overload169
JUNE 2022 £4.50

ACCU Conference 2022

Attendees share their experiences as the ACCU Conference returns as a hybrid event, with many attending in person

Compile-time Wordle in C++20

Vittorio Romeo introduces wordlexpr, playing the game using compiler error messages

Performance Considered Essential

Lucian Radu Teodorescu argues that performance is not just important but is actually the most important thing

Afterwood

Chris Oldwood pulls a few threads

A magazine of ACCU
ISSN: 1354-3172

Join ACCU

Run by programmers for programmers,
join ACCU to improve your coding skills

- A worldwide non-profit organisation
- Journals published alternate months:
  - *CVu* in January, March, May, July, September and November
  - *Overload* in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website

professionalism in programming

www.accu.org

OVERLOAD 169

June 2022

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in *Overload* 170 should be submitted by 1st July 2022 and those for *Overload* 171 by 1st September 2022.

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.

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

Copyrights and Trademarks

What Happened to Demo 13?

Making mistakes and forgetting are facts of life. Frances Buontempo tries to find ways to tackle this.

I've not had time to write an editorial because I attended the ACCU conference in person this year and spoke. Travelling is surprisingly time consuming and tiring, especially if you've not done it in a while. Furthermore, being around crowds of people can be overwhelming when you are out of practice. My talk was about traffic flow and crowds of people moving in space, which involved simulations to remind me how this real life stuff works. The conference was run as a hybrid event this year, so some attended in person while others joined remotely. It worked well. If you've never been to a conference, find one and go. If you can't afford it or persuade work to pay, you can sometimes volunteer and therefore get in for free. Also, don't forget you get a discount for the ACCU conference if you are an ACCU member.

You will be unsurprised to learn my talk involved coding your way out of a paper bag. I have been using this as a toy problem to play around with various machine learning and simulation algorithms for a very long time now. Each attempt involves a paper bag and some blobs moving around trying to get out of the bag. Sometimes the blobs are 'particles' which either simulate Brownian motion [Wikipedia-1], or follow a particle swarm optimization [Wikipedia-2]. Sometimes they are ants, forming an ant colony [Dorigo04] or a bee colony [Scholarpedia]. You could even imagine a miniature cannon firing cannon balls and use genetic algorithms to decide the best angle and velocity to fire them with [Buontempo13]. Having a go-to toy problem often fires off lots of ideas and can also help you focus on specific areas rather than trying to research everything and leaving a trail of half-finished projects. Or maybe the deadline of giving a talk helps. Or both.

Managing to think of a topic and implement something is one thing. Writing up slides is another. This isn't hard, but I'm never satisfied with them. I don't want to crowd them with too many words, but I want enough there to remind me what to say. Speaker notes don't help because trying to talk and look at people is enough multi-tasking without having to try to read as well. I also like to show a live demo of blobs moving in space, which introduces another problem. With about 16 demos, each taking various combinations of parameters, creating batch files meant I didn't need to remember which parameters to use when. I considered naming these files, but naming is hard and I didn't want to forget which order they were supposed to be in. The obvious solution was to number them instead. What could possibly go wrong? One demo per slide might have worked, but of course some slides had no demos, some had one and a few had several. Guess what? During my talk, an audience member asked a question. Great! However, I couldn't remember how real life worked, misheard, and thought something about 'Precinct 13' had been mentioned so I missed the actual question. You may be familiar with the film *Assault on Precinct 13* [IMDB], originally made by John Carpenter back in the 1970s, in which a gang siege a police station. That would be simulated by blobs surrounding a paper bag, whereas my blobs were trying to leave the paper bag at the time, so this 'question' seemed like a quip or suggestion for another talk another day. It then dawned on me I hadn't been listening properly, so I asked for the question to be repeated. Turns out, I had shown demo 12 and then demo 14, begging the question, 'What happened to demo 13?' The 13th demonstration was duly shown and we all moved on. I think I got away with it.

Making mistakes is a fact of life. Sometimes you can gloss over problems, but owning them is often better. Or owning up you didn't hear the question. By using numbers, the audience could spot I had missed something. No, not a lack of a file called `demo13.cmd`, not triskaidekaphobia, the fear of number 13, but rather an inability to count while trying to talk. Conventions, like numbering, help people follow what's happening. People can spot where you have gone wrong or forgotten something if there's some kind of pattern to follow. Conventions in naming help us spot when something seems out of place. What happened to demo 13? What happened to Windows 9? And so on. Convention and pattern recognition helps us navigate around ideas and physical space. You might expect butter to be somewhere near the bread at a buffet breakfast in a hotel. If you are looking for an even numbered house and each house you walk by is odd numbered, crossing the road may help. If you come down our end of the street that won't work. Our house has a name rather than a number, which means a local taxi firm refuses to come here because their computer system can't cope with addresses without building numbers. Furthermore, all the front doors down the road from us are even numbered, starting at 4 and ending at 12. What happened to number 2? I have no idea. And just to keep you on your toes, numbers 49 and 51 are opposite us. I suspect there may be 'historical reasons' for the unconventional numbering, though I can't be sure what. I've been trying to find my way round a large, unfamiliar code-base recently. I resorted to asking a colleague for help locating some code and was told the source file was in an unusual place 'for historical reasons'. This must be a euphemistic way of saying there is no sense in how things currently are, though they have ended up this way because changes made over time, including refactoring and repositioning, may have left a few bits and bobs in unusual places.

So-called 'legacy' code bases can be very difficult to work with, particularly if you don't have the bandwidth to make them less confusing. Some say a complete re-write might be better than tinkering with the spaghetti mess. Others might say: 'Nuke it from orbit, it's the only way to

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.

be sure.' Some of you may be familiar with this phrase, based on a quote from the film *Aliens* [IMDB2], "*I say we take off and nuke the entire site from orbit. It's the only way to be sure.*" For me, it's a stock phrase to use about certain types of code, so I was taken aback when I said this once and was told off for talking about nuclear weapons. The trouble with cultural references is they don't work if a group of people don't have a shared taste in films and the like. It's easy to forget your go-to memes and quotes may not be universally understood. I'm aware I have mentioned a couple of films as I am writing, which you may or may not know, hence the inclusion of an IMDB link. *Overload* deliberately follows the academic style of including references so you can check writers' claims and do further reading around a subject. This is not just convention for convention's sake. Context and background help communication and learning, and sometimes force writers to check they are actually right.

Conventions also tend to come and go. Consider 'Here come the beards', which often gets rolled out when older programmers speak up. So, what happened to all the women? OK, we know what the phrase means; however, not all old people grow beards and in fact many young men are growing beards and wearing their hair up in a bun. Things change and our stock-phrases might need to change accordingly. Let's not hold on to them for historical reasons.

Conventions can be a help. As discussed, the layout of food at a buffet can enable people to move round space and find what they need. Clearly marked exits help people find their way out of buildings. Using pictures as well as words, for example in airports, is useful for people who don't speak the local language. Many usual approaches seem sensible at first sight. For example, write ups and talks frequently seem polished and don't go into false starts and dead ends. We expect to be told the good things and not the bad. *The Guardian* recently wrote about research articles [Guardian22]. Covering various aspects of scientific journals ,including paper copies potentially becoming redundant, it also talks about publication bias. If journals want positive results (this works) rather than negative results (this doesn't work), this can skew the research that gets written up. It's actually really useful to know what approaches have failed and muse on why. This can lead to new ideas or stop people wasting time on things that don't and can't work. The *Guardian* article also claims:

> Studies almost always throw up weird, unexpected numbers that complicate any simple interpretation. But a traditional paper – word count and all – pretty well forces you to dumb things down.

If you have tried to do something and it didn't work, then write an article. If you have weird unexpected numbers (or strings) then chat about it on the accu general mailing list [ACCU]. Maybe write that up too.

