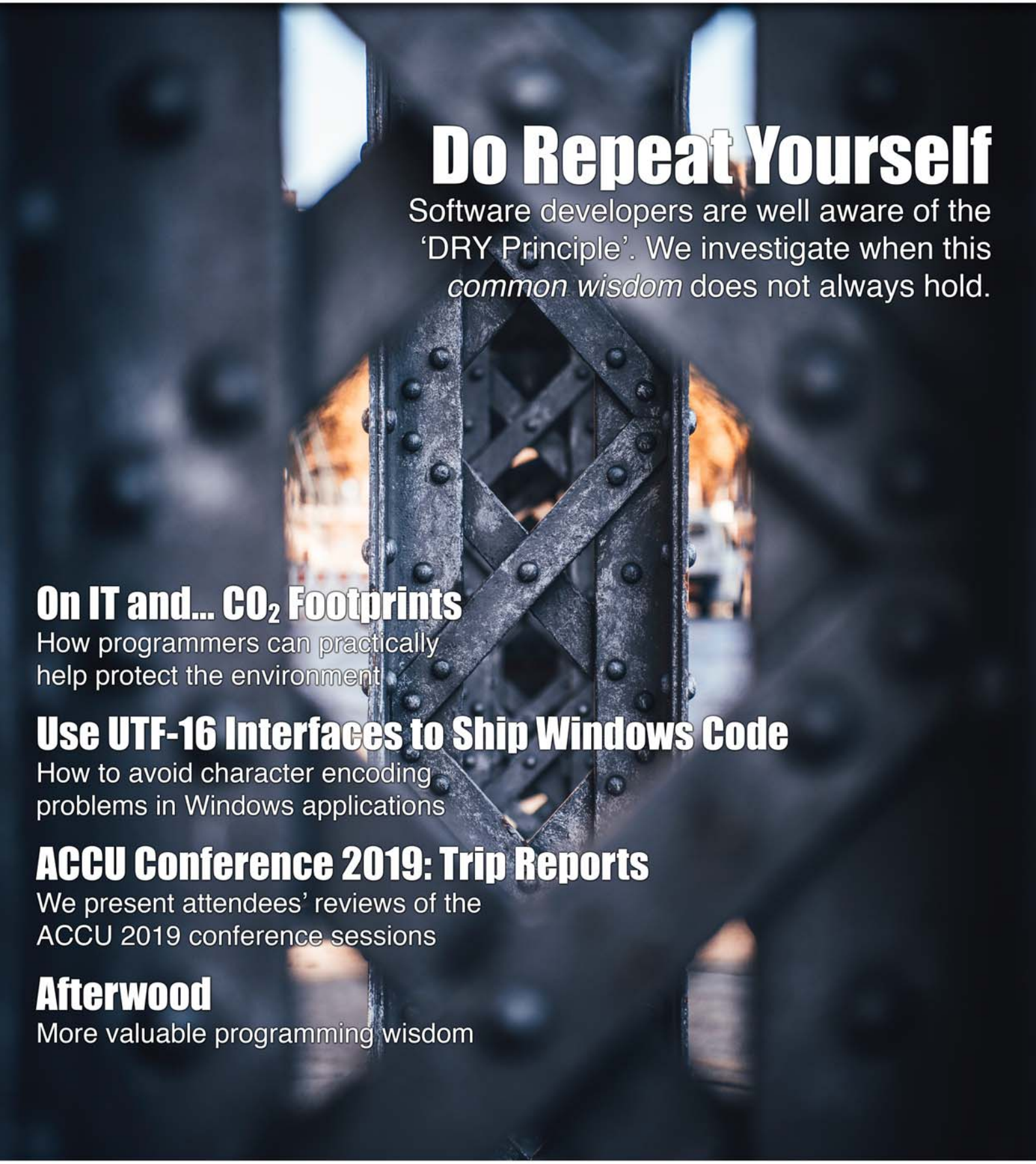
overload 151

JUNE 2019 £4.50



# Do Repeat Yourself

Software developers are well aware of the 'DRY Principle'. We investigate when this *common wisdom* does not always hold.

## On IT and... $CO_2$ Footprints

How programmers can practically help protect the environment

## Use UTF-16 Interfaces to Ship Windows Code

How to avoid character encoding problems in Windows applications

## ACCU Conference 2019: Trip Reports

We present attendees' reviews of the ACCU 2019 conference sessions

## Afterwood

More valuable programming wisdom

A magazine of ACCU

ISSN: 1354-3172

**OVERLOAD 151**

**June 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 152 should be submitted by 1st July 2019 and those for Overload 153 by 1st September 2019.

**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**

# How Hard Can It Be?

Getting different parties to collaborate might sound easy. Frances Buontempo explores where problems and opportunities arise.

We've been considering a house move recently, so I've been drowning in emails and forms and waiting for the solicitors to keep us up to date with what's going on. This has distracted me from writing an editorial, as you can imagine. They are sternly refusing to provide any estimates on how long this might take or what they are waiting on. I mean, how hard can it be? OK, to be fair, when we ask a specific question we do get a specific answer, sometimes to a different question. Sometimes, they have even given approximate timelines, saying something like "Searches usually take two weeks." The estate agents have a different expectation of timelines, warning us this can take much longer. All very confusing and stressful. I'm surprised anyone ever manages to buy a house in the UK. So, as ever, *Overload* is without an editorial. This did get me thinking though. Managing a project involving programmers or solicitors is like herding cats [Urban dictionary09], as the saying goes.

I can see some parallels between people with different perspectives or motivation trying to collaborate on a project, be that a house-move, writing software, or even a magazine or a book. I find it hard to give estimates for software projects. It is hard. And I don't always go deep into gory tech details if someone asks what I'm working on at the moment. I try to adapt my response to the person asking. You can talk about APIs to some people or completed features to others. If you're a team of people working together, it's even harder to come to an agreement and provide a clear answer to a straight question. Larger teams can make this even harder. I can't imagine how Kevlin Henney managed to coordinate the *97 Things Every Programmer Should Know* book [Henney10] – lots of collaboration!

Instead of talking about people, how do we get code to collaborate? We know collaborate means working together, so can we get different parts code to work together? How hard can it be? It depends. Remote procedure calls (RPCs) give one way to get code to work together. I haven't used this for a while, but have done my fair share of DCOM and similar. You need to know a lot about the internal workings of the code you call, and can make all sorts of mistakes. But it can work. Another in-tandem approach, without getting as complicated as remote calls, involves trying to get two different languages to work together. You can build several languages into one product using something like Simplified Wrapper and Interface Generator [SWIG] or other bindings, or manually prototype function calls, making sure you find the equivalent of various types, or use a *lingua franca*, often C. You still need to be very careful about the size and endianness of numbers. Strings are another story. Some shared language and careful testing of assumptions is always required.

Alternatively, you can have several 'products' or programs, and get them to talk via messages rather than function calls. That seems simpler. In many ways, it is. You do need to spend time agreeing the syntax and semantics messages. How might you deal with different versions of each component? If you need to send new parameter, how do you approach this? I'm getting ahead of myself, though. You can also start thinking about network programming in general and various protocols. Would you like to hear a UDP joke? I would tell you, but you probably won't get it [Hacker News]. Anyway, one program, in several languages solves some problems. Two or more programs, in the same language, calling each other's functions is suitable for other situations. Two or more programs communicating via messages give some flexibility, but also needs some thought. Yet another way to get things working together is one program, with several threads. You may have events and delegates, or locks around shared data to orchestrate this. If you are not careful, you can get into trouble. Many interviews immediately ask what a deadlock is when multi-threaded code gets mentioned. I wonder if our solicitors have deadlocked themselves somehow.

Back to collaboration. Orchestration "is the automated configuration, coordination, and management of computer systems and software" according to Wikipedia [Wikipedia-1]. This sounds somewhat like an automated attempt at herding cats. An orchestra plays many instruments together, possibly led by a conductor. The individual players can practise alone, or in smaller groups, but make beautiful music when they come together. This needs some background work and shared language, or at least notation. People can manage to dance together without being coordinated by herds of overseers. If a primary school teacher plays some music, the young people will manage to dance in their own way. They can be individually expressive while moving to the same beat. The idea of timing, through a rhythm or a drum beat is a recurring theme. A rowing team will be guided by a coxswain, in essence counting. It's the London marathon this Sunday. Some people may practise by trying to keep track of their pace. I suspect there's a similarity with the pace at which a team works, in order to get a larger project to succeed. Or even some multi-threaded code. How many problems are 'fixed' by changing the lengths of sleeps, or other timeouts? Can two people work together if they don't speak the same language? I dimly recall a program on the television a while ago, exploring how babies through to toddlers learn. One child, who couldn't yet talk, was encouraged to learn to beat out a drum beat on a table top. He interacted with someone talking to him by responding with various drum beats. Language has a rhythm, and he seemed to instinctively grok that. Little in-roads into getting nearer TCP communication, wherein you get some kind of feedback indicating the message has been received and understood makes some kind of difference.

**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.

Let's expand this out and talk about learning in a programming context. How can you tell if a mentee or apprentice is learning? Chris Oldwood [Oldwood18] suggested counting how many of his jokes the mentee laughed at. Seriously? Damn it. Janet. See what I did there? People who have seen *The Rocky Horror Picture Show*, tend to immediately say 'Janet' after the phrase 'Damn it'. Sometimes shared 'secret' knowledge comes through. This is almost like a beat or tune or, dare I say it, a joke. You are building a common language. If I say 'Knock, knock' many people know to respond with 'Who's there?' Is this really communicating? I suspect it would be relatively straightforward to get a chat bot to at least start trying to join in with certain jokes. This puts me in mind of an old children's game. Has anyone come across *Consequences* before [Wikipedia-2]? You concertina a piece of paper, and fill out words answering questions one each folder, and refolding so you don't see what came before by filling in blanks in a story:

> Once upon a time, a [name an animal]
>
> went to [name a place]…

and so on.

The stories are then read out and hilarity ensues. A variant involves drawing a person, head, neck, torso, legs, and feet. Then you can just laugh at the pictures afterwards, and not waste time reading a story. Does your software project have consequences? Or at least get developed like *Consequences*? Do you write code that needs to interface or integrate with other teams' code? You have probably heard of the Fizz Buzz

> Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz". [Wiki]

Have you heard of 'Evil Fizz Buzz' [codemanship]? This is team work Fizz Buzz, but

1. Split the team into groups.
2. Assign each part of FizzBuzz above to a pair. They can only work on code for that part of the whole.
3. Task them to work together – but only coding/TDD-ing their individual parts – to deliver a complete solution that produces the desired output.
4. Give them about an hour. And stand back and enjoy the train wreck.

Try it one day. Figuring out how to get groups of people and software to work together can be challenging.

I mentioned chat bots earlier. Let's talk about the rise of the machines. How can AI and humans collaborate? At the speakers' dinner at the ACCU conference this year, Echoborg [Echoborg] entertained us. The website proclaims, 'A funny and thought-provoking show that is created afresh each time by the audience in conversation with an artificial intelligence.' An actor voices the words of a chat bot. Members of the audience go up and chat to the actor. Speech recognition software sends the words to the chat bot, and the actor speaks back. You had to be there. What happens if two Echoborgs chat? Apparently this happened by mistake once, and was turned off forthwith. Could they collaborate and take over the world? What would they do? A while ago some sensationalist headline hit the world: 'Facebook shuts down chat bots after they invent their own language.' The theory is that the bots were tasked with trying to negotiate. The New Scientist said:

> One bot was taught to mimic the way people negotiated in English, but it turned out to be a weak negotiator, and too willing to agree to unfavourable terms. A second was tasked with maximising its score. This bot was a much better negotiator but ended up using a nonsensical language impossible for humans to understand. [Reynolds17].

In some ways, negotiation is a form of collaboration, and many industries to end up inventing their own language. Programmers, musicians and solicitors. Just saying!

Asking how to get humans and AI to collaborate possibly has a history that pre-dates computers. Elements of Frankenstein pull on the idea of what happens if people create a 'living' autonomous intelligence. That doesn't end well. I've seen a variety of tales about similar goings on. For example, the adventures of a robot trekking across the US, called HitchBOT. One British newspaper, who shall not be named, said "A friendly robot who was hitchhiking its way across America as part of a scientific experiment has been heartlessly attacked and beheaded just 300 miles into its journey." Informative journalism at its height, not. Wait! Friendly? What would an unfriendly HitchBOT be like? A mash-up of Rutger Hauer in *The Hitcher* [IMDb-1] and Rutger Hauer in *Blade Runner* [IMDb-2]?

