into different ways of doing heap management and the impact they have on fragmentation. Pawel Wisniewski compared different ways of implementing call backs and listed the pros and cons of each way. Daniel Penning explained that generic programming allows us to reuse more code. In particular, he explained that a lot is already available through the algorithms in the standard library and that it will even get better with concepts in C++20. A lot of the things that these speakers talked about were things I already knew, but forgot about. Sometimes they also presented a fresh point of view that I hadn't thought of before.

In my professional life I have to deal with people that think that the only viable language for low level programming is C. Although I'm not a low level developer, I try to convince them that C++ is a better alternative. Now I learned that there is even another alternative: Rust. Jonathan Pallant went into how he used Rust on Nordic Semiconductor's nRF9160 [Nordic] and made the system more secure and use less power. James Munns presented a Rust library for serialization and deserialization: Serde [Serde], and how Serde makes serialization and deserialization easy.

A real gem for me was Diego Rodriguez-Losada and Brennan Ashton's talk about Yocto [Yocto]. Yocto is a system for creating your own embedded Linux distribution. Diego and Brennan first gave an introduction of what Yocto is. Then they went into using Conan [Conan] with Yocto, and into uploading images to your embedded device. One of the things I run into regularly is that building a new Linux image with Yocto takes several hours. Using Conan makes it possible to build the applications beforehand and store them in a location from where they can be retrieved during integration. This will reduce the build time significantly and make it feasible to create an image multiple times a day.

Meeting C++ Day 1

The first day of Meeting C++ [MeetingC++], November 14th, started with the welcome message by Jens Weller, soon to be followed by Howard Hinnant's keynote. Howard described the new things that <chrono> [chrono] gets with C++20, calendars and formatting. An important part of the keynote was explaining why <chrono> was designed as it is. Of course, that was to be expected when the keynote's title is 'Design Rationale for <chrono>'. Some old and many new types were reviewed; **`duration`**, **`time_point`**, **`year`**, **`month`**, **`day`**, **`time_zone`**, **`zoned_time`** and different kinds of clocks (and I still missed some). A recurring theme in the presentation was what operations and implicit conversions are allowed. E.g. converting from **`duration<int, hours>`** to **`duration<int, seconds>`** is implicit as no information is lost, while the reverse conversion is only possible with an explicit **`duration_cast`** as precision is lost (4000 seconds is 1 hour, losing 400 seconds in the process). Also, many of the usual operations on integral and floating point values don't make sense for types in <chrono>. For example, subtracting two **`time_point`**s results in a duration, but adding two **`time_point`**s results in a compile error as it doesn't make sense and isn't implemented. I could go on describing what Howard said. Instead I advise you to watch the presentation on YouTube as soon as it becomes available. Not only will you learn a lot about <chrono>, but also about designing a library in general.

After the keynote, we had a lunch break with plenty of excellent food for everybody. In the afternoon, there were four sessions of one-hour talks, with a 30-minute coffee break between the second and third session and 15-minute breaks between the others. I went to Jonathan Boccara's '10 Techniques to Understand Existing Code', Dawid Zalewski's 'Lambdas – the old, the new and the tricky', Bryce Adelstein Lelbach's 'The C++20 Synchronization Library' and Ivan Cukic's 'Compile-time type transformation'. Four interesting talks worth going to.

The day ended with the evening program that consisted of the pizza-pasta buffet, the Conan C++ quiz and socializing. The Conan C++ quiz, very well presented by Diego, is a quiz where the participants work together in groups of six, and they have to guess/deduce what the output is of small C++ programs. The code presented varies between crazy and psychopathic, and stands for a lot of "what the..." and fun. I only hope to never see such code in production. I would fire the person that wrote that (or find myself another place to work if that's impossible).

Meeting C++ Day 2

The day started with Frances Buontempo's keynote 'Can AI replace programmers?' Frances explained what AI is, went back and forth between yes and no in answering the question. She concluded that AI can replace programmers using genetic programming, but that it takes an awful lot of time. Finally, she also noted the AI effect: as soon as AI solves a problem, the problem is no longer part of AI.

In the afternoon, there were again four sessions of one-hour talks. I went to John Lakos's 'Value Propositon: Allocator-Aware Software', Jonathan Müller's 'Using C++20's Three-way Comparison <=>', Phil Nash's 'The Dawn of a New Error' and Arvid Gerstmann's 'Multithreading 101: Concurrency Primitives From Scratch'. In his talk, John went from the past to the present to the possible future. He described how memory management with the C++11 allocators is difficult to use, how C++17's polymorphic allocators and associated types make this much easier and how it could even be made more easy with the BB20V library that is currently under development at Bloomberg. Jonathan discussed that when you overload one of the operators **==** or **!=** for a custom type, you should also overload the other, and that when you overload one of the operators **<**, **<=**, **>=** or **>** for a custom type, you should overload all of them. Next, he explained what C++20's **`operator<=>`** is and that it is enough to overload this operator and **==**. The other operators are implicitly implemented. Phil discussed new types, and a related extension of the language, for error handling that hopefully will make it into C++23. The main motivation for these types is to have better error handling than the C-style return values, including in situations where exceptions cannot be used. Arvid started with the low-level building blocks of multithreading. He described what a spinlock is and how it can be written. Next he taught us what a futex is. He implemented it first for Linux (with the system call 'futex') and for Windows. Then he went on to implement it with code that would work on any platform that has modern C++. For this implementation he used **`std::condition_variable`**s. To the end of his talk he took just a few minutes to implement a condition variable using a futex (that was implemented with a condition variable; yes, he was aware of that).

In the evening there was food again in the form of a pizza-pasta buffet, and there was time for socializing. And, most importantly, there were the lightning talks. As always, it is fun to see what people come up with to talk about for only five minutes. With 19 lightning talks, the second half of the evening was well packed.

Meeting C++ Day 3

On the third day, the schedule was reversed. First we had the sessions with the talks and then the keynote. This time there were only three sessions of one-hour talks, not four. In Jon Kalb's 'Modern Template Techniques', I learned some things about type traits, tag dispatch, policy classes, perfect forwarding, and that infinity == 12. Deniz Bahadir's 'Oh No! More Modern Cmake' left off where his 2018 talk 'More Modern CMake – Working with CMake 3.12 and later' ended. He clearly described the difference between **`PRIVATE`**, **`INTERFACE`** and **`PUBLIC`**. He explained what object libraries are and how to use them, and he gave several recommendations for using Cmake. The last regular talk I went to was Bart Verhagen's 'Designing costless abstractions'. He compared a few **`std-`** types from modern C++ to their old (C-style) counterparts (e.g. **`std::unique_ptr`** to raw pointers) and how these C++ types are an abstraction without any cost. I had expected that he would go a bit more into how to design those abstractions, or why they were designed as they are. Unfortunately for me that meant that this was the least interesting talk of the conference.