Patterns help us spot outliers and mistakes. Similarly protocols, such as driving on a specific side of the road, help life flow smoothly. Memes and similar can be a great, succinct way of communicating, however, it's worth taking stock once in a while to consider if times have changed or there is now a better way to do things. Our brains seem to be wired up to spot patterns. The last *Overload* had an article about Wordle [Handley22], a daily puzzle to guess a five letter word. Wordle has sprouted many similar games, including Primel, which involves guessing a five digit prime number [Primel]. Prime numbers don't have handy patterns like vowel and consonant combinations, though there are a few rules to help you get started, for example no five digit number (apart from 00002) is even and none end in a five (apart from 00005). For a choice of five digits, some can only be arranged in one way to get a prime number, for example 99991. Other combinations are not prime, for example 19999 is 7×2857. Nonetheless a surprising number of five digit choices can be arranged in several ways to make a prime. 13789 has loads. I periodically get three of the five digits in the right place and find several other prime numbers that would match the pattern. I could make a list or a little program with some regex to find out how many such numbers there are, hoping to find an amazing pattern, but I know full well I won't. It's possible to say many things about prime numbers but I've never seen any useful rules to quickly detect if a number is prime or not, let alone find a prime number containing certain digits.

Patterns can help or hinder. Much has be written about the display of information and how to avoid giving the wrong impressions. For example you may have been encouraged to use donut charts rather than pie charts, since humans seem to be better at judging distances, here the length of the donut's sections, rather than areas which a traditional pie chart uses [Robertson16]. You may also be aware of Simpson's paradox [Stanford] wherein two variables may seem to have a positive or negative (or no) correlation and yet dividing the data into subgroups and running the same analysis make the correlation disappear (or appear). Simpson's original example showed this happening with a medical treatment. For the whole population there was a 50% success rate, so no evidence of any difference in recovery between those taking the medication and those not. However, when he grouped the data by gender, the subgroups each showed a higher success rate with the treatment. Patterns can appear or disappear as your perspective shifts. If you spot an oversight or mistake, call it out. And if you spot any typos in this publication, let me know. They do slip through, even though a whole review team and the production editor try to flush them out. And finally, if you forget what you are doing, having supportive people around to help you out is tremendous. So, thanks to everyone for a great conference this year and thanks to *Overload*'s writers and review team for their hard work.

References

[ACCU] accu-general mailing list: https://accu.org/members/mailing-lists

[Buontempo13] Frances Buontempo (2013) 'How to program your way out of a paper bag using genetic algorithms' in *Overload* 118, published December 2013, available from: https://accu.org/journals/overload/21/118/buontempo_1825/

[Dorigo04] Marco Dorigo and Thomas Stützle (2004) *Ant Colony Optimization* MIT Press

[Guardian22] Stuart Ritchie 'The big idea: should we get rid of the scientific paper' published 11 April 2022 at: https://www.theguardian.com/books/2022/apr/11/the-big-idea-should-we-get-rid-of-the-scientific-paper

[Handley22] James Handley (2022) 'Taming Wordle with the Command Line' in *Overload* 168, published April 2022, available from: https://accu.org/journals/overload/30/168/overload168.pdf

[IMDB] *Assault on Precint 13* (1976): https://www.imdb.com/title/tt0074156/

[IMDB2] *Aliens* (1986): https://www.imdb.com/title/tt0090605/?ref_=nv_sr_srsg_0

[Primel] Primel game: https://converged.yt/primel/

[Robertson16] Andrea Robertson (2016) 'Pie Chart vs Donut Chart: Showdown in the Ring, published 6 December 2016, available at https://medium.com/@hypsypops/pie-chart-vs-donut-chart-showdown-in-the-ring-5d24fd86a9ce

[Scholarpedia] 'Artificial bee colony algortihm', available at: http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm

[Stanford] 'Simpson's Paradox' on *Stanford Encyclopedia of Philosophy*, published 24 March 2021, available at: https://plato.stanford.edu/entries/paradox-simpson/

[Wikipedia-1] 'Brownian motion', available at: https://en.wikipedia.org/wiki/Brownian_motion

[Wikipedia-2] 'Particle swarm optimization', available at: https://en.wikipedia.org/wiki/Particle_swarm_optimization

Performance Considered Essential

We know that performance is important. Lucian Radu Teodorescu argues that it is actually the most important thing.

I sometimes hear fellow engineers say "performance is not a concern in our project", or "performance doesn't matter". I can understand that, in certain projects, performance is not a major concern; and also that, following usual engineering techniques, performance will be in an acceptable range, without needing to dedicate time to performance related activities. But I cannot agree, even in these projects, that performance doesn't matter.

What I find more worrying is that these performance ignoring projects lead to generalisations: programmers claim that performance should be ignored while building software.

The current article tries to counter this trend and argue that all software problems are, in one way or another, a performance problem. That is, performance cannot be fully ignored. I'll give a couple of arguments as to why we can't ignore performance; among them, I attempt to prove that without the performance concern, Software Engineering would be mostly a *solved* domain.

Misinterpreting Knuth

The proponents of the idea that performance should be ignored often quote Knuth, in the simplified form [Knuth74]:

> premature optimization is the root of all evil

Just quoting this leaves out the context, which contains important details. A more appropriate quote would be [Knuth74]:

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

This is a far more nuanced quote. It's not a problem with efficiency in general, but a problem with spending too much energy on improving performance in the wrong places.

But even this larger quote does not capture Knuth's full intent. In the surrounding paragraphs, he is criticising people who condemn program efficiency [Knuth74]:

> I am sorry to say that many people nowadays are condemning program efficiency, telling us that it is in bad taste.

That is, often the simple quoting of Knuth contradicts the idea that Knuth tried to make.

Frequently, when I hear Knuth's short quote in the wrong way, I have another quote ready:

> Ignorance [is] the root and stem of all evil. ~ Plato (disputed)

Ignoring performance considerations can lead to unusable software. Moreover, as we will further explain, performance is an essential part of software engineering.

One algorithm to rule them all

In this section, we will assume that the performance of programs is not relevant at all. We are free to implement any algorithm, with any complexity, as long as it solves our problem; moreover, we should look for simpler algorithms.

We might be proving something that is obvious for many readers. We want to disambiguate some nuances and make it clear that performance is more important than we generally consider. The aim is not to simply convince software developers that performance matters, but rather to be more helpful to theoreticians of software industry. This might help in refocusing our perception of performance concerns.

Theorem. There is an algorithm that can solve all software problems, if one can define an acceptance function for such problems.

Before proving this theorem, we need some clarifications. First, we assume that all programs transform data, and we can express our algorithm as a data transformation. That is, we have input data `In`, and we are producing some output data `Out`. It is irrelevant for our goals how we encode information in the input and output data, and our treatment of time. For example, if the problem we are trying to solve is a GUI, then the inputs would be the set of key presses, mouse movements and clicks, all in relation to time; one can find a way to properly encode this into `In` data. Encoding time is not easy, but we assume that this can be done.

Another important assumption is that the problems we are trying to solve have solutions. We cannot solve a problem that can't be solved. In other words, for each problem, there is at least one sequence of bits that would be accepted for that problem.

To properly build our algorithm, we need an acceptance function. In code, the acceptance function would have the following signature:

```
bool solution_is_valid(In data, Out result);
```

Our algorithm can be represented by a function:

```
Out the_one_algorithm(In data);
```

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

We might have a complex system comprising multiple parts... But, as system parts should be simpler to encode, we can simply apply a recursive decomposition pattern to the problem of encoding

such as:

```
solution_is_valid
  (data, the_one_algorithm(data)) == true.
```

We will define only one variant of `the_one_algorithm`, and we assume that the `solution_is_valid` is given for each type of software problem we need to solve.

To define our algorithm, we consider that `In` and `Out` are bitsets with variable number of bits. We also assume that we don't have limitations on how many bits we can represent.

With all these discussed, Listing 1 is our fantastic algorithm.

Our algorithm is also known as the backtracking algorithm. We are applying it to all possible problems for which we have an acceptance criterion.

The backtracking algorithm will iterate over all possible combinations of output bits, and we are guaranteed that there is at least one sequence of bits that satisfies the acceptance function. This means that we are guaranteed that our algorithm finds our solution.

Q.E.D.

```
bool check_solution(int n, In data, Out& res);

Out the_one_algorithm(In data) {
  Out res; // initially zero bits
  while (true) {
    // keep adding bits
    res.add_bit(0);
    // backtrack until we find an acceptable
    // solution
    if ( check_solution(0, data, res) )
      return res;
  }
}
bool check_solution(int n, In data, Out& res) {
  if ( n == res.size() ) {
    return solution_is_valid(data, res);
  }
  else {
    res[n] = 0;
    if ( check_solution(n+1, data, res) )
      return true;
    res[n] = 1;
    if ( check_solution(n+1, data, res) )
      return true;
  }
  return false;
}
```

Listing 1

Algorithm analysis

Positives:

- can be used to solve any problem (that is solvable, and that has an acceptance function)
- it's simple to understand

Negatives:

- performance: complexity is $O(2^n)$, where n is the smallest number of bits for an acceptable solution

The programmers who argue that performance doesn't matter should argue that this algorithm is great, as it doesn't have any (major) negatives. It's the perfect algorithm!