OK, enough scary dystopian imaginings. How can you have a human in the loop? Many automatic translation programs, in some ways a type of AI to my mind, have relied on humans giving feedback, including better translations. If you have a way to give feedback to a running algorithm (think AI) then you can collaborate. I gave a workshop with Chris Simons at this year's conference, where we showed how Evolutionary Programming can generate the code for Fizz Buzz. This begs the questions, should you? The code generated was disgusting. But if we then gave feedback, and nudged our fitness functions (think tests) a bit, we might end up with improved code. Maybe code readability doesn't matter if the machine writes its own code and it does what's required. No human need ever look at the implementation. People who want code had better be good at writing tests though. Or maybe we do need to learn to collaborate with the machines. 'Human in the loop' learning is a trending topic in AI research currently. Watch this space.

> OK, so fill in a first sentence:
>
> [..]. How hard can it be?
>
> Go.

Yep, OK, getting the number of dots in an ellipsis correct [Buontempo19]. Collaboration is difficult, frustrating, frequently involves making up languages and communication protocols but can be fun. It needn't be a collusion or conspiracy, though AI, software engineers and the legal profession can be regarded that way. Teaming up to produce something magical can be amazing. How hard can it be?

## References


[Buontempo19] Frances Buontempo (2019) on Twitter, tweeted 27 April 2019: https://twitter.com/fbuontempo/status/1122074050946318341

[codemanship] 'Evil FizzBuzz' (or 'So you think you're a team?'), 2017, http://codemanship.co.uk/parlezuml/blog/?postid=1494

[Echoborg] http://echoborg.com/

[Hacker News] https://news.ycombinator.com/item?id=8466276

[Henney10] Kevlin Henney (2010) *97 Things Every Programmer Should Know*, published by O'Reilly https://www.oreilly.com/library/view/97-things-every/9780596809515/

[IMDb-1] *The Hitcher* (1986) https://www.imdb.com/title/tt0091209/

[IMDb-2] *Blade Runner* (1982) https://www.imdb.com/title/tt0083658/

[Oldwood18] Chris Oldwood (2018) 'Are we nearly there yet?' *Overload* 147, Oct 2018, https://accu.org/index.php/journals/2566

[Reynolds17] Matt Reynolds (2017) 'Chatbots learn how to negotiate and drive a hard bargain', *New Scientist*, posted 14 June 2017 at https://www.newscientist.com/article/mg23431304-300-chatbots-learn-how-to-drive-a-hard-bargain/

[SWIG] http://www.swig.org/

[Urbandictionary09] 'Herding cats' (definition) at https://www.urbandictionary.com/define.php?term=herding%20cats

[Wiki] http://wiki.c2.com/?FizzBuzzTest

[Wikipedia-1] https://en.wikipedia.org/wiki/Orchestration_(computing)

[Wikipedia-2] https://en.wikipedia.org/wiki/Consequences_(game)

# Do Repeat Yourself

Software developers are well aware of the 'DRY Principle'. Lucian Radu Teodorescu investigates when this common wisdom does not always hold.

If you are a software developer, chances are that you heard about the DRY principle: "Don't repeat yourself" [Hunt99]. Actually, chances are that you've heard it multiple times; probably many, many times. If you do a quick Internet search, you see that this phrase is repeated *ad nauseam*. But how come a mantra that preaches no repetition is repeated – ironically – so many times? Starting from this paradox, this article analyses why sometimes repetition is vital for people and also useful for software development.

## The name of the game

Software development is a knowledge acquisition process [Henney19]. It's not enough to write code for machines to understand; we need also people to be able to understand it and reason about it. It's mostly a social activity. It's not enough for the actual co-workers to understand your code, future co-workers also need to understand the code. Furthermore, if you understand your code now, you may not be able to do it 6 months in the future – that's how volatile is the understanding of the code.

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand*
> ~ Martin Fowler

The main bottleneck of software development is the understanding capacity of programmers. If, following Kevlin Henney, we rename the term code into codified knowledge [Henney19], then the fundamental problem is arranging this knowledge in such a way that it allows easy acquisition by humans and easy reasoning on it.

There are many aspects of organizing this knowledge, but for the purpose of this article, we are concerned only about the use of repetition.

## Other forms of knowledge representations

Let us take verbal communication as the primary form of interacting with knowledge. First, there is the actual verbal communication, then there is the non-verbal one. The non-verbal communication often repeats the verbal communication; it's used most of the time to strengthen the message expressed through words.

Looking at the language itself, we find that it's highly redundant. Some very common examples of redundancy in English include: plural and gender concordance, the third person singular -s, subject-predicate inversion (in the presence of an interrogative word), etc. It seems that humans are better equipped to understand messages with a lot of redundancy. If people find that processing natural language is easier in the presence of redundancy, why would we want to remove redundancy from the software that people are supposed to read?

**Lucian Radu Teodorescu** has a PhD in programming languages and is a Software Architect at Garmin. In his spare time, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

Let's go further in our analysis of repetition in discourse. Within rhetoric, repetition is an important strategy for producing emphasis, clarity, amplification or emotional effect. It can be of letters/syllables/sounds, words, clauses or ideas. For example, one can see a lot of repetition in the following speech:

> We shall not flag or fail. We shall go on to the end. We shall fight in France, we shall fight on the seas and oceans, we shall fight with growing confidence and growing strength in the air, we shall defend our island, whatever the cost may be, we shall fight on the beaches, we shall fight on the landing grounds, we shall fight in the fields and in the streets, we shall fight in the hills. We shall never surrender.
> ~ Winston Churchill

One can say that the text is just a big repetition of the same idea. How would a DRY fan 'refactor' this text? Probably something like the following:

> There are a few things we shall do: go on to the end, fight – in France, on the seas, in the air (with growing confidence and growing strength), beaches, landing grounds, fields, streets, hills – go to the end, defend our island (whatever the cost may be); but never surrender, flag or fail.
> ~ DRY enthusiast

And, because refactoring is an iterative process, after a few shake-ups of the text, we would arrive at:

> We shall fight and never surrender. ~ DRY enthusiast

For fun, and for the sake of repetition, let's have 3 more examples:

> O Romeo, Romeo! Wherefore art thou Romeo?
> ~ William Shakespeare

Becomes:

> Hey, Romeo! Wherefore art thou?

And:

> I am so happy. I got love, I got work, I got money, friends, and time. And you alive and be home soon. ~ Alice Walker

Becomes:

> I've got happiness, love, work, money, friends, time, you alive; reaching home soon.

And finally:

> Happy families are all alike; every unhappy family is unhappy in its own way. ~ Leo Tolstoy

Becomes:

> Happy families are all alike; the others not.

And, because the last quote was from Anna Karenina, I would like to stress one more point. How would an author construct such a complex novel if it didn't have repetition? How can one construct characters without repeating types of behaviors? How can one distinguish between main characters and other characters without repeating the names of the main characters more? Imagine every name in Anna Karenina written only once, every detail about the world that Tolstoy created appearing only once.

Ignoring its aesthetic value, we can always think of a novel as a knowledge source, and thus similar in some ways with our code. When writing a novel, the author typically wants the readers to understand and remember, if not all, at least some key aspects of the novel. That is exactly what software authors aim for.

In fiction repetition is useful in:

- emphasising what's important
- keeping certain aspects fresh in the reader's memory
- simplifying reading.

The same benefits can apply to repetition in code. Religiously eliminating repetition would remove these benefits as well, making the code harder to read.

## Memory and learning

*Tell the audience what you're going to say, say it; then tell them what you've said*
~ Dale Carnegie

Repetition plays a key role in how our memory works. Both in terms of acquiring new memories and using those memories. And this is extremely important if we want to improve our knowledge acquisition process.

Psychologists and neuroscientists differentiate between long-term memory and short-term memory [Foster09]. For the purpose of this article, we could consider working memory to be a synonym for short-term memory. This long-term and short-term memory would correspond to external storage and CPU registers, in computing parlance.

Since ancient times the best method to acquire knowledge in the long-term memory is rehearsal:

*Repetition is the mother of all learning*
~ Ancient proverb

Not only does it provide means to remember facts, but repetition also plays an important role in what's important and what's not. This is how we teach our children; we repeat a lot the facts the child needs to learn, and we repeat more often the more important facts.

Just like with CPU registers, the most important thing about working memory is that it is very limited. Early models claimed 7 different things (plus or minus 2); recent studies claim that without any grouping tricks, the memory is generally limited to 4 different things. [Mastin]

The other problem with short-term memory is that it easily decays (in the order of seconds). To keep things in short-term memory (e.g., in focus), we need to constantly repeat those things. [Mastin]

Let's take an example. Let's assume that we are exploring a new codebase and we have 100 functions of equal importance. We need to find 3 functions that match the given criteria. Without any form of grouping, or repeating what's essential, we would iterate over the space of functions trying to capture the needed functions. But, the problem is that after a few functions visited, our memory is filled with unimportant stuff. We constantly defocus, and our search procedure is hard. If the important information is repeated just enough, and/or if we have some sort of grouping, it would be much easier for us to find what we are looking for.

## When repetition is preferable

The reader must have repeatedly seen the downsides of repetition, so repeating them here would not be beneficial (pun intended). Instead, we are shall enumerate some of the benefits of repetition:

- **Emphasizes important aspects of the code.** Indeed, if the readers of the code see that a certain principle/pattern/design choice is applied several times, they can easily reach the conclusion that the principle/pattern/design choice is important. Conversely, if an important decision is not repeated at all, but there are other constructs/patterns repeated, then the importance of the decision can easily slip past the reader.
- **Ease the learning.** Repetition is the mother of all learning.
- **Create coherency.** If all items in a group have completely different characteristics, then the group is not coherent at all. To make a group coherent is to give all the elements in a group a certain characteristic. That is, to repeat the characteristic.
- **Keeps abstractions at the same level.** Refactoring techniques that aim to avoid repetition often make the new abstractions operate at different levels; this is typically bad for reading the code. If we want to keep the code at the same abstraction level, sometimes we need to duplicate some code.
- **Efficiency.** Sometimes, to achieve maximum efficiency, certain (low-level) code snippets need to be duplicated.

In the following subsections, we offer examples of when repetition applied in programming is good. However, as all design choices have both pros and cons, we also briefly indicate how not to apply the advice over-zealously.

### Repetition and code documentation

Code documentation is essentially repetition. It repeats (to a certain degree) what the code is saying, but in a manner that is more understandable by people. We all agree that code documentation is good, therefore, a form of repetition is good.

Then, we have repetition inside the documentation itself. For example, if we have an important architectural decision that we want the readers of the documentation to keep in mind, we should repeat it each time it provides insight into why certain things are designed in a certain way.

People should use repetition inside code documentation to highlight what's important.

However… don't overdo it. Avoid documenting things that frequently change. Avoid repeating *ad nauseam* decisions that are not important.

### Repetition in style

It's often a good idea to have a consistent style. But a consistent style can only be produced by repeating the same stylistic elements, so repetition is essential to a consistent style.

Style can apply to a variety of things: from formatting the code, to the way architectural decisions are made. All of them are important, but I would argue that the latter part is more important than the first one. There are only a few things that can damage understandability more than having a set of incoherent decisions. To come back to the 'codified knowledge' interpretation, having inconsistent knowledge is very harmful.

However… don't overdo it. I've seen a lot of time spent in minor formatting style debates. Stylistic unity is good, but that doesn't mean that we have to burn a developer at the stake when they add spaces in the wrong place. Don't be dogmatic on this; use tools like `clang-format` to take the burden off developers.

### Repetition in naming

Let's assume that one is writing code for a system based on the Model-View-Controller pattern. Naming all the model classes with the 'Model' suffix, all the view classes with the 'View' suffix and all the Controller classes with the 'Controller' suffix is generally a good idea. It provides coherence within the 3 groups of classes, and it makes it easier for readers to understand the code. Just by looking at the name of such a class, the reader can have a basic understanding of what the class does, without looking at the details.

Indeed, psychologies would label this naming repetition as a mnemonic system – a learning technique that aids information retention or retrieval in human memory.

However… don't overdo it. If mnemonics are good, it doesn't mean that we should heavily use identifier naming conventions all over the place. Form should never outlive content. For example, Hungarian notation is heavily criticized in modern software literature. [Martin09]

```
template <class II, class OI, class UOp, class P>
OI transform_if(II first1, II last1, OI result,
UOp op, P pred) {
  while (first1 != last1) {
    if (pred(*first1)) {
      *result = op(*first1);
      ++result;
    }
    ++first1;
  }
  return result;
}

template <class II, class OI, class UOp, class P>
OI transform_while(II first1, II last1, OI result,
UOp op, P pred) {
  while (first1 != last1)  {
    if (pred(*first1)) {
      *result = op(*first1);
      ++result;
    }
    else break;
    ++first1;
  }
  return result;
}
```

Listing 1

### Don't complicate algorithms to avoid repetition

At the function level, we often don't encounter pure repetition. Two functions that look very similar can have slight differences. If two functions are 90% the same, we cannot avoid repetition by simply reusing the code. We have to carefully separate the commonalities from the differences.