Instead of the fourth session, Jens gave an update about *Meeting C++*, both the platform and the conference. He also gave some indication when the videos will be uploaded to YouTube [MeetingC++]. Expect the lightning talks first, then the keynotes and, after Christmas and New Year, the regular talks. Then he went on to something for which there was time planned, but that was not on the schedule: the secret lightning talks. They will be on YouTube too, so you may wonder what is so secret about them? (Answer: that you don't know about them when you read the schedule.)

Finally it was time for the closing keynote by Walter E. Brown. I had seen a couple of Walter's talks on YouTube and one live at *C++ on Sea*, so I know that he is a good speaker and I had high expectations for his keynote. He exceeded my expectations by a large margin. He managed to put a lot of humour (I had to wipe away tears on more than one occasion) in his presentation, while keeping it serious at the same time. Watch it when Jens uploads it to YouTube!

From Benedikt Mandelkow

Hello,

I'm Benedikt Mandelkow, a CS student from Germany.

During my time as a Bachelor student, I had the opportunity of meeting Jonathan Müller [Müller], which in turn meant that I got interested in cpp.

In a casual conversation with him, the idea came up that I could apply for a student ticket for meetingcpp and, as you might have guessed, that is exactly what happened. I had been to Berlin before and really liked the city, so after I had received the acknowledgment that I was one of the randomly selected students, it was an easy call for me to book my ticket.

On the first evening before the conference, we went to grab some food and even though my trains were very unreliable, I managed to arrive just in time at a restaurant. Coincidentally, I had already spent my time on the train with a group of developers who were also heading to Berlin for different reasons.

My first conversation was with another student who had attended multiple cpp conferences already, and who briefed me to focus on the conversation in between the talks instead of on just the talks themselves, which proved to be really good advice.

After a few minutes, my next surprise was that I was sitting not far from Arvid Gerstmann, someone I had so far only known from twitter [Gerstmann]. We exchanged a few words regarding an experimental programming language [AEXPL] he is working on and we continued to have very good conversations over the course of the event.

Seeing many people that I had so far only known from videos or blog posts was really nice, but the important thing for me was that the atmosphere I encountered was not elitist but everyone I tried to talk to was very approachable.

I had hoped for this to be the case, but I think without actually attending a conference, it’s hard to believe as it definitely seemed a bit unreal to me as well. Surely there are still ways to improve on this. When I tell people about my experience, they are really surprised, which indicates that there are many people who have reasons that keep them away from possibly joining the community.

The main focus of the conference is, of course, not the well-known people but the vast majority of attendees who work for a wide range of companies. The good news was that it was easy to talk to them as well because when I was standing next to some people I had never seen before, it never took long for one of them to ask me a question or for me to take an opportunity and make a remark. After that, conversations were really easy.

The focus was never on recruitment or testing other people’s knowledge but much more sharing opinions, understanding and stories.

The excellent food surely also contributed to a relaxed atmosphere because this allowed everyone to just focus on the event instead of competing for the last desert.

I will briefly describe some talks I watched but won’t detail everything because the talks will be all available online and it’s more about the personal interactions than the talks themselves.

The first talk was from Howard Hinnant about his work on chrono. While some people I talked to argued that it was unusual in the sense that it was rather specific and not a general design talk, I found it to be helpful as it actually provided direct benefits that one can use in practice to write better code in specific cases.

One benefit of attending talks in person vs watching the videos was that I was more focused, which allowed me to follow the talk about ‘The C++20 Synchronisation Library’ from Bryce Lelbach and the Spaceship operator from Jonathan Müller much more completely, whereas on my own I get distracted more quickly sometimes. (I knew both talks from previous recordings.)

I made the very conscious decision to sit through an hour-long explanation of std::midpoint, which turned out to be a really good one because I liked the problem statement and the approach and care which was taken to address it.

Additionally, I was really interested in seeing the person (Marshall Clow [Clow]) who programs the standard library implementation I use most of the time.

The most important talk for me was ‘Modern Template Techniques’, which was in a perfect spot in the morning where I was still able to actually digest the content. I also had talked to Jon Kalb the evening before, which really made a difference while experiencing the talk, much more personal.

By accident I now know about SFINAE, which was something I had previously explicitly excluded from ever learning because it just seemed way too complicated to spend time on.

But then Jon Kalb tricked me because he explained it on one slide without mentioning the name and when I then realized what I had just learned I was both embarrassed because I had learned something I never wanted to but also relieved because it was actually not as bad as I had thought (famous last words).

The final talk from Walter Brown was very entertaining and had many interesting arguments and the sentiment that struck with me was that we are a profession and should strive to produce more reliable software and take responsibility for our work.

On the other hand, without some messiness and with too much perfectionism, we would not have the world wide web as it is because things would move that much more slowly but I think it is still very valuable to take a strong stance like Walter Brown at times because it’s just, as always, a trade-off and thus discussion about it is valuable.

Finally, I want to thank meetingcpp for offering student tickets, which made it possible for me to attend, all the attendees for the atmosphere and Jonathan Müller for being very patient with me at all times [Brown18]. ■

References

[AEXPL] Arvid's EXperimental Programming Language: https://github.com/aexpl/aexpl

[Brown18] Walter E. Brown ‘Thank You (I’m sorry that it’s taken me so long to say it)’ presented at C++ Conference 2018, available at: https://www.youtube.com/watch?v=L5daPjK00bo

[C4] ‘The C4 model for visualising software architecture’: https://c4model.com/

[chrono] The chrono library: https://en.cppreference.com/w/cpp/chrono

[Clow] Marshall Clow: https://cppalliance.org/people/marshall.html

[Conan] Conan: https://conan.io/

[CppCast] A podcast for C++ developers: https://cppcast.com/

[Dusíková19] Hana Dusíková (2019) ‘Compile Time Regular Expressions With Deterministic Finite Automaton’, presented at *Meeting C++ 2019*, available at: https://www.compile-time.re/meeting-cpp-2019/slides/#/

[Gerstmann] Arvid Gerstmann, twitter conversations: https://twitter.com/ArvidGerstmann/status/1196002167167029249

[Hinnant19] Howard Hinnant (2019) ‘Design Rationale for the <chrono> Library’, presented at *Meeting C++ 2019*, available at: https://meetingcpp.com/mcpp/slides/2019/Hinnant.pdf

[MeetingC++] Meeting C++ website: https://meetingcpp.com/
YouTube channel: https://www.youtube.com/user/MeetingCPP/videos

[MeetingEmbedded] Meeting Embedded 2019 website: https://meetingembedded.com/2019/