However, one might argue that we are moving the problem somewhere else. As it's easy to solve the original problem, the difficulty moves towards encoding the problem and defining the acceptance function. Let's analyse these.

We might have a complex system comprising multiple parts, and it might be hard to combine them into an appropriate encoding. But, as system parts should be simpler to encode, we can simply apply a recursive decomposition pattern to the problem of encoding. That would give us a process for encoding all the problems.

Moreover, encoding is also a software problem. We can apply the same algorithm to generate solutions for it. The reader should agree with me that deciding on the encoding of the problem should be less complex than solving the complete problem.

The other aspect is the acceptance function. In most cases, this is a simpler problem than the solution of the problem itself. As this is problem-dependent, we cannot give a universal algorithm for it; but, we can certainly apply our backtracking algorithm to simplify the acceptance test too.

Thus, even if the complexity doesn't completely disappear, we've separated it into two parts, which, for the vast majority of problems, should be simpler than the entire problem (which needs to include encoding and acceptance testing).

Lehman's taxonomy

Let us analyse whether this algorithm applies to different types of programs. We use the Lehman taxonomy [Lehman80] for this analysis.

Lehman divided the set of programs that can be built into 3 types :

- S-Programs
- P-Programs
- E-Programs

S-Programs are simple programs; the specification can be used to easily derive the functionality of the program. We can derive the acceptability of the solution directly from the specification of the problem. Most mathematical problems are good examples of S-Programs.

P-Programs are slightly more complex. Precise specifications cannot be directly used to derive the acceptability of the solution. The acceptance function can be derived from the environment in which the program operates.

The example that Lehman gives for P-Programs is a program to play chess. We cannot define the quality of the program just by looking at the chess rules. The rules can be relatively simple, but applying them can generate good chess programs or terrible ones. We have to define the acceptance criteria of the program by looking at how well the program fares in competition with other actors (humans, other chess programs, etc.). The evaluation of a chess program should always be done in its operating context.

A good technique for evaluating P-Programs is comparison: we can find a reference model, and then compare the behaviour of the program with this model.

E-Programs are programs that are even more complex. They embed human activities into their output. They can't be separated from the social contexts in which they operate. Any program that has a feedback loop that includes human activities is an E-Program. A good current example of an E-Program is a road traffic program. The program gives traffic information to users; users take that into account when driving, and their driving behaviour is fed in as traffic input to the program. Users driving alternative routes will change the traffic conditions for the main routes and the alternative routes.

E-Programs should also be evaluated in their operating contexts. This time, the environment is more complex.

Kevlin Henney also frequently discusses this taxonomy in a more recent context. See for example [Henney19].

Now, let's analyse how this taxonomy affects our algorithm.

For S-Programs, we can derive the acceptance functions directly from the specification of the problem. This is the simplest case.

P-Programs are harder to deal with. We cannot derive the acceptance function directly from the specification. But, if this is a software problem, there needs to be an acceptance criterion. Here, we have a hint: it's often the case that it's easier to compare the outputs of our program with another model (by another program or involving humans). The comparison can be slow (especially if it involves humans), but for the current discussions we have said that performance can be ignored.

For the chess program example, for every possible solution of the program, after ensuring that it follows the basic rules, we can use the output to play against human players (or other computer programs). Finding a decent solution may take more than the expected lifetime of the universe, but performance is not a concern here.

Finding an acceptance function for E-Programs involves a similar process. There are some basic correctness checks that we can apply automatically, but then we can ask human opinion whether the generated program is acceptable or not. This time, it's mandatory for us to involve human feedback. Again, this can be extremely expensive, but we are entirely ignoring performance aspects.

With this, we've argued that our algorithm applies to all types of problems.

Bottom line

If there are no performance constraints, we can solve all the problems by using the above algorithm. But the time to solve the problem can be incredibly large.

Typically, the more complex the problem we are solving, the more bits we need in the output data. If we require n bits for the solution, then the time complexity of the algorithm is in the order of $O(2^n)$.

The age of the universe is 14 billion years (approx. 4.4×10^{17} seconds). Let's assume that we have a computer with the core frequency of 3.2 GHz (3.2×10^9 cycles per second). With these, we can calculate that our computer can execute about 1.4×10^{27} CPU cycles from the beginning of the universe. This number is less than 2^{91}.

This means that problems that have more than 91 bits in the output require more than the entire life of the universe to compute. But, except for trivial problems, most problems require more than 91 bits in their output.

That is, we have a simple algorithm that can solve all the problems, but for performance reasons, we cannot apply it. Thus, all software problems are, in a way, performance problems.

This might be seen as a trivial result, but it is a fundamental aspect of software engineering. It's just as fundamental as "software is essential complexity" [Brooks95]. We might be saying now that software is performance-constrained essential complexity.

Convergent perspectives

Breaking the Enigma

One of the important points in the history of programable computers was the development of the computers in World War 2 to break the Enigma cipher machine.

The Enigma machine at that time was not breakable with a brute force attack (i.e., performance limitations). As a consequence, the allies started building the Colossus computer; this was the world's first programable, electronic, digital computer [WikiColossus]. So, one way to think of it, the building of programable computers is due to performance considerations.

Furthermore, to speed up attacks on the Enigma machine, the allies developed a series of strategies to allow them to have a higher likelihood of deciphering German messages in short time [WikiEnigma]. That is, at the birth of the computer industry, the first efforts were targeting improving performance.

Performance concerns were central to the development of computers.

The sorting algorithm

Let's look at the sorting algorithms. One of the most well-studied fields in Software Engineering. Probably all software engineers spent hours at looking at various sorting algorithms. Why is that?

Bubble Sort is one of the simplest sorting algorithms. And yet, this is rarely used in practice. That's because it is an inefficient algorithm. Its complexity is $O(n^2)$; and even so, it's slower than Insertion Sort that has the same complexity.

And, Bubble Sort is not the simplest sorting algorithm. A simpler one would be BogoSort (permutation sort). This can be expressed (in Python as):

```
def bogosort(elements):
  while not is_sorted(elements):
    shuffle(elements)
  return elements
```

Very simple to understand. Yet, the performance is terrible: $O((n-1)n!)$.

This is also an indication of why performance matters a lot for Software Engineering.

Textbooks

If one wants to learn programming, one needs to learn about algorithms and data structures. One popular book for learning this is *Introduction to algorithms* [Cormen22]. The first part of the book, called 'Foundations', spends a great deal of time talking about performance of algorithms. Performance-related discussions are present before introducing the first algorithm.

This is yet another strong indication that performance is an important aspect of Software Engineering.

Full-stack development

In recent times, we have often used the phrase 'full-stack development' to refer to a combination of skills for web development that includes expertise both in frontend and backend development. But, as Kevlin Henney points out [Henney19], this is just a narrow view of the stack. If we look at the bottom of the stack, we typically exclude device driver programming, operating-system programming, low-level libraries, etc.

Now, all these lower level layers in our stack are typically written in languages like C and C++. The reason for this is performance. To have decent performance at upper levels, we need to have good performance at lower levels.

If performance was not a concern at the operating system level, we would probably have OSes that would boot up in hours on modern hardware. I don't believe this is something that is acceptable to our users.

Performance and the rest of quality attributes

Architecturally speaking, performance is a quality attribute for the software system. Other quality attributes that are generally applicable include modifiability (how easy it is to change the code), availability (what's the probability for operating the system under satisfactory conditions at a given point of time), testability (how testable is the software), security (how secure is the software system), usability (how easy is it to use the software system).

We briefly investigate here whether we can say about other quality attributes what we said about performance.

One can argue that modifiability is as important as performance. This can be argued to a large extent, but one cannot get as far as we have with performance. Once we have a framework for writing code (language, input methods, building and running), we might not need to spend too much time on modifiability. Some programs are just written once, and then never changed (rare, but it is still the case).

While we are constantly striving to improve the techniques for writing software more easily (i.e., reduce accidental complexity) this is not the dominant concern in software. As Brooks argues [Brooks95], modifiability is an accidental concern, not an essential one. Thus, at least ontologically speaking, it's not as important.