The main problem is the common part is too often interleaved with specifics of the two functions we want to collapse. How would we create a common function that can behave differently between the two cases? Often, we add parameters to the common function and pepper its body with **`if`** statements. And often the common function becomes more complicated than any of the original functions.

As I'm writing these lines, I can almost hear the *Clean Code* [Martin09] fans screaming in my ear: you should create new abstraction classes that implement different policies and pass them to your function. This may work in some cases, but my experience so far is that is seldom a better choice. Two problems with this approach are that we increase the overall complexity of the code (each new abstraction increases complexity) and that it makes the functions hard to follow (the reader may have to jump between different abstractions). But most of the time, a bigger problem arises: to make it work properly, one needs to mix different abstraction level (see the following subsection); this increases a lot the overall complexity.

Abstractions are best to be created as a result of the design process, not as a by-product of eliminating duplication.

There are a lot of cases in which two functions that are 90% identical should be kept separate. It's just easier to understand them independently. If you really want people to read them together, you can add a comment explaining that they are linked, and they do almost the same thing.

Take for example the two functions from Listing 1; it's a scoped down example, but it should be enough to prove our point. The only difference between the two functions is the **`else break;`** line. How would one unify the two functions without creating additional **`if`** clauses and without adding parameters that reflect implementation details? Would the code be more readable?

Similar ideas can also be found (and better presented) in [tef18] and [Metz16]. I think this entire section can be reduced to the following two quotes:

> *The problem with always using an abstraction is that you're pre-emptively guessing which parts of the codebase need to change together. "Don't Repeat Yourself" will lead to a rigid, tightly coupled mess of code. Repeating yourself is the best way to discover which abstractions, if any, you actually need.*
>
> ~ tef

> *Duplication is far cheaper than the wrong abstraction*
>
> ~ Sandi Metz

However… don't overdo it. Sometimes you can shift the abstractions in such a way in which you can eliminate the duplication; analyze each situation separately and don't religiously decide to duplicate code or avoid duplication.

### Avoid mixing different abstraction levels

Two functions that do the same thing should not be combined if they operate at different abstraction levels or they belong to unrelated modules. It adds a great burden on the developer who needs to keep changing the context to properly understand the code.

For example, summing numbers and summing back accounts are two completely different things; one should not combine the functions that perform the summation.

Let us take another example that created a lot of heat in the last couple of months. [Aras18] [Niebler18]. We aim to print the first N Pythagorean triples (computed in a naive way). A simple C-style solution to this problem is presented in Listing 2. It uses an imperative, plain C-style with one abstraction level.

With the C++20 ranges feature, Eric Niebler proposes the implementation from Listing 3 (comments stripped out), arguing for more genericity [Niebler18].

I believe that all readers would consider the latter code much harder to read. There are multiple reasons why this second version is more complex, but one of them is too much change in the abstraction level. Let's analyze this.

The code in Listing 3 mixes imperative style (see **`return`** statements), with functional style (see piping operator), with more mathematical abstractions (Semiregular, iota), range-specific abstractions (**`transform`**, **`join`**, **`take`**), range building blocks abstractions (**`view_interface`**) and C++ in-depth abstractions (**`IndirectUnaryInvocable`**, concepts, move semantics). Too many abstraction levels. If you saw a **`view::transform`**, a **`view::join`** and a **`view::take`** in the same code, it would be fine, even if you type more: all the abstractions are at the same level; but don't mix the levels too much.

A common side effect of using multiple abstraction levels in the same code is the need for more code to bridge between the abstractions. Having too much plumbing code is a good indication that there are multiple abstraction levels involved. And overall, this will make the understanding of the code much harder.

Besides understandability costs, the ranges solution also have pretty high compilation-time costs as Aras points out [Aras18].

Related to this, overuse of generics in the name of eliminating duplicates can lead to major pain points. I had the misfortune to see a lot of cases in which templates are used in the name of genericity, and eliminating

```
int i = 0;
for (int z = 1; ; ++z)
  for (int x = 1; x <= z; ++x)
    for (int y = x; y <= z; ++y)
      if (x*x + y*y == z*z) {
        printf("%d, %d, %d\n", x, y, z);
        if (++i == n)
          return;
      }
```

Listing 2

duplicates, but if you would just write the code without templates, with all the duplication, it would be far smaller than the code with templates.

However… don't overdo it. Taking the advice in this section too dogmatically would prevent you from creating any abstraction or very little abstraction. Of course, software without good abstraction is bad software.

### Repeating the data

Repetition can happen at the code level, but also on the data level. There are cases in which repeating the data leads to a cleaner design and/or improved efficiency.

Such is the case with multithreaded code. Instead of having multiple threads accessing the same data source, with the possibility of data-races and with mutexes (read bottleneck instead of mutex), it's sometimes much simpler to duplicate the data. If each thread would have a copy of the data, then there would be no race conditions when accessing the data, and no need to protect the data access. In this case, synchronizing the data between thread can be done by sending messages from one thread to another (which typically involves other data copies).

```
template<Semiregular T>
struct maybe_view : view_interface<maybe_view<T>>
{
  maybe_view() = default;
  maybe_view(T t) : data_(std::move(t)) {
  }
  T const *begin() const noexcept {
    return data_ ? &*data_ : nullptr;
  }
  T const *end() const noexcept {
    return data_ ? &*data_ + 1 : nullptr;
  }
private:
  optional<T> data_{};
};

inline constexpr auto for_each = []<Range R,
  Iterator I = iterator_t<R>,
  IndirectUnaryInvocable<I> Fun>(R&& r,
  Fun fun) requires
  Range<indirect_result_t<Fun, I>> {
    return std::forward<R>(r)
      | view::transform(std::move(fun))
      | view::join;
   };

inline constexpr auto yield_if =
  []<Semiregular T>(bool b, T x) {
    return b ? maybe_view{std::move(x)}
      : maybe_view<T>{};
  };

using view::iota;
auto triples =
  for_each(iota(1), [](int z) {
    return for_each(iota(1, z+1), [=](int x) {
      return for_each(iota(x, z+1), [=](int y) {
        return yield_if(x*x + y*y == z*z,
          make_tuple(x, y, z));
      });
    });
  });

for(auto triple : triples | view::take(10)) {
  cout << '('
    << get<0>(triple) << ','
    << get<1>(triple) << ','
    << get<2>(triple) << ')' << '\n';
}
```

Listing 3

Another case in which data repetition is used is for pure performance reasons. Cache locality is typically important for performance critical code, and cache locality often involves data copies. The classical example is improving the reads from external memory: one can often cache it in memory, and then, based on the algorithm, cache it in L2, L1 and CPU registers. Read duplicate it instead of cache it.

However… don't overdo it. Of course, both of the cases described here should not be applied blindly. One should typically have a good design/measurements before applying the techniques described here.

## Conclusions

Andrew Hunt justifies the DRY principle mainly by the need to avoid maintenance work [Hunt99]. But we agreed that writing and maintaining code is not the most important part of a programmer's job; instead, reading, understanding and reasoning about the code is far more important. And repetition can help with this. Therefore, the DRY principle is not as justified as one would believe. And, again, ironically, it should not be repeated as often.

The purpose of this article was not to convince the reader of how bad the DRY principle is; in general, this can be a good principle. The goal was to draw attention to the fact that applying the principle doctrinally can be harmful. The reader, who is or aspires to be a virtuous programmer, needs to balance the pros and cons when applying this principle. Therefore, it gives me great pleasure to end with Aristotle's golden rule:

*Virtue is the golden mean between two vices, the one of excess and the other of deficiency.*
~ Aristotle

## References

[Aras18] Aras Pranckevi?ius (2018), 'Modern' C++ Lamentations, http://aras-p.info/blog/2018/12/28/Modern-C-Lamentations/

[Foster09] Jonathan K. Foster (2009), *Memory: A Very Short Introduction*, Oxford University Press

[Henney19] Kevlin Henney (2019) 'What do you mean?', *ACCU Conference 2019*, https://www.youtube.com/watch?v=ndnvOElnyUg

[Hunt99] Andrew Hunt and David Thomas (1999), *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Professional

[Martin09] Robert C. Martin ed. (2009), *Clean Code: A Handbook of Agile Software Craftsmanship*, Pearson Education

[Mastin] Luke Mastin, 'Short-term (working) memory', http://www.human-memory.net/types_short.html

[Metz16] Sandi Metz (2016), 'The wrong abstraction', https://www.sandimetz.com/blog/2016/1/20/the-wrong-abstraction

[Niebler18] Eric Niebler (2018), 'Standard Ranges', http://ericniebler.com/2018/12/05/standard-ranges/

[tef18] tef (2018), 'Repeat yourself, do more than one thing, and rewrite everything', https://programmingisterrible.com/post/176657481103/repeat-yourself-do-more-than-one-thing-and

# On IT and... $CO_2$ Footprints

Recent headlines declare a climate emergency. Sergey Ignatchenko considers how we can make an impact on IT's carbon footprint.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Disclaimer #2: all the numbers within this article are calculated 'on the back-of-an-envelope' (see, for example, [NoBugs17] for discussion), and are at best accurate within an order of magnitude; still, even being accurate within an order of magnitude is good enough for our current purposes.

These days, I happen to know quite a few fellow programmers who are bicycling to work; what's interesting, however, is that most of them are not doing it for fun, but instead are arguing that it is a Good Thing™ to save on their $CO_2$ footprint. I've heard this line of argument soooo many times (if you're not familiar with it, take a look, say, at [Handy18]), that I decided to make some back-of-an-envelope calculations to see whether we as software developers can save a bit more than that. BTW, even if you happen to believe that global warming has nothing to do with $CO_2$, only a few will disagree (I hope) that burning non-renewable resources for nothing is a Bad Thing™.

In [Handy18], it is mentioned that over 16 years the author saved 3 metric tons of $CO_2$ by bicycling to the office instead of driving; that's saving about 190 kilos of $CO_2$ per year. Now, let's see what IT-related changes can do in terms of saving the world (from global warming, that is).

Let's consider a few real-world examples.

## Example 1: Game with 100K simultaneous players

For the purposes of our first example, let's assume that you happen to work on a PC-based game with 100,000K simultaneous players, which uses about 100W when it is running. Now, if you can come up with a trick which saves mere 1% of this power, it means that you'll be saving 1W×100,000=100kW of power at each and every moment, which translates into 2,400kWh per day, or 876,000kWh per year. Now, we can use [CarbonFootprint] to translate this into $CO_2$ and discover that one such optimization will save a whopping 270 tons of $CO_2$ per year(!) – that will cover 1,400 people (such as those in [Handy18]) bicycling to the office and back.

How we can save that 1% of power is a different story. Just as one example, often much more than that can be saved in practice by switching to V-sync by default (that is, in the absence of G-sync/Freesync). I certainly don't want to start a war on which is better for the end-user – screen tearing without V-sync or lag without it (especially because the answer is very game-specific) – but whenever you're in doubt, V-sync can be preferred by default as being more environmentally friendly.

We usually print purely technical articles in *Overload*, but hope that this article on a broader subject is still of interest to our readership.

## Example 2: Multithreaded video codec with 1M active install base

Now, let's assume you're writing a video codec, and you're really good at it, so you have 1 million people actively using your codec, with each of these people using it 10% of the time, and while they're using it, it eats 100W of power.

Also, let's assume that your codec uses fine-grained multithreading, and that when it uses 4 CPU cores – it gets a 2× wall-clock speed-up compared to running on a single core. But this means that in multithreaded mode (which you enabled by default, of course), it takes 4× the power of a single core, multiplied by 0.5× of the time needed to perform the task. This means that overall power consumption will increase by about 4×0.5=2× (in fact, usually a bit less due to interplay with other components, but not by much). In other words, in multithreaded mode only the power of 2 CPU cores is actually used to produce something useful, and anything the other 2 CPUs do goes towards paying for synchronization overheads. This means that you're wasting about 50% of that 100W of power per box – that's about 50W wasted per box!