[Müller] Jonathan Müller (blog): https://www.jonathanmueller.dev/

[Nash18] Phil Nash (2018) ‘Option(al) Is Not a Failure’, presented at *C++ Now 2018*, available at: https://www.youtube.com/watch?v=OsRty0KNDZ0

[Nordic] Nordic Semiconductors: nRF9160, available at: https://www.nordicsemi.com/Products/Low-power-cellular-IoT/nRF9160

[Serde] Serde framework: https://serde.rs/

[Ulbrich19] Tina Ulbrich ‘The Life-Changing Magic of Tidying Up’, presented at Meeting C++ 2019, available at: https://meetingcpp.com/mcpp/slides/2019/The%20Life-Changing%20Magic%20of%20Tidying%20Up.pdf

[Vredeveld19] Hans Vredeveld (2019) ‘Trip Report: Italian C++ 2019’ in *Overload* 152, August 2019, available at: https://accu.org/index.php/journals/2681

[Weigend17] Andreas Weigend (2017) *Data for the People: How to Make Our Post-Privacy Economy Work for You*, Basic Books, ISBN-13: 978-0465044696

[Yocto] Yocto project: https://www.yoctoproject.org/

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

Non-Recursive Compile Time Sort

Compile time sorting usually uses recursion. Norman Wilson shows how C++14 features make this easier.

There was a time when compile time sorting was a holy grail for me. It's still quite a tricky piece of meta programming to pull off. When I came up with this idea, I had a quick look around for prior art including *Stack Overflow* where I first posted this code [Wilson]. The usual approach involves a set of recursively defined meta functions. Since C++ 11 introduced parameter packs and C++ 14 added **`std::index_sequence`** to the library things have go a bit simpler. This is a non-recursive compile time sort algorithm, which I think is much simpler and easier to understand. I like to think of parameter pack expansion as a way of saying 'for each do this'. This algorithm depends on thinking in these terms – thinking about how you can apply a series of simple operations to all elements of a pack at once and end up with a sorted sequence.

The problem

I'm going to define a **`constexpr`** function taking a **`std::integer_sequence`** and returning a sorted version of the sequence. I've deliberately removed as many complications as possible in order to make the technique clear. I leave it to the reader to generalise such things as comparison function (for now, **<**), or the type of thing being sorted (for now, just any integral types). The best way to explain is to walk through the code so without further ado, here goes.

The code

Need these for **`index_sequence`** etc...

```
#include <utility>
#include <array>
```

The public interface takes a sequence and we will see that it returns a sequence. Remember that this is meta programming and really it's all about the types rather than the values.

```
template<typename Int, Int... values>
  constexpr auto
sort(std::integer_sequence<Int, values...>);
```

The pretty interface hides an implementation. This is defined as a **`struct`**. This is just syntactic sugar and saves us having to repeat some common declarations.

```
template<typename Values> struct SortImpl;
```

Our wrapper function passes on the sequence and calls the implementation (see Listing 1).

```
template<typename Int, Int... values>
constexpr auto sort(std::integer_sequence<Int,
  values...> sequence)
{
  return SortImpl<decltype(sequence)>::sort();
}
```

Listing 1

Now the guts. We use partial specialisation to break out the sequences of values and indices.

```
template<typename Int, Int... values>
struct SortImpl<std::integer_sequence<Int,
  values...> >
{
```

Create an index corresponding to the positions in the sorted sequence and call an implementation.

```
static constexpr auto sort()
{
  return sort(std::make_index_sequence
    <sizeof...(values)>{});
}
```

A sorted sequence is one where the positions of the elements correspond to the ranking of the elements' values. By ranking, I mean the order defined by the comparison function (in this case **<**). In other words, position 0 has element with lowest value (rank 0), position 1 has element with rank 1, etc. In general the i^{th} position contains the i^{th} ranking element. Here the index parameter pack gives us all the values of *i* so we can write that in C++ like this:

```
template<std::size_t... index>
static constexpr auto
  sort(std::index_sequence<index...>)
{
  return std::integer_sequence<Int,
    ith<index>()...>{};
}
```

The i^{th} element is the value whose rank is *i*. We can find this by looking at all the values and picking out the one with the correct rank. We have to be a little bit careful though. Repeated values will lead to ties in ranking. eg for the sequence [1, 2, 2, 3] the ranks are 1st, 2nd, 2nd, 4th. We can compensate for this by taking into account the count of each value. In Listing 2 (overleaf), I'm using a side effect within the pack expansion to capture the result.

We can define the rank of an element by counting the number of other elements of lesser value. Note if you we going to generalise the ordering function this is where you would do it.

```
template<Int x>
static constexpr auto rankOf() {
  return ((x > values) +...); }
```

Norman Wilson has been coding since he was a spotty teenager in the early 80s and learned C++ while at university. Since then he's spent most of his career in finance. When not staring at template error messages, he rock climbs, makes music and helps bring up three daughters. You can contact him at norman.wilson+accu@gmail.com

A sorted sequence is one where the positions of the elements correspond to the ranking of the elements' values.

```
template<std::size_t i>
static constexpr auto ith()
{
  Int result{};
  (
    (i >= rankOf<values>() &&
     i < rankOf<values>() + count<values>() ?
    result = values : Int{}),...);
  return result;
}
```

Listing 2

The count is similar.

```
template<Int x>
static constexpr auto count() {
  return ((x == values) +...); }
};
```

To show that it works, I'm defining equality for `integer_sequence`s. Two sequences with the same values are equal.

```
template<typename Int, Int... values>
constexpr auto operator==(
  std::integer_sequence<Int, values...>,
  std::integer_sequence<Int, values...>) {
    return true; }
```

Sequences with different values are unequal (see Listing 3).

As an extra check, this bit of code converts a sequence to an array.

```
template<typename Int, Int... values>
constexpr auto toArray(std::integer_sequence<Int,
  values...>)
{
  return std::array<Int, sizeof...(values)>{
    values... };
}
```

```
template<typename Int, Int... values,
  Int... others>
constexpr auto operator==(
  std::integer_sequence<Int, values...>,
  std::integer_sequence<Int, others...>) {
    return false; }
static_assert(
  sort(std::index_sequence<3, 2, 1>{}) ==
    std::index_sequence<1, 2, 3>{});
static_assert(
  sort(std::index_sequence<3, 3, 1>{}) ==
    std::index_sequence<1, 3, 3>{});
```

Listing 3

In godbolt [Godbolt], we can see the emitted code is sorted.

```
auto x = toArray(sort(std::index_sequence<3, 2,
  1, 9, 42>{}));
```

Is this better than the equivalent recursive definition? I think it's easier to understand and it's shorter. Is it quicker? Technically it would be n^3 since for each element we're finding the i^{th} which involves looking at the rank of each element which requires comparing each element. But since each of these calculations is a template instantiation, the compiler will cache these intermediate values. I think it's actually n^2 but with lower overhead than recursive techniques which are likely to be $n \log(n)$. ■