Don't get me wrong: modifiability is very important to software engineering, but it's not an essential part of it.

We won't insist on the other quality attributes: availability, testability, security, usability. There is still a lot of software for which these may not be applicable. One may not think of the availability and security of a sorting algorithm, one might choose not to test certain software, and, for certain problems, it's hard to define what usability means. These quality attributes are nowhere near as important as performance.

This leaves us with the thought that performance is the most important quality attribute for a software system, more important than modifiability (at least from a theoretical perspective).

Conclusions

Performance is at the core of Software Engineering. It's not just important, it's essential. Otherwise, this would have been a dull discipline: we have one algorithm that can be applied to all the problems, it's just a matter of defining an appropriate acceptance function. But, as we know, this is completely impractical.

To some degree, all software solutions have performance as a concern. This is proven by the entire industry. This is why we use certain algorithms (e.g., QuickSort) over others (e.g., BogoSort). This is why we continuously spend money on researching how to make our programs faster. And, this is why we have books to teach us the best algorithms we know so far.

The fact that some projects may not have important performance constraints doesn't mean they don't have performance constraints at all. It's just that, in those limited domains, it is highly unlikely to go outside those constraints. For example, sorting a 10-element integer array can be done almost in any way possible if the code needs to run in under 100ms. But, most projects aren't like that. As software tends to compose (software is *essential complexity*) inefficient algorithms, when composed, tend to extrapolate slowness; after a certain limit, software built with inefficient algorithms would be too slow.

This might have been a very long article for such a simple idea. But, unfortunately, engineering is not always glamorous, shiny and cool. Oftentimes, it ought to be boring and predictable. Yes, *predictable*, that's the word we should associate more with Software Engineering. However, that is a topic for another article (or, set of articles). ■

References

[Brooks95] Frederick P. Brooks Jr., *The Mythical Man-Month* (anniversary ed.), Addison-Wesley Longman Publishing, 1995

[Cormen22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms* (third edition), MIT press, 2022

[Henney19] Kevlin Henney, 'What Do You Mean?', *ACCU 2019*, https://www.youtube.com/watch?v=ndnvOElnyUg

[Knuth74] Donald E. Knuth, 'Computer Programming as an Art', *Communications of the ACM* 17 (12), December 1974

[Lehman80] Meir M Lehman, 'Programs, Life Cycles, and Laws of Software Evolution', *Proceedings of the IEEE* 68, no. 9, 1980, https://www.ifi.uzh.ch/dam/jcr:00000000-2f41-7b40-ffff-ffffd5af5da7/lehman80.pdf

[WikiColossus] Wikipedia, 'Colossus computer', https://en.wikipedia.org/wiki/Colossus_computer

[WikiEnigma] Wikipedia, 'Cryptanalysis of the Enigma', https://en.wikipedia.org/wiki/Cryptanalysis_of_the_Enigma

Compile-time Wordle in C++20

Wordle is everywhere. Vittorio Romeo introduces wordlexpr, using compiler error messages to play the game.

It felt wrong to not participate in the Wordle craze, and what better way of doing so than by creating a purely compile-time version of the game in C++20? I proudly present to you… Wordlexpr! [Wordlexpr]

Carry on reading to understand the magic behind it!

High-level overview

Wordlexpr is played entirely at compile-time as no executable is ever generated – the game is experienced through compiler errors. Therefore, we need to solve a few problems to make everything happen:

1. Produce arbitrary human-readable output as a compiler diagnostic.
2. Random number generation at compile-time.
3. Retain state and keep track of the player's progress in-between compilations.

Error is the new printf

In order to abuse the compiler into outputting errors with an arbitrary string of our own liking, let's start by trying to figure out how to make it print out a simple string literal. The first attempt, `static_assert`, seems promising:

```
static_assert(false, "Welcome to Wordlexpr!");
```

```
error: static assertion failed: Welcome to Wordlexpr!
    1 | static_assert(false, "Welcome to Wordlexpr!");
      |               ^^^^^
```

However, our delight is short-lived, as `static_assert` only accepts a string literal – a `constexpr` array of characters or `const char*` will not work as an argument:

```
constexpr const char*
  msg = "Welcome to Wordlexpr!";
static_assert(false, msg);
```

```
error: expected string-literal before 'msg'
    2 | static_assert(false, msg);
      |                      ^^^
```

So, how about storing the contents of our string as part of the type of a `struct`, then produce an error containing such type?

```
template <char...> struct print;
print<'a', 'b', 'c', 'd'> _{};
```

```
error: variable 'print<'a', 'b', 'c', 'd'> _'
       has initializer but incomplete type
    3 | print<'a', 'b', 'c', 'd'> _{};
```

Nice! We are able to see our characters in the compiler output, and we could theoretically mutate or generate the sequence of characters to our liking at compile-time. However, working with a `char`... template parameter pack is very cumbersome, and the final output is not very readable.

C++20's P0732R2: 'Class Types in Non-Type Template Parameters' [P0732R2] comes to the rescue here! In short, we can use any *literal type* as a non-type template parameter. We can therefore create our own little compile-time string literal type (Listing 1).

```
struct ct_str
{
  char        _data[512]{};
  std::size_t _size{0};
  template <std::size_t N>
  constexpr ct_str(const char (&str)[N])
    : _data{}, _size{N - 1}
  {
    for(std::size_t i = 0; i < _size; ++i)
        _data[i] = str[i];
  }
};
```

Listing 1

We can then accept `ct_str` as a template parameter for `print`, and use the same idea as before:

```
template <ct_str> struct print;
  print<"Welcome to Wordlexpr!"> _{};
```

```
error: variable 'print<ct_str{"Welcome to
Wordlexpr!", 21}> _' has
       initializer but incomplete type
   22 | print<"Welcome to Wordlexpr!"> _{};
      |
```

Now we have a way of making the compiler emit whatever we'd like as an error. In fact, we can perform string manipulation at compile-time on `ct_str` (Listing 2).

```
constexpr ct_str test()
{
  ct_str s{"Welcome to Wordlexpr!"};
  s._data[0] = 'w';
  s._data[11] = 'w';
  s._data[20] = '.';
  return s;
}

print<test()> _{};
```

Listing 2

Vittorio Romeo is a modern C++ enthusiast who loves to share his knowledge by creating video tutorials and participating in conferences. He has a BS in Computer Science from the University of Messina. He writes libraries, applications and games – check out his GitHub page. You can contact him at mail@vittorioromeo.com

Pseudo-random number generation is always deterministic, and the final result only depends on the state of the RNG and the initially provided seed

```
error: variable 'print<ct_str{"welcome to
wordlexpr.", 20}> _' has
   initializer but incomplete type
 33 | print<test()> _{};
    |               ^
```

By extending `ct_str` with functionalities such as `append`, `contains`, `replace`, etc… we end up being able to create any sort of string at compile-time and print it out as an error.

First problem solved!

Compile-time random number generation

This is really not a big deal, if we allow our users to provide a seed on the command line via preprocessor defines. Pseudo-random number generation is always deterministic, and the final result only depends on the state of the RNG and the initially provided seed.

```
g++ -std=c++20 ./wordlexpr.cpp -DSEED=123
```

It is fairly easy to port a common RNG engine such as Mersenne Twister to C++20 `constexpr`. For the purpose of Wordlexpr, the modulo operator (`%`) was enough:

```
constexpr const ct_str& get_target_word()
{
  return wordlist[SEED % wordlist_size];
}
```

Second problem solved!

Retaining state and making progress

If we allow the user to give us a seed via preprocessor defines, why not also allow the user to make progress in the same game session by telling us where they left off last time they played? Think of it as any save file system in a modern game – except that the 'save file' is a short string which is going to be passed to the compiler:

```
g++ -std=c++20 ./wordlexpr.cpp -DSEED=123
-DSTATE=DJYHULDOPALISHJRBFJNSWAEIM
```

```
error: variable 'print<ct_str{"You guessed
`crane`. Outcome: `x-xx-`.
       You guessed `white`. Outcome: `xxox-`.
       You guessed `black`. Outcome: `xoxxx`.
       You guessed `tower`. Outcome: `xxxoo`.
       To continue the game, pass
`-DSTATE=EJYHULDOPALISHJRAVDLYWAEIM`
       alongside a new guess.", 242}> _' has
initializer but incomplete
       type
 2612 |         print<make_full_str(SEED, guess,
s)> _{};
      |
^
```

The user doesn't have to come up with the state string themselves – it will be generated by Wordlexpr on every step:

The state of the game is stored in this simple `struct`:

```
struct state
{
  std::size_t _n_guesses{0};
  ct_str      _guesses[5];
};
```

All that's left to do is to define encoding and decoding functions for the state:

```
constexpr ct_str encode_state(const state& s);
constexpr state decode_state(const ct_str& str);
```

In Wordlexpr, I used a simple Caesar cipher to encode the guesses into the string without making them human-readable. It is not really necessary, but generally speaking another type of compile-time game might want to hide the current state by performing some sort of encoding.