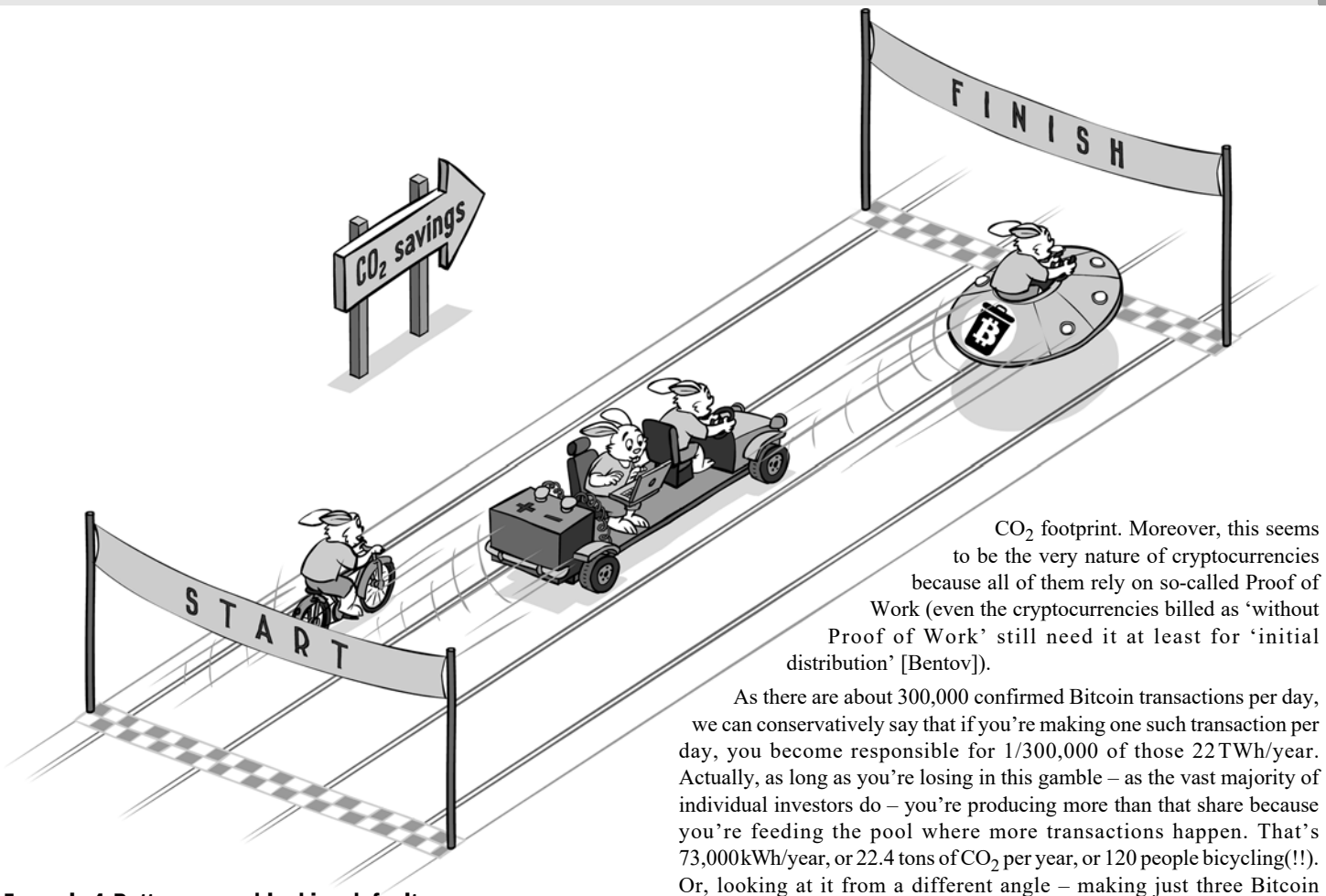
Now, if you rewrite your codec so that it separates jobs between threads on a per-keyframe basis (which means that there is no interaction between the data in different threads, hence there is almost zero thread sync and almost zero overhead), you can get most of that 50W back; let's assume you've got 40W back. With your user base, this translates into 100,000 simultaneous users×40W=4MW. That's 40× more than that of our 1st example – and translates into savings equivalent to those of 56,000 people bicycling per year(!!).

## Example 3: Single R-R op within Linux scheduler

In our 3rd example, let's assume that you've noticed how to save one single R-R operation within Linux scheduler. Now, let's assume that Linux is running on a billion devices (which is a rather conservative estimate, if we take into account all Androids and all IoT stuff), and that the appropriate part of the scheduler is run every 10ms. This means that you'll be saving one R-R operation on $1\times10^9$ devices×(1sec/10ms)=$1\times10^{11}$ times every second. Assuming that an R-R operation takes one CPU cycle, this will very roughly correspond to running a hundred 1GHz CPU cores all the time – and assuming that each of these cores will eat 30W (that's accounting for associated power consumption in memory etc.), this means that you've got savings of 3kW (or 42 people bicycling instead of driving each and every day). That's just for saving one single R-R operation(!!).

**'No Bugs' Hare** Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

**Sergey Ignatchenko** has 15+ years of industry experience, including being a co-architect of a stock exchange, and the sole architect of a game with 400K simultaneous players. He currently holds the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com




### Example 4: Better screen blanking defaults

Now, let's move from considering pure programming into other computer-related things. For the purposes of our 4th example, let's assume that you have the ability to change the default screen blanking time on a desktop system from 30 minutes down to 10 minutes, with a billion such systems working all over the world. Let's further assume that 80% of the users don't change defaults, that the chances that after 10 minutes a user is still staring at the monitor are 10%, that screen blanking fires three times a day, and that the monitor uses 20W when it is not blanked.

Then, this simple change in defaults will result in savings of (30-10 minutes)$\times 0.8 \times 0.9 \times 10^9$ users$\times$20W$\times$5times/day = ⅓$\times$3 hour/day $\times$14W $\times$ $10^9$ = 0.014kWh/day $\times$ $10^9$ ~= $5\times10^9$ kWh/year, or 1,500,000 tons of $CO_2$ per year, which is equivalent to eight million people changing from a car to a bicycle for getting to/from the office.

NB: BTW, if you change just your own 100% screen saver into screen blanking, you'd save about 60 kilos of $CO_2$ per year – which is ⅓ of the savings from bicycling [Handy18]; not a bad gain from such a non-effort.

### Example 5: Giving up on cryptocurrency speculations

Each time you're making your Bitcoin 'investment' (actually, most of the time it is pure speculation – see [WallStreetMojo] for the difference between the two) transaction – think how much $CO_2$ waste it creates.

The whole Bitcoin industry is estimated to use a whopping 22 terawatt-hours per year [Economist] – that's $2.2\times10^{10}$kWh per year, equivalent to 35 million people bicycling to their offices – which is about the population of the whole of California (and about ⅓ of the workforce in the whole US).

In other words,

> *if we all simply give up on Bitcoins (which have a mostly speculative and dark-market value – their use for other purposes such as processing real-world non-dark transactions still hasn't really started) – this would result in* $CO_2$ *savings comparable to the whole of California bicycling instead of driving(!!).*

By trying to speculate on Bitcoin (Ethereal, whatever else), we're not only gambling our hard-earned savings, but are also drastically increasing our $CO_2$ footprint. Moreover, this seems to be the very nature of cryptocurrencies because all of them rely on so-called Proof of Work (even the cryptocurrencies billed as 'without Proof of Work' still need it at least for 'initial distribution' [Bentov]).

As there are about 300,000 confirmed Bitcoin transactions per day, we can conservatively say that if you're making one such transaction per day, you become responsible for 1/300,000 of those 22TWh/year. Actually, as long as you're losing in this gamble – as the vast majority of individual investors do – you're producing more than that share because you're feeding the pool where more transactions happen. That's 73,000kWh/year, or 22.4 tons of $CO_2$ per year, or 120 people bicycling(!!). Or, looking at it from a different angle – making just three Bitcoin transactions per year is enough to offset you bicycling all the year(!!!).

> Yes, *instead of bicycling to the office every day, just give up on three Bitcoin transactions per year – you will get the same* $CO_2$ *reduction*.

## Conclusion

Sure, bicycling does reduce $CO_2$ footprint. However, in IT there are lots of ways to save even more (often much more) than by bicycling; it can be as simple as optimizing your own program so it uses less power, readjusting screen your saver, or even giving up on just three Bitcoin transactions per year(!). ■

## Bibliography

[Bentov] Iddo Bentov, Ariel Gabizon and Alex Mizrahi (no date) 'Cryptocurrencies without Proof of Work', available at: https://fc16.ifca.ai/bitcoin/papers/BGM16.pdf

[CarbonFootprint] Carbon Calculator, available at: https://www.carbonfootprint.com/calculator.aspx

[Economist] 'Why bitcoin uses so much energy', in *The Economist explains*, published 9 July 2018 at https://www.economist.com/the-economist-explains/2018/07/09/why-bitcoin-uses-so-much-energy

[Handy18] Susan Handy, 'Want to save tons of greenhouse gases? Bike it.', in *Science & Climate*, published 6 September 2018, available at: https://climatechange.ucdavis.edu/what-can-i-do/want-to-save-tons-of-greenhouse-gases-bike-it/

[Loganberry04] David 'Loganberry', Frithaes! - 'An Introduction to Colloquial Lapine!', available at http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs17] 'No Bugs' Hare (2017) 'The Importance of Back-of-Envelope Estimates', *Overload* 137, available at: https://accu.org/index.php/journals/2341

[WallStreetMojo] 'Differences Between Investment and Speculation' in *WallStreetMojo*, available at: https://www.wallstreetmojo.com/investment-vs-speculation/

# Use UTF-16 Interfaces to Ship Windows Code

Character encoding can cause problems. Péter Ésik explains why UTF-16 interfaces help on Windows.

Listing 1 is a small program that takes a file path as a parameter, and queries its size. Even though **`stat`** is a POSIX function, it happens to be available on Windows as well, so this small program works on both POSIX platforms and Windows. Or does it? Let's try it out with two test files. For `test.txt`, it correctly reports the file's size. For `Hello, мир.txt`, however, the **`stat`** call fails (on my machine), even though the file clearly exists (see Figure 1). Why is that?

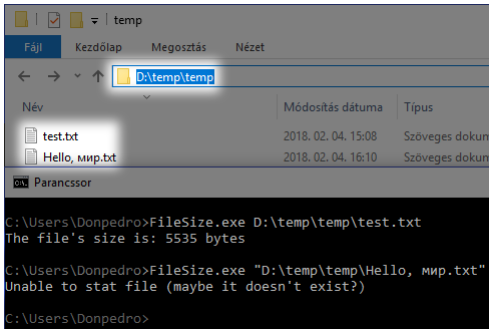



Figure 1

## The 'ANSI' vs. UTF-16 story in five minutes (or less)

Back in the day, character encodings were quite rudimentary. The first encoding widely adopted by computer systems was ASCII [Wikipedia-1] (a 7-bit encoding) , capable of encoding the English alphabet and some other characters (numbers, mathematical symbols, control characters, etc.). Of course, the obvious need arose to encode more characters as users expected computers to speak their language, to type their native letters in e-mails, etc. 8-bit encodings provided a partial solution: code points 0–127 were the same as ASCII (for compatibility), while extra characters were encoded in the range 128–255.[1] Those extra 128 code points were not enough to encode all letters of all languages at once, so character mappings were applied, most commonly known as code pages.

This means that a character or a string that's encoded like this has no meaning in itself, you need to know what code page to interpret it with (this is somewhat analogous with files and their extensions). For example, code point 0x8A means ä (lowercase a with an umlaut) if you interpret it using the Macintosh Central European encoding [Wikipedia-2], but encodes Š (uppercase S with caron) if you use the Windows-1252 [Wikipedia-3] (Latin alphabet) code page.

**Péter Ésik** Péter has been working as a C++ software developer for 5 years. He has a knack for everything low level, including (but not limited to) OS internals, assembly, and post-mortem crash analysis. His blog can be found at http://peteronprogramming.wordpress.com, and he can be contacted at peter.esik@gmail.com

1. There are some languages with much more symbols than 128 or 255 (Japanese, Chinese, etc.), which led to the invention of DBCS/MBCS character sets. I'm not mentioning them here for simplicity.

```
#include <iostream>
#include <sys/stat.h>

int main (int /*argc*/, char* argv[])
{
  struct stat fileInfo;
  if (stat (argv[1], &fileInfo) == 0) {
     std::cout << "The file's size is: "
       << fileInfo.st_size << " bytes\n";
  } else {
    std::cout << "Unable to stat file (maybe it
      doesn't exist?)\n";
  }
}
```

Listing 1

This approach has two obvious problems: first, it's easy to get encodings wrong (for instance, .txt files have no header, so you simply can't store the code page used), resulting in so-called mojibake [Wikipedia-4]. Second, you can't mix and match characters with different encodings easily. For example, if you wanted to encode the string "Шнурки means cipőfűző" (with Windows code pages), you would have to encode "Шнурки" with code page 1251 (Windows Cyrillic) [Wikipedia-5], " means " with a code page of your choice (as it contains ASCII characters only), and "cipőfűző" with code page 1250 (Windows Central European) [Wikipedia-6]. To correctly decode and display this string later, you would have to store which code pages were used for which parts, making string handling inefficient and extremely complex.