References

[Godbolt] Matt Godbolt administers 'Compiler Explorer', and this article and code can be found at: https://godbolt.org/z/BeMHZe

[Wilson] 'C++ calculate and sort vector at compile time', posted on stackoverflow at https://stackoverflow.com/questions/32660523/c-calculate-and-sort-vector-at-compile-time

The full text and code of this article are available on Github: https://github.com/abwilson/compile_time_sort_article

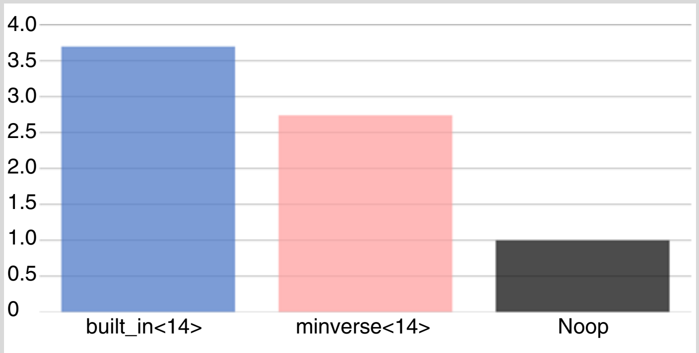
Quick Modular Calculations (Part 1)

Compilers are good at optimising modular calculations. Can ~~we~~ they do better? Cassio Neri shows they can.

Searching for 'fast divisibility' on *Stack Overflow* indicates that the question of whether hand-written C/C++ code can be faster than using the % operator fosters the curiosity of a few developers. Some answers basically state "Don't do it. Trust your compiler." I could not agree more with the principle. Compilers do a great job of optimising code and programmers should favour clarity. Nevertheless, this series shows ways to get improved performance when evaluating modular expressions. The intention here is not to 'beat' the compiler. On the contrary, this series is an open letter addressed to compiler writers presenting some algorithms that, potentially, could be incorporated into their product for the benefit of all programmers. Performance analysis shows that the alternatives discussed in this series are often faster than built-in implementations. With the exception of one algorithm which (to the best of my knowledge) is original, the others are not. Indeed, they have been covered by the classic *Hacker's Delight* [Warren] for more than a couple of years. Nevertheless, major compilers still do not implemented them.

Warm up

We start looking at an example of the improvement that can be achieved. Figure 1[1] graphs the time it takes to check whether each element of an array of 65536 uniformly distributed unsigned 32-bits dividends in the interval [0, 1000000] leaves a remainder 3 when divided by 14.



ratio (CPU time / Noop time)
Lower is faster

Figure 1

This is performed by two in-line functions: the first one, labelled *built_in*, contains a single return statement:

```
return n % 14 == 3;
```

The one labelled *minverse* implements the *modular inverse* algorithm (*minverse* for short) which is this article's subject. Both measurements include the time to scan the array of dividends without performing any modular calculation. This time is also plotted against the label *Noop* and it is used as unit of time. The times[2] taken by *minverse* and the built-in algorithms are, respectively, 2.74 and 3.70. Adjusted times (obtained by subtracting the array scanning time) are then 2.74 - 1.00 = 1.74 and 3.70 - 1.00 = 2.70, which implies a ratio of 1.74 / 2.70 = 0.64. This represents a respectable 36% performance improvement with respect to the built-in algorithm. This ratio largely depends on the divisor, as will be made clear later on in this article.

```
built_in
   0:   89 fa                  mov    %edi,%edx
   2:   b9 93 24 49 92         mov    $0x92492493,%ecx
   7:   d1 ea                  shr    %edx
   9:   89 d0                  mov    %edx,%eax
   b:   f7 e1                  mul    %ecx
   d:   c1 ea 02               shr    $0x2,%edx
  10:   6b d2 0e               imul   $0xe,%edx,%edx
  13:   29 d7                  sub    %edx,%edi
  15:   83 ff 03               cmp    $0x3,%edi
  18:   0f 94 c0               sete   %al
  1b:   c3                     retq
```

Listing 1

```
minverse
 0: 83 ef 03             sub    $0x3,%edi
 3: 69 ff b7 6d db b6 imul   $0xb6db6db7,%edi,%edi
 9: d1 cf                ror    %edi
 b: 81 ff 92 24 49 12 cmp    $0x12492492,%edi
11: 0f 96 c0             setbe  %al
14: c3                   retq
```

Listing 2

Listings 1 and 2 contrast the code generated by GCC 8.2.1 (optimisation level at `-O3`) for the two algorithms. Although code size is not a perfect indication of speed, these two aspects are highly correlated.

Cassio Neri has a PhD in Applied Mathematics from Université de Paris Dauphine. He worked as a lecturer in Mathematics before moving to the financial industry. He can be contacted at cassio.neri@gmail.com.

1. Powered by quick-bench.com [qb]. For readers who are C++ programmers and do not know this site, I strongly recommend to check it out. In addition, I politely ask all readers to consider contributing to the site to keep it running. (Disclaimer: apart from being a regular user and donor, I have no other affiliation with this site.)

2. YMMV, reported numbers were obtained by a single run in quick-bench.com using GCC 8.2 with `-O3` and `-std=c++17`.

the question of whether hand-written C/C++ code can be faster than using the % operator fosters the curiosity of a few developers

Preliminaries

This series of articles concerns the evaluation of modular expressions where **the divisor is a compile time constant and the dividend is a runtime variable. Dividend and divisor have the same unsigned integer type.**

The algorithms were implemented in a C++ library called **qmodular** [qmodular], short for *quick modular*. They can be implemented in standard C/C++ but **qmodular** uses a few GCC extensions including inline assembly. Our discussion focuses on GCC 8.2.1 for an x86_64 target but it is applicable to other compilers and architectures.

A fundamental reason for the currently built-in algorithm not being the most performant is that often it needlessly calculates remainders. For instance, `n % d == m % d` is, roughly speaking (more on this later), equivalent to `(n - m) % d == 0`. In this approach, there is no need to compute either `n % d` or `m % d`. However, the compiler evaluates both and ends up doing two modular calculations rather than one.

The built-in algorithm

For any given divisor, the compiler selects the best algorithm it knows for this particular value. Undoubtedly, the best case is where `d` is a power of two and `n % d` is evaluated as `n & (d - 1)`. This is a classic bitwise trick [Warren]. For other divisors, `n % d` is evaluated by the equivalent expression `n - (n / d) * d`. (Recall that `/` is integer division.) Hence, `n % d == r` becomes `n - (n / d) * d == r`.

The most important and widely implemented optimisation replaces the expensive division with a multiplication (instruction `mul` in Listing 1) by a 'magic' number (the constant `0x92492493`) and other cheap operations. Additional more trivial micro-optimisations are also applied. Two examples of such micro-optimisations follow.