Third problem solved!

Conclusion

I hope you enjoyed this brief explanation of how Wordlexpr works. Remember that you can play it yourself and see the entire source code on Compiler Explorer. Feel free to reach out to ask any question! ■

References

[P0732R2] C++20's P0732R2: 'Class Types in Non-Type Template Parameters', available at https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0732r2.pdf

[Wordlexpr] Play Wordlexpr on Compiler Explorer: https://gcc.godbolt.org/z/4oo3PrvqY

This article was previously published on Vittorio's website on 27 February 2022: https://vittorioromeo.info/index/blog/wordlexpr.html

ACCU 2022 Trip Reports

The ACCU conference returned in hybrid mode this year. Several writers share their experiences.

From Phil Nash

I've been attending ACCU conferences since 2001, and I don't believe I have missed one since 2009 (except for 2020, when the whole event was canceled). It was the first programming conference I started attending, and also where I launched my speaking career in 2004, speaking at every one since 2009. So ACCU is a conference very close to my heart.

I was obviously disappointed that 2020 had to be canceled – it was too short notice at the time to transform it to an online event. 2021 did run online. But that means that the 2022 event was the first in-person ACCU event since 2019! Actually about half of the attendees were still online, as it was a hybrid event. I didn't really interact with the online component, though, except for some minimal interactions on the Discord server. I heard from several people, however, that the recreation of the physical venue in Gather Town was very impressive and helped them to keep more connected to the in-person event. Kudos to Jim and Jonathan Roper, and the others at Digital Medium, for doing a great job of that while also handling the recording and streaming on the in-person side, this year. As a conference organizer myself, who has been navigating many of the same things (also in collaboration with Jim) I have an idea of just how huge a task that is – with so many possible ways to fail! Other than a handful of minor hitches, from what I saw everything went remarkably smoothly!

But what about the content? Well, personally, I didn't get to see many of the talks. On the first day, I was focused on getting ready for my own talk, at the end of the day, and a lightning talk after that. On day two I was more focused on the Sonar booth – then had to leave before the end of the day as I was off on a family holiday the next day! So I'll cop out and say that I'm familiar enough with many of the regular speakers and the types of content they were presenting that I'm 100% sure that 2022 continued to be an exceptionally high quality conference year – both for C++ developers and others. Traditionally two of the five tracks have been C++ specific, and the others have been about other languages and technologies, as well as less technical things, such as agile practices – but almost always accessible to a C++ audience, which is still very strongly represented at ACCU.

I did see the first 30 minutes of Guy Davidson's opening keynote, which had been excellent up to that point – so I will definitely be finishing it in video form later. He had just gone through an extended, and entertaining, setup for why the role of mentoring is so important, and I have to say I agree with that.

Of course, I was there for my own talk. We had a slightly delayed start due to some technical difficulties. Since the last time I have presented at an in-person event I have bought a new Macbook Pro (M1 Pro). These machines caused a bit of a stir for bringing back several non-USB-C ports, including an HDMI port – so I thought I would have less trouble with connecting to a display. In fact the main display was fine – but connecting to Jim's video capture device (or devices, he tried a couple) had a lot of issues. This year it was not just recording that was impacted – but the online attendees were relying on that stream to be able to see my slides at all! Eventually we had to give up and they pointed a camera at the projector screen and streamed that. There were some complaints that it was not quite readable in places, but I think it just about worked. For anyone that was watching that stream – sorry about that. You might want to watch one of the other versions of it I have recorded, such as the one from CPPP last December.

In the evening there was an hour of lightning talks, hosted by the shoeless Pete Goodliffe! In fact there were lightning talks every evening (except Saturday), but this was the only session I was present for, so I had requested that my submission be on that night. The ACCU lightning talks seem to have become a forum for speakers to challenge each other to do sillier, funnier or more off-beat presentations each year! This time saw Dom Davis spend most of his five minutes talking about USB connection standards – all to set up reciting an extract from Queen's *Bohemian Rhapsody*, "Thunderbolt and Lightning, very very frightening, me" (and it continues). Pete, himself, did a series of programming related visual puns. The standout, for me, though, was Andy Balaam's (pre-recorded, but otherwise) live-coding of a Snake game in Javascript, all to set up a set of puns about chasing our own tail and being consumed by ourselves. My own lightning talk wasn't really a talk at all. The exact nature is a heavily guarded secret (so far I have only done it at conferences that don't record their lightning talks – and I hope to do it at least once more before

It goes without saying that it was amazing to catch up with so many people that I hadn't seen for 3+ years (as well as meeting a few new people)

I make it fully public). All I'll say is that it relates to build dependencies and ABI stability in C++ ... but barely.

On day two I went to see the sponsored session by two of my colleagues at Sonar. After all, it would have been rude not to! In fact one of the presenters was our CEO, Olivier Gaudin – a very busy man who took an unprecedented amount of time out of his schedule to join us at the conference! The other was PM for the C++ Analyzer, Geoffray Adde. I could be biased but I think they did a fantastic job of explaining what the problem is and how it affects all of us – even if we are very careful about software quality. Of course they followed up with how our tools not only help, but have a few special features that make it much easier than you might think. It was great to see a good audience in attendance – almost a packed room – which was very impressive given the in-person attendance numbers this year.

Although these were the only sessions I saw this year, conferences like the ACCU are as much about the so-called 'hallway track' as they are about the sessions themselves. As my first in-person event with a Sonar booth, since joining last June, the booth experience was also particularly important for me. Unfortunately, the goodies that we had planned to have on our table to give away didn't turn up (an occasional bane of the conference booth business)! So all we had were some Swiss chocolate bars and a set of C++ puzzler questions I was able to print out while I was there. Foot traffic was also lower, as everyone fit into the main food areas at either end of the hallway we were in, rather than having to spill down the hallway during all the breaks! Nonetheless we had some great conversations – many with people already familiar with Sonar tools from other language ecosystems – and the quiz questions proved to be very 'sticky' with people hanging around and coming back repeatedly – determined to solve them all. In retrospect we may have made them a little too hard. By the time I left, only one of the chocolate bars (which we promised to anyone that answered all the questions correctly) had gone – and even they admitted that a few people (most or all of whom were standards committee members) had pooled together to solve the questions! Welcome to C++!

It goes without saying that it was amazing to catch up with so many people that I hadn't seen for 3+ years (as well as meeting a few new people). I also managed to spend a whole evening in the bar discussing coroutines with Nicolai Josuttis.

Which seems like a good point to do a `final_suspend`.

Phil Nash is the original author of the C++ test framework, Catch2, and composable command line parser, Clara. He is Developer Advocate at SonarSource, and a member of the ISO C++ standards committee, organiser of C++ London and C++ on Sea. He co-hosts and produces the cpp.chat and No Diagnostic Required podcasts. You can contact him at accu@philnash.me

If you've not been, then next years conference is 2023-04-19 To 2023-04-22. You really should come because it's the people who make the conference.

From Dom Davis

The three hardest things in software development are:

- Explaining what I do to my mother
- Agreeing on what the hardest things are
- Writing date/time libraries

I write this on March 770th 2020 [2022-04-10 as Russel would have written it in the old fashioned Gregorian calendar] after a whole bunch of us got together for the first time in... well we're not really sure. Time, it seems, has been extra weird in the past 750 or so days.

I am, of course, referring to *ACCUConf 2022*, a partial return to the ACCUConf we knew and loved in The Before. Carrying on from 2021's online only conference, 2022 saw a hybrid conference with some people attending in person [God, I'd missed you lot], and some attending online [I still miss you].