Because of problems like these, encodings were desired that could represent 'all' characters at once. One of these emerging encodings was UCS-2 (by the Unicode working group), which used 16-bit wide code units and code points. Windows adopted UCS-2 quite early, Windows NT 3.1 (the very first OS of the NT series, released in 1993) and its file system, NTFS, used it internally. Even though the 32-bit Windows API debuted with NT 3.1 as brand new, support for 8-bit encodings was still necessary.[2] As UCS-2 used 16-bit code units, and the C language does not support function overloading, Microsoft introduced two versions of every API function that had to work with strings (either directly or indirectly): a UCS-2 version, with a W suffix ('wide', working with **`wchar_t`** strings), and one for 8-bit code paged strings, with an A suffix ('ANSI'[3], working with **`char`** strings).

2. One major reason for this was (other than UCS-2 not being widespread at the time) that the consumer line of Windows OSes (95, 98, etc.) had very limited support for UCS-2, but applications targeting Win32 had to run on both lines of Windows.
3. Technically, it's not correct to call these functions 'ANSI' versions, as none of the supported code pages are ANSI standards. This term has historical roots, as the first Windows code page (1252) was based on an ANSI draft. On recent versions of Windows 10, the ACP can be set to UTF-8. Therefore, it's best to think about 'ANSI code pages' as 'some encoding that's not UTF-16'.

**The problem is that there might be characters in the UTF-16 command line that have no representation in the currently active code page**

The A functions act as mere wrappers, usually[4] they just convert the string parameters and forward the call to the corresponding W version. So for example, there is no such function as **`MessageBox`**, there is only **`MessageBoxA`**, and **`MessageBoxW`**. Depending on the strings you have, you need to call the appropriate version of the two.[5]

Which code page is used to interpret strings in the A family of functions? Is there a code page parameter passed? No, they use a system-wide setting called the *active code page*, located in the registry at HKLM\SYSTEM\CurrentControlSet\Control\Nls\CodePage\ACP. This value is decided based on your region you choose at installation time, but can also be changed later in the Control Panel.

Eventually, UCS-2 evolved into UTF-16, and starting with Windows 2000, the OS had support for it. Since UCS-2 is fully compatible with UTF-16, programs didn't need to be rewritten or even recompiled.

## Back to the test program

Armed with this knowledge, it's easy to see why the small test program doesn't work for certain files. This is what happens:

1. The program is started (with whatever parameters).
2. Very early in the startup phase, Windows converts the (native) UTF-16 command line to an 'ANSI' string using the active code page, and stores it in a global variable.
3. Because regular **`main`** was used (with 'narrow', **`char`** parameters) in this application, early in the startup phase the CRT queries the command line with **`GetCommandLineA`** (this just returns the global that was set up by the previous step), converts it into an array, and passes it down to **`main`**.

The problem is that there might be characters in the UTF-16 command line that have no representation in the currently active code page. For example, my computer's locale is set to Hungarian, therefore my ACP is 1250 (Windows Central European) [Wikipedia-6]. Cyrillic characters such as м, и, and р have no representation in this encoding, so when the UTF-16 to 'ANSI' conversion is performed, these characters are replaced with question marks (see Figure 2).[6] When **`stat`** is called with the string **`"D:\temp\temp\Hello, ???.txt"`** (which by the way involves an 'ANSI' to UTF-16 conversion internally), of course it fails, because there is no file named `Hello, ???.txt` in that directory.

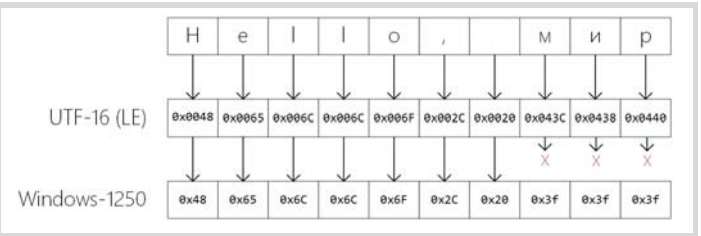



**Figure 2**

## cURL

It's not that hard to bump into applications or libraries suffering from these problems. cURL, for example, is one of them. Now don't get me wrong, I'm in no way saying that it's a badly written piece of software, quite the contrary. It's a battle-tested, popular open source project with a long history and a plethora of users. Actually, I think this is what makes it a perfect example: even if your code is spot on, this aspect of shipping to Windows is very easy to overlook.

For file IO, cURL uses standard C functions (such as **`fopen`**). This means that for example, if you want your request's result written into an output file, it will fail if the file's path contains characters not representable with the system's current 'ANSI' code page.

Another example is IDN (internationalized domain name) handling. cURL does support IDNs, but let's see what happens if I try it out using the standalone command line version (see Figure 3).

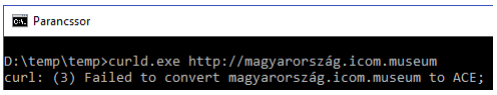



**Figure 3**

Even though `magyarország.icom.museum` exists, and its string form is perfectly representable with my machine's ACP (1250), cURL fails with an error. Looking at the source code quickly reveals the culprit:

- cURL needs to convert the IDN to so-called Punycode [Wikipedia-7] before issuing the request.
- It does so with **`IdnToAscii`** [WindowsDev], but this function expects a UTF-16 input string.
- Even though the original string (which originates from the command line parameter) is an 'ANSI' string, conversion to UTF-16 is attempted assuming it's UTF-8. This makes the conversion fail, and thus cURL aborts with an error message.

cURL developers are aware of this category of problems: it's listed on their known bugs page [curl].

4. One exception I know of is **`OutputDebugString`**, where the 'ANSI' version is the native one (**`OutputDebugStringW`** will convert to 'ANSI' and call **`OutputDebugStringA`**)
5. It's possible to create programs that can be compiled to support either the W or A interfaces without source changes, using predefined macros [Microsoft18]. Nowadays, however, that's highly irrelevant. If you are writing programs that target modern Windows versions (only NT), there is almost absolutely no reason to use A interfaces.
6. The exact mappings are defined in `.nls` files located in the System32 directory.

some people think that ... using char strings and 'ANSI' interfaces on Windows is 'good enough'

```
#ifdef _WIN32

using nchar = wchar_t;
using nstring = std::wstring;

#define NSTRLITERAL(str) L##str
#define nfopen _wfopen
/* ... */

#else

using nchar = char;
using nstring = std::string;

#define NSTRLITERAL(str) str
#define nfopen fopen
/* ... */

#endif  // #ifdef _WIN32
```

Listing 2

## Solution

The solution is in the title of this article: use UTF-16 (native) interfaces on Windows. That is:

- Instead of regular **`main`**, use **`wmain`** as an entry point, which has **`wchar_t`** string arguments.
- Always use the wide version of runtime functions (**`_wfopen_s`** over plain **`fopen`**, **`wcslen`** instead of **`strlen`**, etc.).
- If you need to call Win32 functions directly, never use the 'ANSI' version with the A suffix, use their UTF-16 counterparts (ending with W).

While this sounds great on paper, there is a catch. You can only use these functions on Windows, as:

- Some of the widechar runtime functions are Windows-only (such as **`_wfopen`**).
- The size and semantics of **`wchar_t`** are implementation defined. While on Windows it's a 2-byte type representing a UTF-16 code unit, on POSIX systems it's usually 4 bytes in size, embodying a UTF-32 code unit.

One possible solution is to utilize **`typedef`**s and macros. See Listing 2.

This simple technique can go a long way (it can be done somewhat more elegantly, but you get the idea), unless you need to exchange strings between different platforms (over the network, serialization, etc.).

## Closing thoughts

I know some people think that the problem presented in this article is marginal, and using **`char`** strings and 'ANSI' interfaces on Windows is 'good enough'. Keep in mind though that in commercial environments the following situation is not that rare:

- Company X outsources work to company Y, but they reside in different parts of the world.
- Therefore, the computers of company Y have a different ACP from those of company X.
- It's very likely that the outsourced work involves using strings in company X's locale, which will be problematic on Windows, if the software(s) used for doing said work misbehaves in this situation.

Don't be surprised if a potential client of yours turns down a license purchase because of problems like this. ■

## References

[curl] 'Known Bugs', curl: https://curl.haxx.se/docs/knownbugs.html#can_t_handle_Unicode_arguments_i

[Microsoft18] 'Working with strings', published on 31 May 2018 at https://docs.microsoft.com/en-gb/windows/desktop/LearnWin32/working-with-strings

[Wikipedia-1] ASCII: https://en.wikipedia.org/wiki/ASCII

[Wikipedia-2] Macintosh Central European Encoding: https://en.wikipedia.org/wiki/Macintosh_Central_European_encoding

[Wikipedia-3] Windows -1252 code page: https://en.wikipedia.org/wiki/Windows-1252

[Wikipedia-4] Mojibake: https://en.wikipedia.org/wiki/Mojibake

[Wikipedia-5] Windows-1251: https://en.wikipedia.org/wiki/Windows-1251

[Wikipedia-6] Windows-1250: https://en.wikipedia.org/wiki/Windows-1250

[Wikipedia-7] Punycode: https://en.wikipedia.org/wiki/Punycode

[WindowsDev] IdnToAscii function, Windows Dev Center: https://docs.microsoft.com/en-gb/windows/desktop/api/winnls/nf-winnls-idntoascii

# ACCU Conference 2019: Reports

ACCU holds an annual conference. Several attendees tell us what they learnt there this year.

## From Felix Petriconi

What a great week has just passed. I attended the ACCU conference in Bristol, UK for the fifth time. On my flight back, I started to make these notes. I am still overwhelmed by all the information and the impressions.

For me, the conference started one day before the actual sessions. I attended one of the offered one day tutorial workshops. My choice was the 'Introduction to Rust' workshop by Katharina Fey. She clearly described the differences from other languages like C++, so the introduction was quite easy. Piece by piece, she guided us through our first programming tasks. From that day, I received a good overview of the language, and the advantages of the extreme strong type system and its focus on value semantics.

The Agile Bath & Bristol User Group again held their meeting in the evening at the conference's location, and Giovanni Asproni [Asproni19] talked about 'One Team, Two Teams, Many Teams: Scaling Up Done Right'. It was an excellent talk where he emphasized that scaling up teams only works in a limited way and that problems that already exist in a single team are increased disproportionately when more teams work on the same project. It would be great if more project manager or other leads in companies could know this too.

For the very first time, we had a small intro video five minutes just before the conference. Dom Davis [Davis19] created an usual boot sequence, including a funny animation by Dylan Beattie of Eric Holscher's Pac-Man rule and the schedule of the day. After a welcome from the Chair, Russel Winder, the conference was opened by Angela Sasse, formerly a professor at the Department of Computer Science, University College London, UK and now of Human-Centric Security at the Ruhr-University Bochum, Germany. In her keynote, she emphasized the importance of the need to communicate between the security department of an organization and the users of the IT infrastructure. If users are not convinced by the importance of certain security measures, they will bend the rules as much as possible to make them work in their environment. As well, I learned that there are already organizations out there that offer – beside intrusion or hacking tools – first-class full support with help desks etc. for their 'customers'.

The first regular session that I attended was given by Vittorio Romeo on higher-order functions. He gave an overview of higher-order functions that already exist in the C++ language in this very well structured talk. In the second half, he explained the concept of a `function_ref` that he is currently proposing for the C++ standard. This is a construct for referencing functions and function objects, similar to `string_view` for character sequences.

Unfortunately, I could not attend any session after lunch even they were all very interesting, because I had to go to my own. Here I spoke about the enormous number of traps that one can step into when one uses low-level synchronization primitives likes atomics, mutex etc. and that high-level abstractions take away most of the pain. At this point I want to thank John Lakos for the engaged discussion within the session. This is the way that I wish more sessions would be, because the word 'conference' comes from Latin 'conferre', to compile, to discuss, to debate, to confer.

Abstracts for most of the sessions are available at: https://conference.accu.org/2019/sessions.html

Recordings of the sessions are available on the ACCU Conference YouTube channel at: https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw

Visit https://conference.accu.org for details of other conferences planned for 2019 and 2020.

The last session of the first day that I attended was given by Patricia Aas about the 'Anatomy of an Exploit'. The given example was based on exploits that were ten or more years old because of the time limit of 90 minutes for the session. She described very clearly the way of thinking that is necessary to understand how exploits work. I very much liked her illustration of 'Programming the Weird Machine'. Here one leaves the intended state machine of a program via the vulnerability and goes into a new, weird state machine.