When `r` is known to be zero at compile time, the subtraction is avoided and the expression is evaluated as `n == (n / d) * d`.

As mentioned earlier, when evaluating `n % d == m % d`, the compiler computes both remainders and, for each of them, performs a multiplication by the 'magic' number. Rather than twice loading this number into a register, this is done only once.

The modular inverse algorithm

This section covers the basics of the *minverse* algorithm by means of an example. A more detailed exhibition, including a mathematical proof of correctness, is provided in [qmodular]. We are interested in the case where `n`, `d` and `r` are unsigned integer numbers usually of 32 or 64 bits. For easy of exposition, we assume the number of bits is just 4 and hence, $n \in \Omega_4 = \{0, \ldots, 15\}$, which is a set small enough to allow manual inspection of all possible dividends. For this example we set the divisor `d` = 6.

Any strictly positive integer is a product of a positive odd number and a power of two. The algorithm starts by finding this decomposition for the divisor. In our example, $d = 6 = 3 \cdot 2^1$ and we set $h = 3$ and $k = 1$ (the exponent of 2). Since h is odd, by well known arithmetical results, there exists a unique $g \in \Omega_4$ such that $g \cdot h \equiv 1 \pmod{2^4}$. Indeed, $11 \cdot 3 = 33 = 2 \cdot 2^4 + 1$ and

n	$11 \cdot n \pmod{2^4}$	$(11 \cdot n)_2$	$(ror(11 \cdot n, 1))_2$	$ror(11 \cdot n, 1)$
0	0	0000	0000	0
6	2	0010	0001	1
12	4	0100	0010	2
2	6	0110	0011	3
8	8	1000	0100	4
14	10	1010	0101	5
4	12	1100	0110	6
10	14	1110	0111	7
3	1	0001	1000	8
9	3	0011	1001	9
15	5	0101	1010	10
5	7	0111	1011	11
11	9	1001	1100	12
1	11	1011	1101	13
7	13	1101	1110	14
13	15	1111	1111	15

Table 1

thus $g = 11$. More generally, the modulus can be any power of two, 2^w. (Again, the cases of practical interest are $w = 32$ and $w = 64$ but here we take $w = 4$.) Numbers g and h are said to be *modular inverse* (modulo 2^w) of one another and hence the algorithm's name.

For any $n \in \Omega_w = \{0, \ldots, 2^w - 1\}$, let $ror_w(n, k) \in \Omega_w$ be obtained by rotating the bits of n by k positions to the right. For instance, $ror_4(3, 2) = 12$ since $3 = (0011)_2$ and $12 = (1100)_2$.

Given $n \in \Omega_w$, the algorithm proceeds with the calculation of $ror_w(g \cdot n, k)$. Table 1 shows detailed steps to get $ror_4(11 \cdot n, 1)$ for every $n \in \Omega_4$. (Rows are ordered by the last column.) One can easily inspect two crucial properties of the map $n \rightarrow ror_w(g \cdot n, k)$: it is bijective and for multiples of d it matches division by d, that is, $ror_w(g \cdot d \cdot i, k) = i$. (See first three rows.) This map has other interesting[3] properties but the two cited here are enough for the algorithm to work.

Let N_r be the number of elements of Ω_w that are equivalent to $r \pmod d$ and $n = d \cdot i + r$, with $0 \le i < N_r$, be one of them. Then $n - r = i \cdot d$ and we obtain $ror_w(g \cdot (n - r), k) = i < N_r$. Reciprocally, if $n \in \Omega_w$ is not equivalent to $r \pmod d$, then $n - r$ is not multiple of d and $ror_w(g \cdot (n - r), k) \ge N_r$.

3. More generally, for any r (not necessarily zero), numbers leaving remainder r are strictly increasingly mapped into (disjoint) intervals. For instance, 1, 7 and 13 leave remainder 1 and are mapped to {13, 14, 15}, whereas 4 and 10 leave remainder 4 and are mapped into {6, 7}. Boundaries from shaded to unshaded rows mark remainder changes.

The most important and widely implemented optimisation replaces the expensive division with a multiplication by a 'magic' number

In our example $d = 6$. Take, for instance, $r = 2$ and note the numbers in Ω_4 that leave this remainder when divided by $d = 6$, namely, 2, 8 and 14. From this we get $N_r = 3$. Subtracting $r = 2$ from these three numbers yields 0, 6 and 12, respectively. They are mapped by $n \to ror_w(g{\cdot}n, k)$ into 0, 1 and 2, which are exactly the elements of Ω_4 smaller than $N_r = 3$.

In summary, given $d = h{\cdot}2^k$, with h odd and $0 \le r < d$, we have $n \equiv r \pmod{d}$, if and only if $ror_w(g{\cdot}(n - r), k) < N_r$, where g is the modular inverse of $h \pmod{2^w}$ and N_r is the number of elements of $\{0, \ldots, 2^w - 1\}$ that are equivalent to $r \pmod{d}$. Notice that g and k depend on d only and can be computed at compile time. However, N_r depends on d and r. It can be known at compile time provided that r is.

At this point, Listing 2 becomes much less cryptic as the code generated for

```
unsigned k = 1, g = 0xb6db6db7, r = 3,
  N = 0x12492493;
return ror(g * (n - r), k) <= N - 1;
```

The snippet above calls the function `ror`, defined in **qmodular**, which is translated to the assembly instruction with the same name.

As in the built-in case, micro-optimisations can be applied. For instance, when `r == 0` the subtraction from `n` can be elided. The same holds for the multiplication, when `g == 1`, and for `ror`, when `k == 0`. These optimisations are performed by the compiler and we should not worry about doing them ourselves. However, there is another trick that the compiler cannot figure out by itself. It is based on a property of our map that we have not yet used. To each divisor, there corresponds a special remainder $s \neq 0$ for which the subtraction can be eliminated provided the comparison is reversed. More precisely, $n \equiv s \pmod{d}$, if and only if $ror_w(g{\cdot}n, k) \ge 2^w - N_r$. For instance, in Table 1 we see the three numbers leaving remainder 1, namely, 1, 7 and 13 are mapped, respectively, into 13, 14 and 15. Hence, the remainder is 1, if and only if the result is greater than or equal to $13 = 16 - 3 = 2^w - N_r$.

For evaluation of `n % d == m % d`, we use the fact that $n \equiv m \pmod{d}$, if and only if $n - m \equiv 0 \pmod{d}$. Because unsigned types implement modulo 2^w arithmetic (not modulo d) extra care must be taken for the subtraction not to underflow. Hence, we apply the test to either $n - m$ or $m - n$ depending on whether $n \ge m$ or not, that is, `n % d == m % d` can be evaluated as follows:

```
ror(g * (n >= m ? n - m : m - n), k) < N;
```

In reality, digressing from the modular inverse algorithm, this last reasoning equally applies to the built-in algorithm:

```
(n >= m ? n - m : m - n) % d == 0;
```

This way, the compiler calculates just one remainder instead of two as we have mentioned earlier. For the sake of completeness, the performance analysis includes the expression above.