And wow, did I learn a lot. Including such esoterica as "I can't read the GoDocs for fmt"; spam, 419 advance fee fraud, and man in the middle attacks are hundreds, if not thousands of years old; and new ways to code my way out of a paper bag. And that was just the advertised sessions! As always, a huge amount of learning, networking and communication was going on in the 'corridor conference', but then if you're reading this I'm likely preaching to the converted. If you've not been, then next year's conference is 2023-04-19 to 2023-04-22. You really should come because it's the *people* who make the conference.

And it's that point I want to raise. We want *all sorts* at *ACCUConf*, both as attendees *and* as speakers. Fresh idea, new perspectives, bonkers personal projects, or something that you find fun or interesting. Chances are if *you* find it fun and interesting, lots of others will too. Honestly, some of my favourite talks are the ones that go off the beaten track because so often they surprise and delight.

Don't get me wrong, I still enjoy the Big Name speakers ACCU continues to attract, but I also want to see the next generation of Big Names. The future luminaries in our field. So here is your challenge: Go out there among your networks and encourage new people to submit talks, even if it's just a lightning talk – in fact *especially* if it's just a lighting talk, they're the best bit of the whole conference.

For those that are completely new to public speaking I am absolutely positive that some of us who are old hands would be happy to offer help, guidance and mentorship. I know I would. Plus we can **all** offer smiling faces and interested looks from the audience. We are, after all, a friendly, welcoming, and inclusive crowd with a diverse array of interests.

If you include lightning talks I did 4 talks this year. My ego absolutely loved it. You can be sure I'll be submitting a plethora of talks for next year [*cough* Keynote *cough*], do you *really* want to see me speak *another* four times? Actually, thinking about it, forget everything I just said. We're all good ☺.

Dom Davis is a veteran of The City and a casualty of The Financial Crisis. Not content with bringing the world to its knees, he then went off to help break the internet before winding up in Norfolk where he messes about doing development and devops. Dom has been writing code since his childhood sometime in the last millennium – he hopes some day to become good at it. You can contact him via @idomdavis or dom@domdavis.com

From Hannah Dee

ACCU was my first conference for a long time and I threw myself into the conference experience, enjoying talks on all sorts of topics. The event was a bit more dynamic in planning than many conferences because people kept pulling out with COVID (and so other people kept stepping up and offering talks to fill the gaps). This meant there were more short talks than I think had originally been planned. There were also more general talks, which suited me fine. Here are the things I'm taking away from those sessions attended:

Day 1 highlights

Guy Davidson's keynote on growing better programmers: Lots of good insight into how to be more friendly and humane in code reviews and how to mentor junior staff.

Seb Rose on behaviour driven development (BDD) and how to write good scenarios, talked about how we break down programming tasks as part of the development process, and how we communicate ideas. He introduced a handy acronym for this. BRIEF: scenarios should be Business readable, use Real data, be Intention revealing, be Essential, be Focused (and be brief). I think this applies to all systems communication to be honest – use shared language, sensible examples, right level of detail, don't waste each others time.

Charles Weir & Lucy Hunt ran an online session on different ways to discover technical security requirements. I was a bit late to this one as I went to the wrong online system, but the general idea was to investigate a couple of different card games for information security [Berkeley, Microsoft, Washington]. I might have to pick up some of the games and look into this for teaching.

Day 2 highlights

Jutta Eckstein looked at how the Agile principles of development can be sustainable. This was workshop where we considered of each of the agile principles with regard to the *triple bottom line* of sustainable development: environmental, social and economic sustainability. Some

the tension between information security professionals (and security policies) and developers – often security is seen as a bolt-on by developers, and fundamental by security professionals

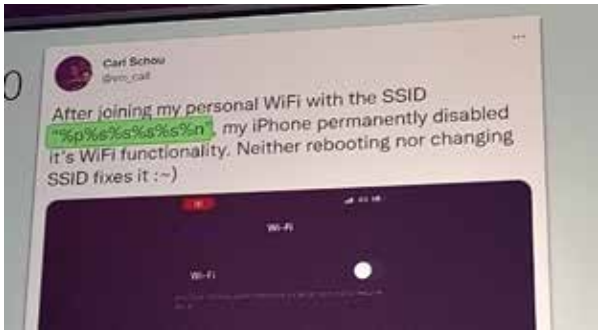
of the agile principles (e.g. "simplicity: maximising the amount of work not done") fit really well with sustainable goals. Others not so much.

Dom Davies talked about remote vs distributed working based upon many years in distributed teams – starting with global teams in the late 90s. He suggested that the key is to deal with everyone as if they're distributed, even if you're sitting next to each other. This makes a lot of sense to me. The hybrid experience is a strange one.

Matthew Dodkins talked about designing systems that would run for a long time without maintenance – specifically, bat and dolphin detectors which could run for a year in a rainforest or underwater. This talk covered a lot of detail about planning, testing, and concepts like sentinel functions (things which spot when something stops happening). Always think about **what happens next**.

Day 3 highlights

Patricia Aas's keynote was one of my favourite sessions of the conference. She looked at some "classic" vulnerabilities (heap manipulation, format string vulnerabilities etc.) and showed how they related to modern security issues. This is a talk I will watch again, and that I will heavily recommend to my first year infosec students.



Next up was a talk after my own heart: Andy Balaam spoke about "vim for fun". I have been a user of vim for about 25 years now (whoops) so have a fairly good understanding of how it works, however, it's always good to visit a session where you know. This time I picked up new movement commands: } and { to go forwards or backward to the next empty line.

Kate Gregory's talk was another strong contender for favourite presentation. It was about *abstraction*, which is a pretty big topic. Increasing abstraction localises complexity, which reduces the cognitive load; quite often, you can work out abstractions from the code without actually understanding the domain much at all.

Hannah's contribution was originally published on her blog on 27 April 2022 and is available from: https://www.hannahdee.eu/blog/?p=1820

Useful rules of thumb:

- **magic numbers->** named constant gives type and semantics
- **groups of variables ->** struct or class

Variables can be grouped by similar names (**`empdate`**, **`empname`**, **`empfirstname`**… are we looking at an employee class here?) or by '*load bearing white space*'. I love the concept of load bearing white space. So often we stick extra lines in code to break stuff up visually – but not conceptually. why not make that break explicit and part of the abstraction?

It was really interesting to see a talk which looked at this from the perspective of code, rather than problem analysis – Kate described being brought in as a consultant to fix legacy systems with tens of thousands of lines, and thinking about how we can abstract from code to tidier code (rather than from a problem to code) was very interesting. One on my 'will watch again' list.

Day 4 highlights

Gail Ollis and Ian Reid spoke on the tension between information security professionals (and security policies) and developers – often security is seen as a bolt-on by developers, and fundamental by security professionals. This is going to be a difficult circle to square, but they've been doing some interesting work around interviews with both communities. A good analogy came up – infosec professionals are like goalies: their aim is a clean sheet; developers are like strikers: their aim is to score goals. Success in one case is defined by absence of failure, which is going to lead to different risk-taking behaviour.

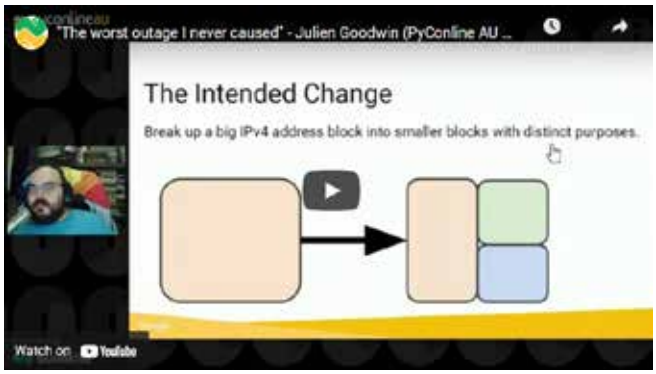
Titus Winters delivered the final keynote of the event on how we measure the cost of tradeoffs in the software engineering workflow. How to you measure the cost of a mistake or the value of preventing a defect? The earlier you detect, the lower the cost in terms of time (developers etc.). Titus is dealing with very large systems and teams, with static analysis, IDE, code review, CI, fuzzing, canary releases etc. etc. so the ability to manage and measure this stuff is something he's got some very interesting thoughts on. Particularly liked:

> It's programming if 'clever' is a compliment, it's software engineering if 'clever' is a criticism.