**Felix Petriconi** Felix studied electrical engineering and has been a programmer since 1993. He is a programmer and development manager at the MeVis Medical Solutions AG in Bremen, Germany, where he develops radiological medical devices. A regular speaker at the C++ user group in Bremen and a member of the ACCU's conference committee, he can be contacted at felix@petriconi.net

**Stefan Turalski** Stefan is a software developer who, contrary to common sense and fond memories of working for a software house, still makes bits flow between various financial institutions. Incurable optimist, mechanical sympathiser, who believes it's possible to learn to play piano, Clojure, Haskell or even C++ one day, hence takes it easy coding in C#.

**Ori Ben-Shir** Ori is currently a C++ software engineer at SentinelOne. Learning the Rust programming language, and documenting the experience in his blog, 'Afternoon Rusting'. Passionate about programming languages, exploring their implementation and how language authors shape the way we code with their design choices. Follow @oribenshir on Tweeter or contact at oribenshir@gmail.com

**Mathieu Ropert** Mathieu has worked in various areas, ranging from kernel programming to web development, financial software, databases and videogames. His current favourite subject is package management, and he thinks the lack of it has been holding back C++ for years now. Contact him via Twitter: @MatRopert

**Anthony Williams** Anthony is the author of *C++ Concurrency in Action*. As well as working on multi-threading libraries, he develops custom software for clients, and does training and consultancy. Despite frequent forays into other languages, he keeps returning to C++. He is a keen practitioner of TDD, and likes solving tricky problems. Contact him at anthony@justsoftwaresolutions.co.uk.

**Conference after conference, I feel more a part of a family that cares for each other. Each year is a reunion**

The first day closed with a set of very entertaining and informative lightning talks hosted by Chris Oldwood and assembled by CB Bailey. At this point, many thanks again for their work assembling the presentations between the sessions. I especially recall Dom Davis' one, 'Where is Kevlin Henney'.

After an hour of 5-minute presentations, all attendees could discuss their first impressions during the welcome reception sponsored by mosaic.

The second day was opened with a keynote by Herb Sutter in which he presented a proposal for the C++ Standard. This proposal has the purpose of removing one of the biggest hurdles that many C++ users have to using the complete feature set – including exceptions – by extending the language with the keyword `throws` (`throw` was marked as deprecated with C++11 and its functionality better expressed with `noexcept`). The central idea is to use the existing return channel of a function for two different purposes: like a union, either for the regular return value or, in case of an error, for a `std::error_code`, or everything similar to `boost.outcome` that has become available with the recent release 1.70. As two years ago, when Herb presented his Meta Class proposal, this was a great and very successful session.

Before lunch I attended Timur Doumler's talk about a low-level audio interface that he and a group of contributors are currently proposing for the C++ Standard. Before he went into the technical details, he gave a very good introduction to buffers, channels, sampling rate etc. The extension to the standard will provide a unique interface for audio devices over different platforms. It will remove the diversity of the approaches currently needed to bring audio signals into or out of the machine. From my point of view, it looks both very promising and like it could go into the one of the next versions of the standard.

After lunch, Ahto Truu gave a good introduction into hash trees and possible use cases of them in his presentation.

I hosted a 'C++ Pub Quiz' as one of the last sessions on that day. It was held in the hotel bar with free beverages sponsored by the conference headline sponsor, Bloomberg. Many thanks again! The nearly 80 attendees had to figure out or guess what eleven pieces of code would print to the console. The format was developed by Olve Maudal several years ago and he allowed me to continue it. Never before have so many nice terms like 'twisted mind', 'sadist' and similar been said to me :-). But that was fine, I deserved probably it.

The day ended again with a set of entertaining and instructive lightning talks.

Friday, the third day of the conference, was opened with a keynote by Paul Grenyer. He gave an historic overview of how he and a group of supporters were able to create an active developer community in Norfolk that even today has its own developer conference, nor(DEV):con.

After lunch I listened to Dom Davis' entertaining and informative parable of creating a presentation with graphics that needed to be done by an external contractor. The parallels to a software development process were obvious and intended.

After lunch, I attended a set of 20 minutes presentations.

At the end of the day, I listened to Niall Douglas on 'Elsewhere Memory'. He led the audience through an expert-level set of proposals to change the C++ Memory Model in a way that e.g. shared memory is regularly supported by the C++ Standard. Today this works 'by accident' because of the individual machine assumptions of the compiler and the underlying hardware. It is a pity that this would be Niall's last presentation at ACCU for a long time, but his engagement on the C++ Standard's committee consumes most of his spare time.

After a final set of lightning talks, the conference dinner was served. Here we had again the rule that all the speakers had to enter the room first and distribute themselves among the tables with at least one seat empty between them. This meant that the non-speakers, who came in afterwards, sat between two speakers. Such nice conversations could emerge. After each course, the non-speakers had to switch places to give themselves the opportunity of sitting beside a different speaker and having a chat. After the desert, we were entertained by Echoborg [Echoborg], when several members of the audience had to try to make contact with an artificial intelligence by talking to it. The first approaches e.g. by trying the Voight-Kampff test (from the film Blade Runner) on the AI or by just repeating everything the AI has just said were very entertaining but not very successful. Later, with different strategies, we managed to 're-program' the AI's core routines, which ended the 90-minute session. It was really an intelligent entertaining evening.

On the last day, the conference did not start with a keynote, but with regular sessions. For me, it was Kevlin Henney asking 'What do You Mean?'. He explained, in an excellent way, why meaning is so important in all aspects of software!

After lunch, I attended two 40-minute sessions by James Cain and Dom Davis. To both, again our thanks that they stepped into the gap on very short notice after a speaker could not give his presentation. James introduced an open source project that he is working on. It offers a virtual file system running in Windows user mode. It is something that I want to look into when my time permits. Dom later gave a live-programmed introduction to the Go language.

After Russel, the Conference Chair, expressed his thanks to Archer-Yates, the organizers of the conference, all the speakers, the sponsors and all attendees, Kate Gregory gave her closing keynote. She made it clear to us that code is emotional and that we really should take care about how we keep our code clean and how we name things.

Conference after conference, I feel more a part of a family that cares for each other. Each year is a reunion. Summing up, this year it made it clearer for me that the sessions make up about 50% of the conference. The other 50% is the direct interaction between the other attendees. While discussing problems of my domain and listening to problems of other domains, I learned so much that I am already looking forward to ACCU 2020!

Disclaimer: I want to add that I am somewhat biased, because I am a member of the conference committee and I am the Deputy Chair for 2020.

**A chance to learn, equally, from people on the forefront of software development, those fighting daily fires in mature code-bases and the few with incompatible views who always open up your mind**

## From Stefan Turalski

This year's ACCU 2019 conference was my first ACCU gathering since the conference moved to Bristol. I had really missed the unique vibe and the opportunity to grow through thought-provoking conversations. A chance to learn, equally, from people on the forefront of software development, those who are fighting daily fires in mature code-bases and those few with incompatible views who always open up your mind. Therefore, when an opportunity arose to leave the office for a few days of 'training', I knew instantly where I must be headed. It actually helped that ACCU publishes its sessions, as having watched these dutifully for the last couple years, I could easily secure sign-off. (Un-)Surprisingly, I haven't got much of a problem convincing higher-ups that ACCU is still the best place to: catch-up on recent developments in C++ (obviously), identify the maturity of alternatives (with Swift and Rust under scrutiny this year), discover new tools (or things I'd never expect are possible with tools I've used for years) and learn; learn tons!

Now, as the dust settles and ACCU 2019 is a thing of past, I can only report that I was evidently correct. Seeing so many familiar faces around, I felt at home almost instantly. By the second day, I'd fulfilled my goals and – with my mind blown – I can safely spend another year catching-up. It was bliss.

Going into details, by now I've got a chance to re-watch some of the sessions and I think I can recommend a few for your attention. Let me start from these I'd say are 'must':

- Herb Sutter on 'De-fragmenting C++: Making exceptions more affordable and usable', with Herb stirring up some debates with a few ideas that are captured in P0709 R2 [Sutter18]. As far as I know, it's the first time Herb has presented this argument. If you find time to watch just one talk, I guess it would be this one. It's a must if you want to know in which direction the discussion about error-handling in C++ is headed. For a pragmatic approach and a healthy, balanced view on the subject, I'd definitely recommend Phil Nash's 'The Dawn of A New Error'.
- Stephen Kelly on 'Extending clang-tidy in the Present and in the Future'. Stephen is the person who contributes to CMake and now clang-tidy, clang-query; I watched 'Refactor your codebase with Clang tooling' from code::dive, but I probably wasn't paying enough attention. I recommend it for anyone who has never heard about clang-tidy and clang-query. It's amazing what can be done with these tools.
- Greg Law on 'More GDB wizardry and 8 other essential Linux application debugging tools', which I missed during the conference (as I went for Anthony Williams' talk on callbacks). Watching it now, I can only say that Greg's talks are getting better and better, and as far as I know Greg is the gdb guru (god?). By the way, Greg started up undo.io. Sadly not open-source, but clearly a great tool for recording what your system is/was doing. Chatting with some Microsoft guys, it seems they are trying to provide something similar [Microsoft], but that's behind a VS Enterprise licence and it still doesn't work nowhere near as smoothly. Regardless, if you ever run gdb, you need to see Greg's session.

As always there were a few talks that are worth checking out to keep up to date and learn how others are solving theirs problems. In this category I'd put:

- Björn Fahller on 'Programming with Contracts in C++20', as there is a version available from the new-comer conference *Cpp on Sea*. During ACCU 2019, I was sitting in the next room (where the talk on C++ package management concluded with a recommendation of conan for dependencies management). However, the noise next door was so inspiring that I've watched Björn's talk, twice. I'd say it's definitely worth the time put into it: it's about contracts after all.
- Arne Mertz on 'Clean(er) Code for Large Scale Legacy Applications' was one of the talks that I missed (as I've seen a year-old version from *Meeting C++* and I went to an entertaining, as always, talk by Kevlin Henney instead). Corridor chats recommended it to me and indeed it's definitely worth a watch, especially if you are looking for ideas to curb a code-base getting a little out-of-hand.
- John Lakos on 'Allocator-Aware (AA) Software', which I'd say watch at ×2 speed, go have a drink, watch again… actually no, this talk is one of the most approachable of the John Lakos presentations I've seen so far. Allocators are coming to C++, so I guess there is no excuse for ignoring the subject.
- Follow that up with Vittorio Romeo on 'Higher-order functions and `function_ref`,', which I've seen recorded at *C++ on Sea*, and which works perfectly in tandem with Ivan Čukić on 'Ranges for distributed and asynchronous systems'. Ivan wrote *Functional Programming in C++* and clearly demonstrated a set of rather interesting pattern(s). Such code styles, applied together with approach outlined by Vittorio, might leave you with a rather particular, functional?, style of C++: you were warned.

There are also the talks which you would probably see because these belong to your area of interest. Here I'd put

- Hubert Matthews on 'Optimising a small real-world C++ application', which I've seen recorded at *NDC*. ACCU 2019 received a slightly improved version (and questions were, of course, on much higher level). On the subject of questions, one could easily justify going for the ACCU conference just for the Q&A. These are always brilliant!
- Felix Petriconi (taking over the organisation of the ACCU conference from Russel) on 'An Adventure in Race Conditions', talking about concurrency [Parent] and executors, handling of `std::future` etc
- Patricia Aas' interesting talk on 'The Anatomy of an Exploit'.

I'm sure you would find something for yourself in this category among the overwhelming number of ACCU 2019 talks published on YouTube only a few weeks post-conference.

I'm in love with the concept of technical talks. I find them to be the most effective learning method for me. The opportunity to meet a lot of tech enthusiasts is both fun and enriching

Finally, there are the best talks, the esoteric ones, like:

- Simon Brand and Peter Bindels on 'Hello World from Scratch', which you know you won't find useful, until you do (it's just a crazy talk about all that happens before the hello world is actually printed).
- Alisdair Meredith on 'How C++20 Can Simplify `std::tuple`', which – as far as I'm aware – Alisdair presents at every C++ Standard revision to demonstrate why things are changing and what's possible.
- Jim Hague on 'It's DNS, Jim, but not as we know it', which introduced me to the changing world of DNS. Here, please make sure you know what DNS over HTTPS is. It's all going to be very confusing when browsers switch to it.
- Last (but definitely not least), I recall Roger Orr on his adventures in 'Windows Native API'.

I'm sure that everyone who has picked up this edition of *Overload* is more than capable of building their own list of best-picks from ACCU 2019. I'm looking forward to lively discussions in YouTube comments – hopefully until ACCU 2020! Happy watching.