The unbounded case

An implicit assumption of the previous section is that remainders are bounded, $r < d$, otherwise the modular inverse algorithm does not work. Since this condition does not always hold, *qmodular* implements two functions similar to those in Listing 3. The first one is as in the previous section and assumes `r < d`. The second function tests the condition and delegates the call to the first one.

```
bool
modular_inverse_bounded(unsigned n, unsigned r)
{
  /* ... */
}

bool
modular_inverse_unbounded(unsigned n, unsigned r)
{
  return r < d & modular_inverse_bounded(n, r);
}
```

Listing 3

When `r` is known at compile time the compiler can figure out by itself whether `r < d` or not. In the affirmative case, the test is removed altogether and the second function simply calls and returns the result of the first one. Similarly, the negative case is reduced to `return false`.

Some readers might be thinking the `&` (bitwise *and*) in Listing 3 is a typo and `&&` (logical *and*) was intended. In reality, this is intentional. Due to short circuiting, `&&` yields a branch and performance suffers accordingly. Since computations in `modular_inverse_bounded` are cheap, it is much more efficient to carry them out rather then aborting evaluation when the first condition is false.

Performance analysis

As in the warm up, all measurements shown in this section concern the evaluation of modular expressions for 65536 uniformly distributed unsigned 32-bits dividends in the interval [0, 1000000]. Remainders can be either fixed at compile time or variable at runtime. Charts show divisors are on the x-axis and time measurements, in nanoseconds, on the y-axis. Timings are adjusted to account for the time of array scanning.

For clarity we restrict divisors to [1, 50] which suffices to spot trends. (Results for divisors up to 1000 are available in [qmodular].) In addition, we filter out divisors that are powers of two since the bitwise trick is undoubtedly the best algorithm for them. The timings were obtained with the help of Google Benchmark [Google] running on an AMD Ryzen 7 1800X Eight-Core Processor @ 3600Mhz; caches: L1 Data 32K (x8), L1 Instruction 64K (x8), L2 Unified 512K (x8), L3 Unified 8192K (x2).

Figure 2 concerns the evaluation of `n % d == 0` and plots times taken by the built-in and *minverse* algorithms. The latter is clearly faster than the former. *Minverse*'s pretty regular zigzag is due to the microoptimisation that removes `ror`, if and only if $k = 0$. That is, `ror` is used for all even divisors and only for them.

Figure 3 covers `n % d == r` where `r` is variable and uniformly distributed in $[0, d)$. Despite the validity of precondition `r < d`, the compiler cannot know this and the boundness check is kept in the assembly code. Compared

For most divisors, the **built-in algorithm is faster** than minverse ... there are **exceptions**, though

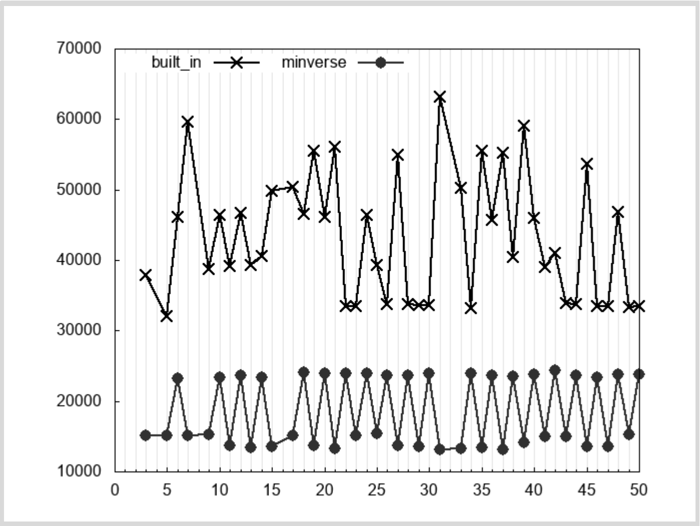


Figure 2

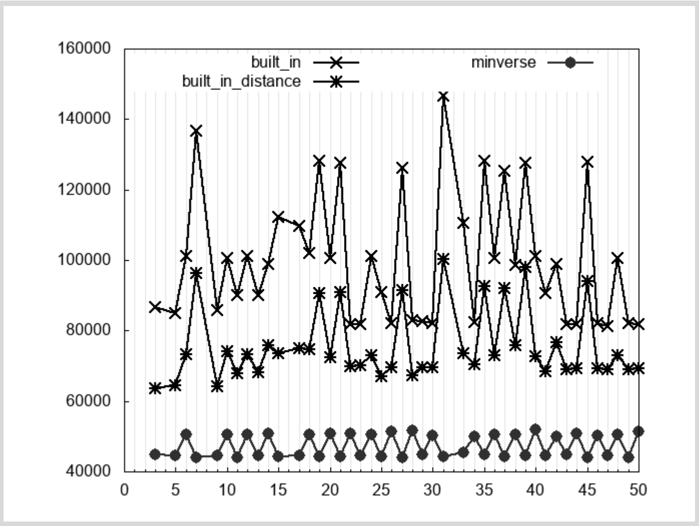


Figure 4

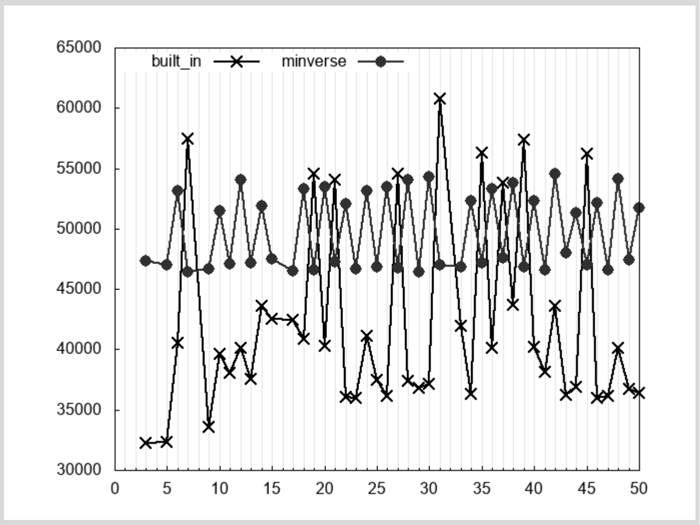


Figure 3

to the case of a fixed remainder, the built-in's performance barely changes whereas *minverse* becomes visibly slower. This is due to the computation of N_r now being performed at runtime.