Other intriguing things and references from the conference

- This website: https://www.vimgolf.com/
- A paper from Google about gender/age/race effects in code review: https://cacm.acm.org/magazines/2022/3/258909-the-pushback-effects-of-race-ethnicity-gender-and-age-in-code-review/fulltext

- This video from pyconline AU, available on YouTube: https://www.youtube.com/watch?v=AUTsDTVtfFE&t=2s



References

[Berkeley] 'Adversary Personas', available at https://daylight.berkeley.edu/adversary-personas/

[Microsoft] 'Elevation of Privilege', available at https://www.microsoft.com/en-gb/download/details.aspx?id=20303

[Washington] 'Security Cards', available at https://securitycards.cs.washington.edu/

Hannah Dee is a lecturer in computing at Aberystwyth University. She is interested in computer vision, robotics, information security and data science. You can contact her via her blog at hannahdee.eu

From Timur Doumler

From April 6-9, 2022, I attended ACCU 2022, the 25th edition of the ACCU conference, in Bristol, UK. It was only my third in-person conference post-COVID (after CppCon'21 and ADC'21). It felt very special to be back in Bristol after a three-year break, and it felt as good as ever to be among real people again. With all COVID restrictions now lifted in the UK, this was the first event on the C++ conference circuit that truly felt like it was back to normal. There was no social distancing or any other measures to impact the conference experience, and the vast majority of attendees were not wearing masks.

For me, ACCU has a special place in the C++ calendar for many reasons. First, ACCU is known for its exceptionally friendly and nice community, including folks who have been attending ACCU regularly for a decade or more, but also being especially welcoming and approachable to first-timers. This year was no different – it was great to be back meeting old friends and making new ones.

ACCU is also special in that it is not only about C++ (although it has a large proportion of C++ content), so you also get content on other technologies and non-technical talks about soft skills and other interesting things. The quality of the talks is very high and the size is just right: bigger than 'small' conferences like *C++Now*, but not as big as *CppCon* or *MeetingC++*, so you still get the chance to meet and talk to most attendees.

In pre-pandemic times, I would usually expect about 400 people at ACCU. This year, however, we only had about 150 people on site, so it was noticeably quieter than usual (although the vibe was as positive as ever). In addition, there were about as many people attending online, as the conference was fully hybrid this time, with three on-site and two online tracks and the option for online attendees to watch the on-site talks and ask questions.

The organizers went to great lengths to make the online component of the conference really enjoyable, including recreating the whole conference venue (Bristol Marriott) online inside gather.town. But I have to admit that I did not engage with the online component. Why spend time in video chat rooms when you are on site at the venue and have real people to interact with? Therefore, the following trip report is focusing exclusively on the on-site component of the conference.

Not only were there fewer people on-site, but also fewer booths. I spotted the traditionally present Bloomberg booth, as well as Graphcore and SonarSource booths. And we had a #include table again, which was a very popular hangout spot. Sadly, JetBrains did not have a booth this time, as the rest of my team was unfortunately unable to attend. In fact, out of all of the ACCU conferences I've attended, this was the first one with no JetBrains booth! One of my most memorable conference experiences was walking up to the JetBrains booth at ACCU 2017 and asking them whether they would consider hiring me, and the following year I was working behind the very same booth. I very much hope we will be back there in full strength next year, but this time I was representing JetBrains on my own.

Conference day 1 started with Guy Davidson's keynote 'Growing Better Programmers'. Guy is a good friend of mine and this was his first proper conference keynote. I have to say he did an amazing job! Speaking from many years of experience, Guy shared many great thoughts about mentoring and supporting programmers, promoting good practices, conducting helpful code reviews, being a good manager, and much more.

I missed the talks following the keynote because I was busy practising my own talk, how to implement a lock-free atomic **`shared_ptr`**, which I presented that afternoon. Following that, I was met with the usual dilemma at such conferences: there are multiple tracks and you can only go to one. The program is so high-quality that you end up missing a lot of fantastic talks you really want to see. My talk of choice was 'Zen and the Art of Code Lifecycle Maintenance', a very deep and insightful investigation of what we actually mean by 'code quality'. The talk was given by Phil Nash, my predecessor as C++ Developer Advocate at JetBrains, who now works at SonarSource.

My first day at ACCU 2022 wrapped up with a lightning talk session moderated by Pete Goodliffe. This is the true highlight of the ACCU conference: the lightning talks are usually very entertaining, creative, funny, thoughtful, and mostly not about programming at all. In between the talks, Pete tells dad jokes, and if any speaker overshoots their 5 minute speaking limit, Pete takes away the microphone mid-sentence and pushes them off the stage. It's hilarious!

Day 2 opened with a keynote by Hannah Dee, `Diversity Is Too Important to Be Left to Women`. I was really impressed by this talk. It covered why diversity and inclusion are important, what we should do, and what we should not do, and covered it all in a way that felt extremely well-researched and backed up by data and facts. Even though this issue was already very close to my heart before Hannah's talk, I learned about so many new things from her, like the concepts of gender role spillover and stereotype threat, how self-efficacy affects performance, the Petrie multiplier (a very simple yet powerful mathematical model), pareidolia, and more. I highly recommend this talk as a solid, approachable, and enjoyable introduction to anyone interested in the topic of diversity in tech.

Switching gears after the keynote and diving into C++ code again, I went to 'C++20 – My Favourite Code Examples' by Nico Josuttis. This talk was full of interesting C++20 code. Nico started by showing various ways to use C++20 concepts and constraints in practice. Among other things, I learned that we can use a **`requires`**-clause inside an if **`constexpr`**, like this:

```
void add(auto& coll, const auto& val)
{
 if constexpr (requires { coll.push_back(val); })
   coll.push_back(val);
 else
   coll.insert(val);
}
```

This is a very convenient way to branch on the existence of a member function, something that was a lot more cumbersome pre-C++20. Nico then showed practical examples of how to use ranges, views, the spaceship operator, and other C++20 features all working together. Along the way, I also learned how to implement your own range sentinel! Of course, it would not be a Nico Josuttis talk without a healthy dose of ranting about things that the C++ standard committee has gotten wrong in Nico's opinion: how cbegin is utterly broken, how 'forwarding reference' is a bad name, how **`std::views::elements<0>`** does not work for user-defined types unless you declare your own tuple customization points *before* you **`#include <ranges>`** (which is quite unfortunate indeed), and so on.

ACCU is one of the conferences that offer talk slots to sponsors, but often these can be very interesting, so I went to a sponsored session during the lunch break: 'The Power of Clean Code' by the SonarSource folks. I was very happy to see CLion in action there, supplemented by their SonarLint plug-in, which looks like a very powerful tool that I definitely need to check out in more detail.

After lunch, another ACCU speciality was waiting. The normal talk length is 90 minutes, but they also have 20-minute 'quick talks', usually several of them back to back. The ones I chose were very interesting and relevant explorations of 'soft' topics: a talk by Björn Fahller about burnout, a talk by Dom Davis about remote working, and a talk by Joe Pascoe about how to be a good manager.

The second day finished with another round of lightning talks, followed by the C++ Pub Quiz hosted by conference chair Felix Petriconi.

Day 3 started with a keynote by Patricia Aas about software vulnerabilities, and this was another absolute highlight of the conference. Patricia took us on a fascinating time travel journey through the last two decades. Her slides were packed with interesting code examples demonstrating vulnerabilities arising from malloc, use after free, heap buffer overflow, integer operations, and printf format strings. Among other things, I learned what a 'Write-What-Where primitive' is and that format strings are an almost Turing-complete programming language! Patricia's talk ended with the thought that we would all benefit from more cross-pollination between the systems programming community, which C++ is part of, and the binary exploitation/vulnerability community.

The next talk slot had no fewer than three coroutines-related talks scheduled against each other. I decided to listen to Björn Fahller's talk about using coroutines for asynchronous I/O on Linux and how they really help to avoid callback hell. In C++20, we got a basic low-level API with hooks into the compiler that enable the use of coroutines, but no library facilities whatsoever on top of that to actually use them. So even for the most basic coroutine example, we need to write a lot of boilerplate code by hand – the coroutine type, the promise type, and so on. This is very hard to learn and to teach, but Björn did a good job guiding the audience through the required machinery. His talk also contained references to other helpful introductory talks about C++20 coroutines, such as Pavel Novikov's 'Understanding Coroutines by Example'.

The other highlights of Day 3 for me were Kate Gregory's 'Abstraction patterns' and another talk about coroutines, this time by Andreas Weis, showing how to use them for data processing pipelines (reading a file from the disk, compressing it, and so on). With my background in audio software development, I was fascinated by how Andreas' data pipelines were structurally virtually identical to audio processing graphs, containing sources, sinks, filters, and buffers, with data channels exchanged in between them. This makes me think (again!) that coroutines would probably be very useful in that domain, too.

In the evening, there was a third lightning talk session hosted by Pete (they were so popular this year that Pete ran out of lightning talk slots!), which was just as much fun as the others, and finally the traditional conference dinner, which I always enjoy. The ACCU conference dinner is geared towards enabling as many conference attendees as possible to mingle with speakers. Speakers remain at the same table throughout the dinner, while everyone else changes tables after every course. This is good fun and you get to meet lots of new people. In addition, there is always a theme – this year it was film.

And now we're at the last day of ACCU! I started the morning by attending Mathieu Ropert's 'Basics of profiling'. Mathieu delivered a really good beginner-friendly introduction to profiling, why it's important, things you need to know such as sampling profiling vs. instrumentation profiling, and more. He then actually live-demoed profiling his video game Hearts of Iron IV with the Optick Profiler, a instrumentation profiler, and Intel VTune, a very powerful sampling profiler. He explained what to look for and how to interpret what you see, and finished off his talk with useful general tips on how to optimize a C++ program. This hands-on approach including live demoing is rarely seen in conference talks, and I really enjoyed it.

Later that day I attended John McFarlane's talk 'Contractual Disappointment in C++', which I also highly recommend. The title would suggest that the talk was about contracts in standard C++ and how we ended up not having them, but actually it was not about that at all. Instead, John talked about contracts as a programming concept, the difference between bugs and errors, different types of contracts, and other things that every programmer should think about.

Finally, it was time for the last talk of the conference: Titus Winters' closing keynote! Titus, the author of *Software engineering at Google*, aka the Flamingo Book, decided to fully embrace his new brand and showed up on stage in a flamingo shirt. His keynote titled 'Tradeoffs in the Software Workflow' was just as high-quality and thought-provoking as I had hoped and tackled really big questions for our industry such as 'What is the value of the code that you write?' and 'What is the value of preventing a defect?' This is another talk that I highly recommend. In fact, all the talks I saw this year were so good that I am very impressed once again by the quality of the ACCU conference program.

Of course, in between all the talks, there was the all-important hallway track. Even though there was no JetBrains booth this time, I was still very busy talking to people about JetBrains products in general and CLion in particular. Some noteworthy interactions involved helping a CLion user figure out how to build the LLVM project in CLion (which gave me some good ideas for putting together a 'CMake in CLion' tutorial) and multiple folks asking about the new 'thin client' remote development mode in CLion (which is currently still in Beta, but nevertheless already very usable, so I recommend you check it out now!).

And that was it from ACCU 2022! I am really looking forward to next year's conference, but in the meantime, I will be attending other in-person C++ events, with the next one on the calendar being C++Now in beautiful Aspen, Colorado. See you there! ■

Timur Doumler is C++ Developer Advocate at JetBrains and an active member of the ISO C++ standard committee. As a developer, he worked for many years in the audio and music technology industry and co-founded the music tech startup, Cradle. Timur is passionate about building inclusive communities, clean code, good tools, low latency, and the evolution of the C++ language. You can contact him at timur.doumler@jetbrains.com

The contribution from Timur was originally published on the JetBrains CLion blog on 21 April 2022, and can be accessed here: https://blog.jetbrains.com/clion/2022/04/accu-2022-trip-report/

Afterwood

Threads can mean many things. Chris Oldwood pulls a few to see what happens.

I was recently watching an episode of Marvel's Agents of S.H.I.E.L.D with my youngest before his bedtime. In the episode they had travelled back in time to 1940s America and were changing into clothes considered more appropriate for the times, to blend in. After they all got changed and met back up again one character remarked to the other, "Nice threads!" In this instance they were referring to another person's clothes, but it got me thinking that nobody has ever said that to me, which, given my weak fashion skills is not surprising, but more topically for this publication, it's not a phrase I've ever heard somebody remark about another person's code either. Given the problems that all too often arise from the introduction of additional threads into a program it's far more likely that you'll be chastised for your threads rather than commended for them.

The late Russel Winder was a programmer who was no stranger to problems involving concurrency and had the good fortune to work with some serious parallel hardware when some of us were still all gooey eyed over the second CPU in our desktop machine. I'm pretty sure Russel would never congratulate anyone on their choice of threads as he was a big proponent of solving concurrency problems by using higher-level concepts. Like so much in the world of Computer Science many of the techniques for managing concurrency have been around for decades and Russel was always keen to promote the Actor Model, Communicating Sequential Processes (CSP), etc. in his talks and writings. I never really grokked either of these until Russel published his Introduction to GPars in *CVu* 22(6) back in 2011. I've still never written a line of Groovy or done anything significant on the JVM but this article provided the clarity I needed to start seeing how these ideas were realized in a modern language.

If you grew up in the UK during the 1980s you might already have an aversion to threads due to the BBC's film of the same name which depicted the state of Britain after a nuclear war. I was slightly too young to watch it first time around, although like any teenager that didn't stop me trying to because apparently there were 'other kids' in our school who had allegedly watched it. Despite the grown-ups slapping a 15 certificate on it to advise us youngsters against being foolish and dabbling in issues we were emotionally under-equipped to deal with, we jumped right in anyway and regretted it later. Why do we never listen to our elders? As the old joke goes, "*Some programmers, when confronted with a problem, think 'I'll use threads', now problems two have they.*"

I once interviewed someone for a highly technical role and casually asked them how they felt about lock-free programming. They simply replied, "*I'll give anything a go*." While I admire their positive outlook on life, this was not really the response I was expecting. Anthony Williams' book on concurrency with C++ weighs in at six hundred pages and Joe Duffy's concurrency book for Windows hits nine hundred. What this tells me is that it isn't something you can 'dabble in'. It feels more like a career in its own right.

Debugging multi-threaded programs is always an interesting prospect, especially trying to single step through functions which are executing similar workloads on different threads. It reminds me of my first foray into the Usenet way back at university before the common availability of 'threaded newsreaders'. Contrary to what you might be thinking, these weren't programs which used multiple threads to achieve better UI responsiveness (we're talking Unix terminals here), this was about stitching together a continuous stream of forum posts on different topics so that you could focus on one conversation at a time instead of constantly context switching between subjects. Single stepping through a multi-threaded program is always a bit of a shot in the dark as you wonder how far you'll get before you're whisked away to another land. At least we eventually get to return to where we left off unlike poor old Sam Beckett in *Quantum Leap*. "*Oh, boy!*" indeed.

It wasn't just late 80s newsreaders though – this was also how Twitter felt in its early years. Fortunately, everything was largely unrelated anyway so there wasn't really a context to be dragged away from and return to. The introduction of the 280 character limit in late 2017 was swiftly followed by the appearance of 'threads' as Twitter tried to convince its users that brevity was no longer the soul of wit. Further correlation between social media platforms and concurrent programming are possible when you consider their problems with coherence and false sharing.

One career that seems to have died out since the C++ committee finally decided to come clean with C++11 and define a thread-aware memory model is that of answering Stack Overflow questions about how to safely implement a Double-Checked Lock (DCL). The Internet was awash with solutions on how to safely acquire a Singleton (though, once again, you now have two problems) that turned out to be wrong. The dominance at the time of the Intel CPU meant that 'works on my machine' was almost a statistically valid argument. A few people working on more advanced CPU architectures got bitten but the strong view taken by the incumbent x86 meant that many of us lived in ignorant bliss. When Herb Sutter declared to the world that the free lunch was over, he was talking about CPU single core performance, but he might as well have been talking about the rise of the ARM which has a weaker view on ordering and a stronger view on messing with your head. Java had its DCL crisis just after Y2K calmed down whereas .Net had another decade to go before its bubble finally burst, even the JVM & CLR are not immune it seems.

With the continued working from home and decline of the suit and tie in the workplace, I suspect my chances for a fashion compliment have long since passed. As for the prospect of never having to deal with a threading issue again, all I can say is, "*Promises, promises!*" ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He has resumed commentating on the Godmanchester duck race but continues to be easily distracted by messages to gort@cix.co.uk or @chrisoldwood

To connect with like-minded people visit accu.org

accu