## From Ori Ben-Shir

First published on Ori's blog 'Afternoon Rusting' on 20 April 2019: https://oribenshir.github.io/afternoon_rusting/blog/ACCU-Summary

I attended this year's ACCU conference, and I am very eager to share my impression of the conference. ACCU is an annual conference located in the lovely city of Bristol. The conference is mostly dedicated to C++ developers. While C++ developers are the focus, the conference is not limited to C++ material, and it includes talks for various topics and even some other programming languages. Yes, there was a Rust talk and even a workshop this year.

It was the first time I had attended a big conference. And I must admit it was a great pleasure! I'm in love with the concept of technical talks. I find them to be the most effective learning method for me. The opportunity to meet a lot of tech enthusiasts is both fun and enriching. Wrapping it all with a vacation in such a lovely city such as Bristol is immensely satisfying. If you have the opportunity, I encourage you to attend this conference next year. I also think the organizers did a great job. I genuinely like the extra social session. The pub quiz, for instance, was perfect, though some of the code samples from it were as far from perfect as possible.

I have a lot to say about the content itself. I tend to believe I have more to say than you want to read. So let's focus on some of the talks I think are most relevant:

### C++ ranges and functional programming

The first session I want to discuss is about ranges in C++. Ivan Čukić gave an incredible talk. He demonstrated some splendid functional programming with C++. Ranges are actually quite simple. It is a struct containing an iterator and a sentinel value which mark where the iteration should stop. As a concept, we can already use it today, although it is planned to be a part of the standard library in C++ 20. While the idea is simple, it provides us with the capability to implement a very complex functional system, which was very enjoyable to see. I was impressed with the pipeline he demonstrated, and how flexible it is. It can support both async and sync programming. Even more impressive, he managed to introduce a process or even a machine boundary in the middle of the pipeline. A point I'm still not sure about yet is how simple it is to create an entirely lazy iterator. Meaning, we want to pay the price of computation only when the pipeline is actually being executed, and not during its declaration.

### C++ error handling

The second talk by Herb Sutter. He discussed a new proposal for error handling. Today C++ error handling is painful, there aren't any real best practices, and the community is greatly divided by various methods, which does not play nicely with each other. Making it one of the reason it is so hard to integrate libraries in C++. This issue alone is one of the major reasons I wanted to investigate Rust. The talk has shown a new suggestion Herb is working on making exception useful. I enjoyed hearing the exact points that bother me so much about C++ error handling today:

1. Lack of conformity: Some use exception, some use types (similar to Rust 'Error' Type), and many still use plain old integer with error code (and out parameters for the real output blah!).
2. Too many errors: Today, too many of the errors in C++ are not actual errors. Some errors should be caught by the compiler (e.g., out of boundary, lifetime issues), others should panic instead (failing to allocate an `int` with the global allocator). Some of those issues will be solved with the contract feature of C++.
3. Lack of visibility of exception: Neglecting all other problems with exception (and there are many of them) exceptions suffers from a severe lack of visibility. It is not always obvious what can go wrong when integrating a new code. And it might be tough to handle all (mostly invisible) error flows.
4. Performance – Exception today introduces performance penalty (Even if not being used).

The second and third points integrate poorly together. A program in C++, even in modern C++, suffers from an extreme number of hidden code path, representing error states, which just shouldn't be there. These code path can't be tested and usually are invisible to the programmer!

Herb's suggestion was very interesting. I think it might actually work. First, he wants us to be explicit when a function can throw, with the `throws` keyword. Second, he wants to allow easy propagation of errors with the `try` keyword. He also wants to improve the performance of exceptions by making them statically allocated, and caught by value. Seeing the full suggestions, it looks very similar to Rust error handling. I assume that unlike Rust, the compiler won't force us to handle the error case. Due to backward compatibility, I don't think it would be mandatory to state if a function returns an error. The only hope is that those two features will be integrated into static analyzers, like clang tidy. Which is far from optimal, yet I think it can work, and finally allow a reasonable error handling for C++.

It feels like the community is more and more open to breaking backward compatibility in order to simplify C++ and increase its safety

### Monotron

The last talk for this post, given by Jonathan Pallant. It was about Rust, embedded Rust to be precise. The talk was about his monotron project, a simple 1980's home computer style application [Pallant]. It involved two of my favorites topic: Rust and system programming. The talk started with the state of embedded rust, evidently making considerable strides to maturity. And it continues with his effort to add more and more features to a very limited hardware. A small spoiler, the results are amazing. This project is the ultimate proof of power for Rust. It demonstrates that Rust can be as fast as the hardware it runs on allows it to be. The kind of property any C++ developer looks when he considers an alternative programming language.

### On Rust & C++

I must admit this conference set me thinking about the interaction between C++ and Rust. Yes, modern C++ is not news anymore, and I don't believe it emerged because of Rust. I do think some Rust ideas manage to trickle into C++, yet I believe the actual impact on code abstraction itself is small. But C++ is going through additional change, one of mentality. It seems the community finally understands that, even with the right abstraction, a 1000-pages book of best practices just won't do. We are human after all, as the wonderful Kate Gregory reminded us. It feels like the community is more and more open to breaking backward compatibility in order to simplify C++ and increase its safety. And weirdly, I think it is managing to find a way to break backward compatibility without breaking it. It seems like a Turing complete compile-time abstraction, together with a configurable compiler, is the answer. A very complicated answer to be sure, yet one where the average developer doesn't need to be aware of its details. To sum it up, I have the feeling that the question: 'Should I write my new project in C++ or Rust?', is becoming more and more relevant every day. And the answer is getting more and more complex.

### See the talks!

One last thing, the talks from the conference are uploaded to YouTube. Strongly recommended!

## From Mathieu Ropert

First published at https://mropert.github.io/2019/04/19/accu_2019/

This year I was ready. I had prepared a stock of jokes about Britain, its food, its weather, the absence of good wine and the tumultuous relationship with the EU. It was time for ACCU 2019.

This year's edition of the ACCU conference was held from April 10th to April 13th, in Bristol as always. I arrived a day earlier from Paris after a short stop in France, which was supposed to offer a supply of good weather and trips to a few winemakers in preparation for the harsh conditions of Great Britain.

From the start, things went awry as I could only spare half an hour for a visit to a winemaker in Vouvray who turned out to be quite forgettable, not to mention the weather that was only barely keeping it together. Still, I didn't immediately notice that something was off, having spent the pasts months enduring the cold winter of Sweden. It took a second flight from Paris to Bristol to realize it: spring is there (although a couple of Bristol locals apologized for the weather being unexpectedly non-terrible).

### Many meetings

My arrival was pretty unremarkable. It was, of course, raining and people still drove on the wrong side of the road. I had come across my former colleague Jonathan Boccara (of *Fluent C++* fame [Boccara]) while waiting at my gate. We traded some war stories and he told me about his book, which he would be showing at the conference. I haven't had the time to read it yet but I have already heard some positive feedback about it.

ACCU is, like most conferences, a good time for me to spend some time face-to-face with friends from the C++ community living around the world. It is sometimes said that there is more value in the discussions with the people you meet at conferences than with the conference content, and I would partly agree. Depending on the circumstances, I do feel like the bulk of the value falls slightly one side or the other. At times there's a presentation that justifies the whole ticket in itself; sometimes I meet someone and have a discussion that is as valuable to me as the sum of all the talks I attend.

The other reason I often see so many familiar faces is that, in my opinion, people don't try to attend conferences nearly enough. After asking around a bit, it does seem like I'm not the only one to have noticed that. Regardless of the company, there will be a small minority who ask their manager to be sent there, and a large majority who will never do so. I am not sure how to explain it. Not feeling like it's worth the time? Thinking it's only for some 'elite'? Maybe simply too focused on the day-to-day job, on the next deadline?

I don't claim to have the answer, but I will certainly encourage anyone who never asks to go to do so, and those who do to encourage their colleagues to do the same. We are always happy to see new faces, meet new people and buy them a drink at the conference bar at the end of the day.

### Keynotes

As I mentioned in my trip report last year [Ropert18], I was a bit disappointed by ACCU 2018's opening keynote. This time was quite the opposite. ACCU 2019 opened with M. Angela Sasse telling us about security. The key takeaway was that the human, the user, will always be the weak link regardless of the technology deployed. More importantly, the fact that security is everyone's business and not just the IT department's means it must offer a good UX else it will be badly used or worked around. This was pointed out in the 90s and it still hold true today, with sadly little progress to show for it.

The great Herb Sutter travelled from his Redmond office to England to tell us about his vision for the future of error handling in C++. While I already knew about his work on the matter (it was sent to SG14 for review a couple months back), it was nice to have a refresher in front of the whole conference. In short, the direction is toward better exceptions, with bounded, predictable throw and catch times. No more need for dynamic allocation. No extra cost when no exceptions occur (this is already mostly

the case on x86_64) and a push for `noexcept` becoming the default unless otherwise specified.

The closing keynote was given by none other than Kate Gregory, who walked us through a nice lecture on code empathy, or how to read the previous programmer's emotions through existing code, trying to understand what triggered those emotions and how to react when confronted by them. I have a hunch that it will be a nice complement to Jonathan's book on how to deal with legacy code, as the two seem closely related.

### Talks

At the rate of there talks a day outside keynotes, there was a total of 12 I could potentially attend during the conference. Subtract one because I had to attend mine and perhaps another one where I was busy writing slides and we get a rough estimate of about 10. While that number could make for a nice clickbait section ('10 ACCU talks you won't believe I attended'), I will stick to my boring routine of mentioning the ones I remember the most. Also keep in mind that there were 5 tracks, meaning I saw roughly 17% of the conference content.

The two talks that made the biggest impression on me were Vittorio Romeo's 'Higher-order functions and `function_ref`' and Andreas Weiss' 'Taming Dynamic Memory – An Introduction to Custom Allocators'. The first one did a good job of explaining what higher-order functions are and also the content and benefits of the `function_ref` proposed addition to the C++ standard, all in one session. The second one offered a good tour of custom allocators, how they work and when they can be considered to replace the standard ones. Both presenters also had to accomplish their tasks while fending off the many questions coming from John Lakos, who sat on the first row each time (a victory he congratulated them for at the end).

The next two talks that come to mind are Peter Bindel's and Simon Brand's 'Hello world from scratch', and Andy Balaam and CB Bailey's 'How does git actually work?'. Both explained things we do every day by taking a very simple use case (building a very simple program and committing some changes to a VCS) and showing what happens under the hood. They also both ran out of time before showing all they had planned because it turns out abstraction is no myth: even our simplest tasks are actually fairly complex when you look at how they are done. I think they both did a good job of it and would gladly schedule both in a 'how does XXX work' track. That is a good theme that I would suggest having at every conference.

Next up is Kevlin Henney's 'What do you mean?'. Kevlin is quite the celebrity in Bristol and I really liked his talk at the previous conference. While perhaps not as remarkable (I would have appreciated a clearer outline to follow), this one was still quite interesting. The main point was that meaning is derived from both what is said or written and the context that surrounds it. The latter being subjective, it implies a bunch of assumptions by both parties that, when not in line, lead to quite a misunderstanding. The main obstacle to solving that problem is that assumptions are, by definition, implicit and so can only be discovered when proved wrong ('Oh, but I assumed that…'). This of course brings us back to the software craftsmanship practices of frequent iterative deliveries and testing.

Finally I'd like to mention Christopher Di Bella's 'How to Teach C++ and Influence a Generation'. Last year, Christopher started SG20, a study group in the standard committee focused on education. Education and teaching is an important subject to me, partly because of my own personal experience of learning C++ in school, then learning another language also called C++ around 5–10 years later. As you may guess, the first one was more in the line of 'C with classes' while the second looked more like the C++ we know and recommend today. To that end, the group has worked on some guidelines on how to write teaching materials. They also run polls to better understand how and when people learn C++. A good complement to this talk would be Bjarne's keynote at *CppCon 2017* Learning and Teaching Modern C++.

### Lightning talks

One of the best things at ACCU is how the lightning talks sessions are organized. They are simply done in the keynote room as the closing session of each day. That way, most of the conference attends before going out for beers or dinner. The programme is usually decided between the day before and a couple hours before the session, meaning last minute entries are definitely an option.

It's a great opportunity to bring up a point you had in mind but couldn't get in as a talk, respond to a previous talk (or lightning talk) or simply raise awareness in the community on a particular matter. For example, upon arriving in Bristol on Tuesday I learnt that the great people from the Paris meetup were planning to announce a new conference [CPPP]. I put a few slides together, slipped in a joke or two about English food and Brexit, then went up on stage on Wednesday to tell everyone about *CPPP*.