For most divisors the built-in algorithm is faster than *minverse*. There are exceptions, though. Such divisors, up to 50, can be clearly seen in the chart and are 7, 19, 21, 27, 31, 35, 37, 39 and 45. The blame lies in the 'magic' number. For these divisors, corresponding 'magic' numbers do not fit in 32-bits registers. Consequently, they are truncated and more instructions are needed to correct the result. (I would like to point out to an interesting work [ridiculous_fish] which suggests alternative ways of dealing with such divisors. An idea very much worth exploring.)

Finally, Figure 4 considers the expression `n % d == m % d` where both `n` and `m` are variable. As announced earlier, we also consider the variation that uses the absolute difference of `n` and `m`:

```
(n >= m ? n - m : m - n) % d == 0;
```

This variation is labelled *built_in_distance*. Comparing this to the plain built-in algorithm we can see how effective this simple optimisation is. Nevertheless, *minverse* is still the fastest (it also uses the trick above).

Good and bad news regarding GCC 9.1

Starting at version 9.1, GCC implements the modular inverse algorithm. This welcome fact can be easily verified in [godbolt] by comparing the code generated for the following snippet with the code shown in Listing 2:

```
bool f(unsigned n) {
  return n % 14 == 3;
}
```

With GCC 8.2 for a x86-64 machine we obtain the code in Listing 4, and using GCC 9.1, we obtain the code in Listing 5.

A small difference between *qmodular*'s implementation described here and GCC's is worth mentioning because of its performance implications: while *qmodular* computes $g\cdot(n - r)$, GCC computes $g\cdot n - g\cdot r$. It would be very costly to compute $g\cdot r$ at runtime and, therefore, GCC only selects the *minverse* algorithm when the remainder is a compile time constant. (Even though and as we have seen, for some divisors *minverse* can beat the classic built-in algorithm.) Second, when r is a compile time constant it appears as an immediate value in the assembly code seen in Listing 2. In GCC's

This work is intended to help compiler writers consider when they should use the modular inverse algorithm

```
movl %edi, %edx
movl $-1840700269, %ecx
shrl %edx
movl %edx, %eax
mull %ecx
shrl $2, %edx
imull $14, %edx, %edx
subl %edx, %edi
cmpl $3, %edi
sete %al
ret
```

Listing 4

```
imull $-1227133513, %edi, %edi
subl $613566757, %edi
rorl %edi
cmpl $306783378, %edi
setbe %al
ret
```

Listing 5

implementation we see $g{\cdot}r$ instead. Now, for (not so) small divisors and dividends (which is probably the most common case in practice), r is also small but $g{\cdot}r$ is large. As it turns out, there are limitations on 64-bit immediate values which might force GCC's implementation to use more instructions and registers.

Unfortunately, there seems to be yet another other issue with GCC's implementation of *minverse*. For instance, for reasons unknown to me, changing the remainder from `3` to `4` is enough for the generated assembly to fall back to the old algorithm. Finally, regarding `n % d == m % d`, GCC does not use *minverse* at all and, instead, it computes the two remainders and compares them.

Conclusion

This article presents the modular inverse algorithm that can be used to evaluate expressions `n % d == r` and `n % d == m % d`. This algorithm is not new [Warren] but has not been widely implemented by compilers. Version 9.1 of GCC does implement it but there is room for further improvements. This work is intended to help compiler writers consider when they should use the modular inverse algorithm rather than what they currently use.

The greatest drawback regarding the *minverse* algorithm is that, to the best of my knowledge, it can not be efficiently used for other types of expressions like `n % d < r` or `n % d ≤ r`. These expressions and alternative algorithms for their evaluations are covered in Part II and Part III of this series. ■

Acknowledgements

I am deeply thankful to Fabio Fernandes for the incredible advice he provided during the research phase of this project. I am equally grateful to the *Overload* team for helping improve the manuscript.

References

[godbolt] https://godbolt.org/z/a7wr3U

[Google] https://github.com/google/benchmark

[qb] http://quick-bench.com/mhIaqB1ZvsBVROTGaO9opgx5ZTE

[qmodular] https://github.com/cassioneri/qmodular

[ridiculous_fish] ridiculous_fish, Labor of Division (Episode III): Faster Unsigned Division by Constants, October 19th, 2011, available at: http://ridiculousfish.com/blog/posts/labor-of-division-episode-iii.html

[Warren] Henry S. Warren, Jr., *Hacker's Delight,* Second Edition, Addison Wesley, 2013

Afterwood

We are aware of the film *Get Carter*. Chris Oldwood asks if it should be called *Acquire Carter* instead.

There is a three-letter word commonly used as a prefix in programming for naming functions and methods that has the propensity to really irk me – get. It's not alone in this as its partner in crime – set – also has a similarly undesirable effect but features less regularly and therefore avoids most of my ire. To me, the overuse of 'get' speaks volumes about the level of thought that commonly goes into its selection, i.e. none. Hence I often wonder if the title of the classic 1970s gangster movie *Get Carter* had suffered a similar affliction and whether there are other choices to be made that would better convey the artists' intent?

The most obvious starting point for this discussion would be to assume that to 'Get' Jack Carter would be to make some kind of journey to where he is situated. Maybe we only need to know his whereabouts and therefore we merely need to *Find Carter*. Perhaps once his position is secured we only intend to chat with him, a quiet word in his shell-like ear so to speak, and if the journey is the focus of our story then we might prefer *Locate Carter* instead. Gangsters in films have a habit of ending up in a ditch or grave in the woods and therefore while *Unearth Carter* might be grittier it could also give some of the game away to the audience before we've even started, so perhaps we should settle for the less revealing *Discover Carter* instead. (Let us put aside for a moment the words of George Box and hope that this analogy, while wrong, like every other, is still at least somewhat useful, somehow, at some point.)

What is more likely though is that it's not enough to simply know where he is; a gang's head honcho probably needs to 'interact' with him in a violent way, perhaps to extract information about his enemy's intentions, in which case we not only need to find him but bring him back too. This naturally leads us to *Fetch Carter*. Jack is no doubt fairly suspicious of the people he deals with and therefore it would do little good to try and *Request Carter* without following up in person lest it be met with a metaphorical 404 (one or two finger salute) or just time-out waiting for a response (he's done a runner). Pubs seem a particularly common residence for gangsters so *Extract Carter* might convey more clearly the level of force required to obtain his full attention.

Maybe though we're thinking about the need for Jack Carter in the wrong way; do we actually require *the* Jack Carter or will someone *like* Jack Carter do instead? What if just need 'a' Jack Carter – someone who fulfils the same role by having the same knowledge and skills, what if there was a pool of Jack Carters, would it be enough to *Acquire Carter* and 'go to work' on whoever we've been given in the hope that we'll still get our desired outcome?

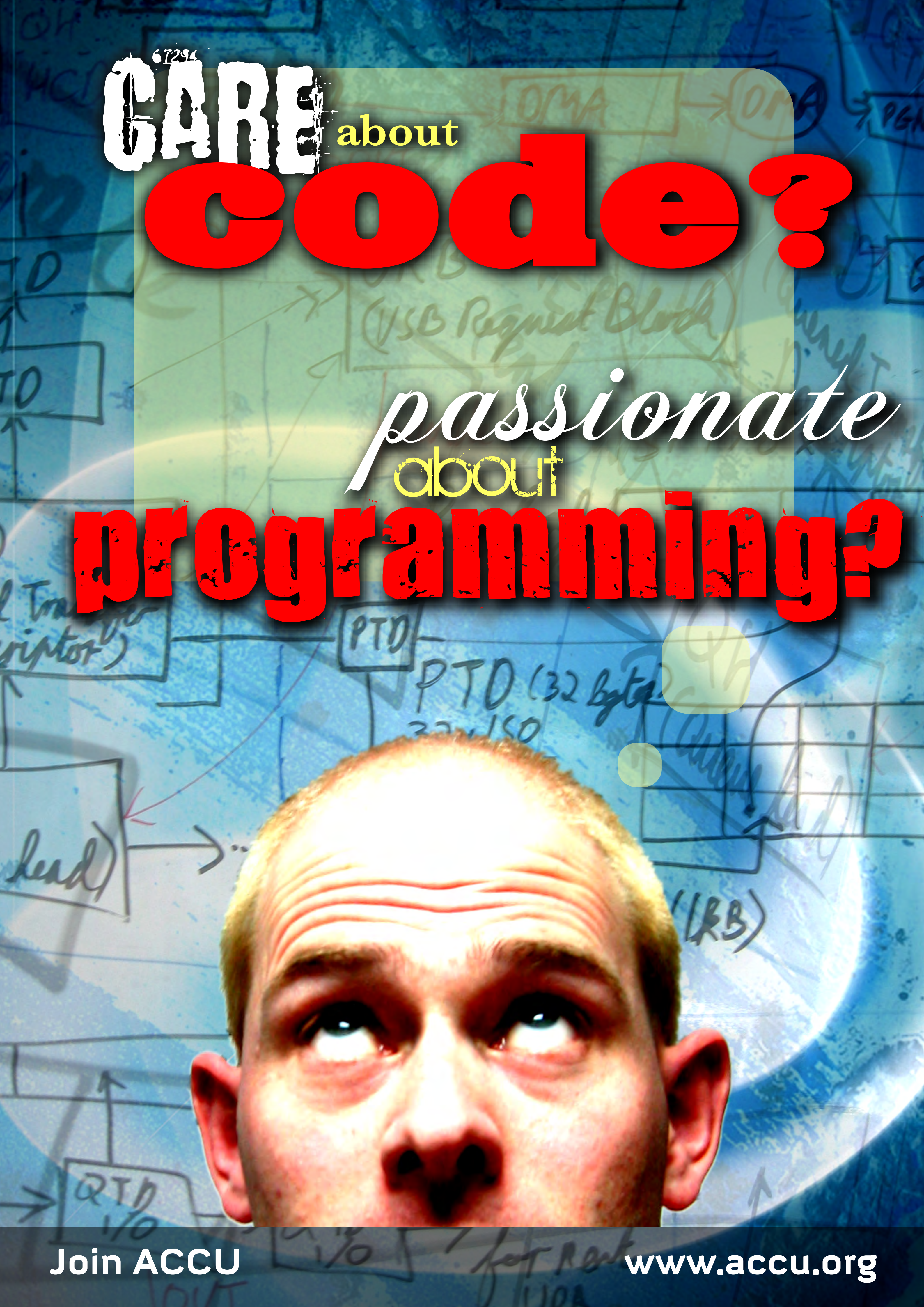
This raises another interesting philosophical question about Jack Carter, what if it's not about needing him physically, but about comprehending what makes him tick, what if 'getting' him really means *Deconstruct Carter* or *Read Carter*? Michael Caine's character is surely a product of his actions and interactions with many people over his life; so what if the emphasis in the title is on the 'creation' process, e.g. *Allocate Carter*? This seems a little too raw and so I wonder whether he was a product of his own destiny or moulded by circumstances outside his control; was there a higher power attempting to *Make Carter* or *Build Carter*? But people aren't buildings formed according to a blueprint, they're shaped over time, carefully revealed like Michelangelo's David, to wit we should add *Sculpt Carter* to the ever growing list of far more expressive terms.

Realistically, though, it's a 1970s gangster movie and that means there is going to be plenty of 'claret' spilled as everyone ultimately meets a grisly demise. Hence we find the most plausible variation of 'get' if we say the film's title in a classic deep London accent. In this sense to 'get someone' is to chase after them, on the premise that you're going to do them a serious amount of harm at the very least, while death is the more inevitable outcome. In this sense, *Kill Carter* might be more representative, if (once again) a little spoiler-ish. (I wonder if the working title for Quentin Tarantino's two-part volume staring Uma Thurman was the more ambiguous *Get Bill*?) Gangsters don't tend to do things by half; they like to go over the top for dramatic effect, which would lead to the more extensive *Destroy Carter*. (Such is the level of ambiguity here that it's interesting to note we've covered both birth and death using the same word.)

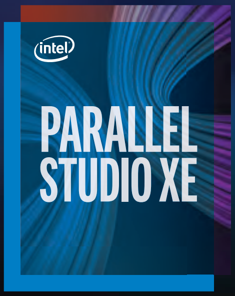
Really though we've barely scratched the surface on this topic as we've tried to fit in (albeit dubiously) within the confines of the genre in question. If we move into the realms of science fiction we can easily see a variety of plots that could give rise to a bunch of alternatives. What if our aforementioned 'Mr Big' wanted to build an entire army of Jacks that were at his beck-and-call to help build an Empire, we could look to *Clone Carter*. Or maybe his raw intellect is all that's desired to satisfy some hare-brained AI project and *Derive Carter* is a modern-day tale of Dr Frankenstein playing with neural nets and machine learning.

Okay, I'll stop now with the flights of fancy and take George Box's observation a little more seriously. The title of a film is a hook, its ambiguity is a selling point designed to draw you in and explore it. That's almost the opposite of what naming functions and methods is about – they should provide you with a good indication of their purpose without you needing to either read the documentation or worse, the implementation. Granted it can be difficult to convey subtleties with only a single verb (or handful of words) but the difference between finding and creating, calculating and cloning, or reading and formatting is already pretty substantial and gives the reader a fighting chance of understanding at an abstract level what your big picture is. Remember: just because it's called code doesn't mean it has to be cryptic. Get it? ■

Chris Oldwood 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


CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org

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