Of all the C++ conferences I go to, I think this formula works best and is one of the reasons ACCU feels like a big family gathering. If you are a conference organizer and have some lightning talk sessions, I strongly suggest you consider this option. It might feel intimidating to step up on stage in front the entire conference, but then again, I feel the familial atmosphere helps in reducing the pressure.

### Until next time

On Friday I gave my talk, 'The State of Package Management in C++'. Frequent readers of my blog will probably be familiar with the topic. I gave a tour of package management in C++, why we want it and how far we've come yet (spoiler warning: enough for you to try it). As you can see, the ACCU has made a fantastic job of uploading the recording on YouTube in less than a week.

But the greatest learning of all for me came after the conference, when I discovered that airlines will now charge you £50 when boarding your plane for bringing a laptop bag with your carry-on luggage. I used to do that all the time, but today it appears you can be charged extra depending on the mood. I suppose next time I will have to put my stuff in cargo ☹.

Do not let that stop you from attending conferences though, I still hope to see you there!

## From Anthony Williams

First published on Anthony's 'Just Software Solutions' blog on 22 April 2019 at https://www.justsoftwaresolutions.co.uk/news/accu-2019-report.html.

I attended ACCU 2019 a couple of weeks ago, where I was presenting my session 'Here's my number; call me, maybe'. Callbacks in a multithreaded world.

The conference proper started on Wednesday, after a day of pre-conference workshops on the Tuesday, and continued until Saturday. I was only there Wednesday to Friday.

### Wednesday

I didn't arrive until Wednesday lunchtime, so I missed the first keynote and morning sessions. I did, however get to see Ivan Čukić presenting his session on 'Ranges for distributed and asynchronous systems'. This was an interesting talk that covered similar ground to things I've thought about before. It was good to see Ivan's take, and think about how it differed to mine. It was also good to see how modern C++ techniques can produce simpler code than I had when I thought about this a few years ago. Ivan's approach is a clean design for pipelined tasks that allows implicit parallelism.

After the break, I went to Gail Ollis's presentation and workshop on 'Helping Developers to Help Each Other'. Gail shared some of her research into how developers feel about various aspects of software development, from the behaviour of others to the code that they write. She then got us to try one of the exercises she talked about in small groups. By picking developer behaviours from the cards she provided to each group, and telling stories about how that behaviour has affected us, either positively or negatively, we can share our experiences, and learn from each other.

### Thursday

First up on Thursday was Herb Sutter's keynote: 'De-fragmenting C++: Making exceptions more affordable and usable'. Herb was eloquent, as always, talking about his idea for making exceptions in C++ lower cost, so that they can be used in all projects: a significant number of projects currently ban exceptions from at least some of their code. I think this is a worthwhile aim, and hope to see something like Herb's ideas get accepted for C++ in a future standard.

Next up was my session, 'Here's my number; call me, maybe. Callbacks in a multithreaded world'. It was well attended, with interesting questions from the audience. My slides are available [Williams19] and the video is available on youtube. Several people came up to me later in the conference to say that they had enjoyed my talk, and that they thought it would be useful for them in their work, which pleased me no end: this is what I always hope to achieve from my presentations.

Thursday lunchtime was taken up with book signings. I was one of four authors of recently published programming books set up in the conservatory area of the hotel to sell copies of our books, and sign books for people. I sold plenty, and signed more, which was great.

Kate Gregory's talk on 'What Do We Mean When We Say Nothing At All?' was after lunch. She discussed the various places in C++ where we can choose to specify something (such as `const`, `virtual`, or `explicit`), but we don't have to. Can we interpret meaning from the lack of an annotation? If your codebase uses override everywhere, except in one place, is that an accidental omission, or is it a flag to say 'this isn't actually an override of the base class function'? Is it a good or bad idea to omit the names of unused parameters? There was a lot to think about with this talk, but the key takeaway for me is 'Consistency is Key': if you are consistent in your use of optional annotations, then deviation from your usual pattern can convey meaning to the reader, whereas if you are inconsistent then the reader cannot infer anything.

The final session I attended on Thursday was the 'C++ Pub Quiz', which was hosted by Felix Petriconi. The presented code was intended to confuse, and elicit exclamations of 'WTF!', and succeeded on both counts. However, it was fun as ever, helped by the free drinks, and the fact that my team 'Ungarian Notation' were the eventual winners.

### Friday

Friday was the last day of the conference for me (though there the conference had another full day on Saturday). It started with Paul Grenyer's keynote on the trials and tribulations of trying to form a 'community' for developers in Norwich, with meet-ups and conferences. Paul managed to be entertaining, but having followed Paul's blog for a few years, there wasn't anything that was new to me.

'Interactive C++: Meet Jupyter / Cling – The data scientist's geeky younger sibling' was the next session I attended, presented by Neil Horlock. This was an interesting session about cling, a C++ interpreter, complete with a REPL, and how this can be combined with Jupyter notebooks to create a wiki with embedded code that you can edit and run. Support for various libraries allows to write code to plot graphs and maps and things, and have the graphs appear right there in the web page immediately. This is an incredibly powerful tool, and I had discussions with people afterwards about how this could be used both as an educational tool, and for 'live' documentation and customer-facing tests: 'here is sample code, try it out right now' is an incredibly powerful thing to be able to say.

After lunch I went to see Andreas Weis talk about 'Taming Dynamic Memory – An Introduction to Custom Allocators'. This was a good introduction to various simple allocators, along with how and why you might use them in your C++ code. With John Lakos in the front row, Andreas had to field many questions. I had hoped for more depth, but I thought the material was well-paced, and so there wouldn't have been time; that would have been quite a different presentation, and less of an 'introduction'.

The final session I attended was 'Elsewhere Memory' by Niall Douglas. Niall talked about the C++ object model, and how that can cause difficulties for code that wants to serialize the binary representation of objects to disk, or over the network, or wants to directly share memory with another process. Niall is working on a standardization proposal which would allow creating objects 'fully formed' from a binary representation, without running a constructor, and would allow terminating the lifetime of an object without running its destructor. This is a difficult area as it interacts with compilers' alias analysis and the normal deterministic lifetime rules. However, this is an area where people currently do have 'working' code that violates the strict lifetime rules of the standard, so it would be good to have a way of making such code standards-conforming.

### Between the sessions

The sessions at a conference at ACCU are great, and I always enjoy attending them, and often learn things. However, you can often watch these on Youtube later. One of the best parts of physically attending a conference is the discussions had in person before and after the sessions. It is always great to chat to people in person who you primarily converse with via email, and it is exciting to meet new people.

The conference tries to encourage attendees to be open to new people joining discussions with the 'Pacman rule' – don't form a closed circle when having a discussion, but leave a space for someone to join. This seemed to work well in practice.

I always have a great time at ACCU conferences, and this one was no different. ■

## References

[Asproni19] Giovanni Asproni, 'One Team, Two Teams, Many Teams: Scaling Up Done Right', https://www.meetup.com/Agile-Bath-Bristol/events/260202604/

[Boccara] Jonathan Boccara, *Fluent {C++}*, https://www.fluentcpp.com/

[CPPP] CPPP Conference, 15 June 2019, Paris: https://cppp.fr/

[Davis19] 'The Pac-Man Rule': https://conference.accu.org/pacman_rule.html

[Echoborg] Echoborg: http://www.echoborg.com/

[Microsoft] 'Introducing Time Travel Debugging for Visual Studio Enterprise 2019', https://devblogs.microsoft.com/visualstudio/introducing-time-travel-debugging-for-visual-studio-enterprise-2019/

[Pallant] Jonathan Pallant, 'Monotron', https://github.com/thejpster/monotron

[Parent] Sean Parent, Foster Brereton and Felix Petriconi (date unknown), 'Concurrency' on the *stlab* website: http://stlab.cc/libraries/concurrency/

[Ropert18] Mathieu Ropert, 'ACCU 2018 trip report', posted 20 Apr 2018 at https://mropert.github.io/2018/04/20/accu_2018/

[Sutter18] Herb Sutter, 'Zero-overhead deterministic exceptions: Throwing values', P0709 R2, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0709r2.pdf

[Williams19] Anthony Williams, 'Here's my number; call me, maybe. Callbacks in a multithreaded world' (slides), https://www.justsoftwaresolutions.co.uk/files/heres_my_number.pdf

Although different to the type of article we normally publish in *Overload*, we hope you have found these reviews of the ACCU 2019 Conference interesting.

# Afterwood

There are parallels between writing and programming. Chris Oldwood shares his journey into learning to write well.

Ten years ago today (as I write) I created a blog and published my first post. It was to be the start of a new chapter in my career as a programmer, spurred on by my fairly recent introduction to the world of ACCU which itself came about through lamenting the demise of the major printed programming journals on the company's chat tool.

My decision to begin writing was not an easy one, though. For a start, my English exam results as a teenager were somewhat abysmal; so bad, in fact, the exam board wouldn't even give me a grade (a 'U' for those familiar with the '80s English school system)! I eventually passed my English Language exam, but not without a struggle. I thought developing software would have little to do with 'proper' writing and so it wouldn't matter in the long run; I couldn't have been more wrong.

Aside from the act of writing itself, which I had accidentally started to do anyway through some lengthy diatribes dressed up as emails about the state of the architecture and codebase I was working on at the time, the other major concern I had was around originality. What was I going to write about? If there was something I had learned from all my time spent reading up to that point, it was that so much had already been said, what could I possibly say that was new? In particular I'd only really worked in 'the Enterprise' which is hardly a breeding ground for cool and exciting new inventions in the world of software. If that wasn't enough, *StackOverflow* had recently taken off and was fast becoming a major source of knowledge that looked to be the nail in the coffin for many shorter topics which seemed to be a nice way to ease oneself into the writing process.

What I came to discover was that neither of these concerns were really anything that I should have been quite so worried about. Writing, much like programming, is something which you can only get better at by doing. What had made those emails particularly hard to write was getting the tone right so that they framed the problems in a light that was positive rather than simply sounding like an empty rant. I wanted the problems to be acknowledged and to inspire my team to think about how we could improve matters going forward. Consequently, I found myself continually reading back what I'd read, and then re-writing bits, again and again until I felt I had finally expressed myself in a way that I hoped was somewhat inspirational. I don't know why it took me so long to recognize that what I was already doing with code – continually reviewing and refactoring to best express the solution – was perfectly normal when writing prose too. And naturally as I got better I found myself triangulating towards a piece I became happy with much sooner and therefore the experience became more enjoyable as a result.

It's probably no surprise that the more you do of both – programming and writing – the more interesting parallels you discover between them because, after all they're both about languages with rules where ambiguity can have unfortunate effects. As such, I've found an unexpected feedback loop developing, where my interest in programming languages has generated an interest in natural languages too, where for a long time there was only really disdain. This in turn has generated a degree of confidence that has caused me to consider being more adventurous in my writing style.

Looking at the subject from the content perspective, what I've found is that originality is essentially a moot point because as an industry we seem to spend a considerable amount of time rediscovering ideas and concepts from the past. In part I suspect that is because there is still a considerable gap between the theory and practice of programming due to the imperfections and limitations of the tools we use. Hence there is a continual need to bridge the theoretical and practical worlds by framing the discussion in the context of different toolsets and problem domains as this helps the message to get though to different people. Coupled with the imperfections and limitations in our writing, this means any given topic can, and must, be said in a number of different voices to help others relate to what we are saying. Just as one size does not fit all, neither does one explanation, which is exemplified by the number of people who have attempted to explain what a Monad is.

There is also a heavy bias in the industry to focus on what the 'cool kids' are doing in places like Silicon Valley despite the fact that the large majority of jobbing programmers are not involved in greenfield work. While it is useful and interesting to know what new stuff looms on the horizon, I also want to know how some this can be applied under the tight constraints of the aging brownfield codebases which my day job entails. A lack of knowledge sharing from those in older establishments is less surprising when you understand what constraints they apply around protecting intellectual property despite the fact that what many of them are producing is anything but rocket science. As such, I've become happy to help populate the body of technical literature which helps document the experiences of how one person has tried to apply their ever growing tree of knowledge to what might be considered by some to be the less glamorous world of the maintenance programmer.

My decision to start recording my observations and practice using the written word has undoubtedly been fulfilling. At the very least, I have been able to refer back to my own notes when the same issue turns up again, but occasionally I have also been fortunate enough to learn that other people have benefited too, and that alone still makes it all worthwhile. ■

**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

# JOIN THE ACCU!

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

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

With full ACCU membership you get:

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

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

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


### How to join

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

### Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

intel Software

# CODE MAXIMIZED


from £510

#HighPerformance

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



---

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

To find out more about Intel products please contact us:
020 8733 7101 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE