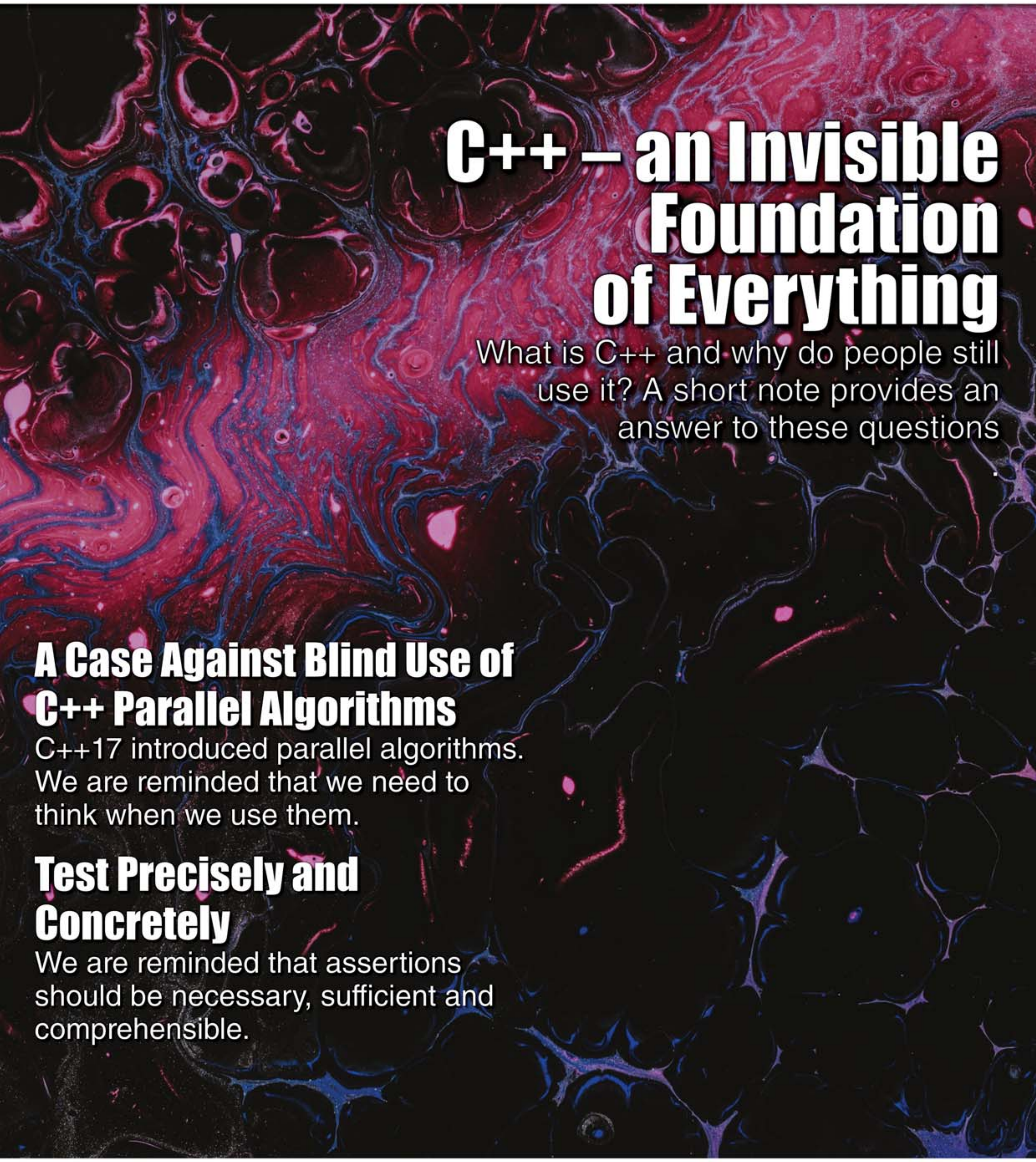
overload 161

FEBRUARY 2021 £4.50



C++ – an Invisible Foundation of Everything

What is C++ and why do people still use it? A short note provides an answer to these questions

A Case Against Blind Use of C++ Parallel Algorithms

C++17 introduced parallel algorithms. We are reminded that we need to think when we use them.

Test Precisely and Concretely

We are reminded that assertions should be necessary, sufficient and comprehensible.

A magazine of ACCU

ISSN: 1354-3172

JET
BRAINS

A Power Language Needs Power Tools

Smart editor with full language support
Support for C++03/C++11, Boost and libc++, C++ templates and macros.

Reliable refactorings
Rename, Extract Function / Constant / Variable, Change Signature, & more

Code generation and navigation
Generate menu, Find context usages, Go to Symbol, and more

Profound code analysis
On-the-fly analysis with Quick-fixes & dozens of smart checks

GET A C++ DEVELOPMENT TOOL THAT YOU DESERVE

R++

ReSharper C++
Visual Studio Extension for C++ developers

AC

AppCode
IDE for iOS and OS X development

CL

CLion
Cross-platform IDE for C and C++ developers

Start a free 30-day trial
jb.gg/cpp-accu

Find out more at **www.qbssoftware.com**

QBS
SOFTWARE

OVERLOAD 161

February 2021

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

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

Photo by Paweł Czerwinski on Unsplash

Copy deadlines

All articles intended for publication in Overload 162 should be submitted by 1st March 2021 and those for Overload 163 by 1st May 2021.

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

The ACCU

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

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

In. Sub. Ordinate.

Mindless rebellion is mindless. Frances Buontempo encourages mindful consideration of when to refuse.

Yet again no proper editorial from me. I just won't do the thing. Oh well. Not everyone does what they are told. Whether we're talking rebellious children, defiant pensioners, eco-warriors or the population at large picking and choosing what advice they are willing to follow, everyone is insubordinate from time to time. Why do we obey any laws or suggestions? Many philosophers have asked this question, including Hobbes, Locke and Kant. They suggest some form of social contract, whereby people give up freedoms either for absolute government avoiding the prospect of anarchy or less extremely than Hobbes' view, increasing the chance of respect and a quieter life [Wikipedia-1] What would happen if all laws were abolished is an interesting, but ultimately unanswerable, question. Laws are laid down, and often, though not always, obeyed.

Some laws are more observations than laws. Moore's 'law' springs to mind. His observation that the number of transistors tended to double every two years captures a trend in data which seems unlikely to continue forever [Moore]. There are many similar observations in computing, regarding storage and so on. These are not laws. They are neither enforced nor does anyone run the risk of incarceration if they disobey. On the other hand, many outfits have coding 'guidelines' which are more like actual rules, either officiously enforced or automatically applied. I recently set up the Python formatter Black [Black] on CI for a new repo because our coding guidelines say 'We use Black'. It describes itself as an uncompromising formatter, so that "Formatting becomes transparent after a while and you can focus on the content instead." What has in fact happened is a whole slew of commits with messages like "Black, oh why? Oh why?" or "Black. Again. Grr". If I change the CI step from just checking the formatting to reformatting, or set up pre-commit hooks, that would avoid the agro. Black could then silently do its thing with whitespace so we can concentrate on the code in between the spaces. Or I could remove the step from the CI build in a small act of rebellion. Little victories.

The UK had all kinds of guidelines and rules to attempt to keep us safe from COVID-19 over the Christmas holidays. How obedient was your break, if you had one? Did you stay indoors, video conference your family, say hello to neighbours from two metres away and above all costs only sing if you were in a choir, or alone in a shower? Our mixture of rules and guidelines are somewhat confusing and prone to last minute change. Some are using this as an excuse for disobedience. Many are trying their best, though under a little duress. Where the rules seem to make no sense, it's useful to be able to discuss why and what the alternatives might be. With difficult political situations that's not always possible. Having a discussion about code format and formatters seems likely. Asking our parliament why they issue the guidance they have seems less so.

Now, insubordination and disobedience differ. The former involves a refusal to be lorded over, rejecting submission to a supposed higher authority. To disobey, in contrast, is a more direct refusal to comply with a given instruction. The diktat is snubbed not the dictator. It is possible to do exactly as you have been told and, while not disobeying, you can nonetheless be insubordinate. Your computer may well obey you, but may appear to be attempting some kind of insurrection or at least insubordination at times. Upon the bash instruction `echo variable` the defiant machine will echo `variable` verbatim to the screen, rather than the contents of the variable, since you forgot the dollar sign. Flip. It could be worse though; sometimes computers try to guess what you really meant. Don't get me started on autocorrect. Undefined behavior is neither insubordinate nor disobedient. Walking off the end of an array allows your compiler to do whatever it feels like, if you believe computers have feelings. Obedience isn't always helpful.

Furthermore, doing exactly what you are told can cause all kinds of trouble. The phrase 'work to rule' springs to mind. Rather than striking, employees may take to doing exactly what the rules say, as a "form of industrial action where the employee will follow the rules and hours of their workplace exactly in order to reduce their efficiency and output." [Voice] So often, extra voluntary duties keep the wheels moving. By calling this form of action work to rule, the degree of spikiness is obvious, pointing out that much of the day to day work involved is above and beyond written and signed off contracts. Let's avoid devolving into discussions of workers' rights: that strays far too close to me writing opinions and puts this in danger of becoming an editorial. Totally unacceptable.

Doing exactly what you are told can get you into trouble, even if you are really not trying to be insubordinate. My special superpower is following instructions, often to others' bemusement, with unexpected consequences. This probably means I should go into software testing when I grow up. Have you ever read instructions and followed them precisely? Try it on a document you have written, or a recipe or DIY instructions. Don't do what you think you wrote, do what you actually wrote, or at least consider what that might involve. Writing clear instructions is a hard technical job that few of us are any good at. Though I could relay many personal stories, I shall stick to one. For a practical science exam, the first instruction was to find the volume of a cylinder. To my mind, the capacity was the amount of volume of a liquid I could fit inside the cylinder, so it stood to reason that the volume of the aforementioned cylinder was precisely that. I therefore set about trying to find how much volume the walls of the cylinder would displace, much to the consternation of my teacher. Having noticed her facial expression, I read the subsequent instructions and concluded I would need to know the capacity of the container in order to proceed. Lesson learnt. The instructions are usually wrong – read them all first to decide how insubordinate you need to be to accomplish your mission. How many

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.

times have I followed steps one at a time, to find the next sentence says don't carry out the last step under some specific circumstances? If you write up instructions please, please, please put the instructions inside an "If" so Fran doesn't break stuff. Glancing ahead, and wondering "Why am I being told to do this?" helps one to discern the actual instructions or indeed alternatives. A small degree of disobedience might genuinely be in the best interests of all involved: no insubordination intended.

Trying the software you have written and discovering what happens when you stray off the 'happy path' can be illuminating too. What happens if you press the same button twice? What happens if you don't enter your date of birth? If you don't have the imagination, try sitting with someone else using your creation. No matter how obvious or clear you think something is, there is always a chance someone else may have different ideas. In fact, you sometimes look back at notes you have previously made and either can't read your own hand-writing or have no idea what you were driving at originally. Sometimes things don't pan out, despite an attempt to follow instructions.

Following instructions can even cause no end of trouble. There's a long standing tension around the subject of tax avoidance and tax evasion. I have to concentrate on which is which. One involves obedience, sticking to the law, doing as instructed and thereby paying less tax than you would have had you not stopped to think through which numbers to put in which boxes or paid an accountant enough to put the right numbers in the right boxes. The other involves lying and not owning up to cash-flows you owe tax on. So often big tech companies have been called out for 'not paying their tax', though they seem to be sticking to the rules on the face of it. They don't seem to be money laundering, though by registering different parts of their business in different countries, they do seem to have managed to optimise their bill. Is this a crime? Probably not. Does it seem unfair? Well, maybe. Many criticisms are levelled against big tech companies, variously referred to as FAANG (Facebook, Amazon, Apple, Netflix and Google) or the Four (Horsemen (of the apocalypse implied here)), stretching far beyond questions about tax payment and accounting. 'Avoid taxes, invade privacy, and destroy jobs' according to some [CNBC]. Whatever you think, I suspect no company would have that as their mission statement. I suspect many of us have a ground state of suspicion of those who are incredibly rich. Forbes announced that Jeff Bezos became the richest person ever last year [Forbes], worth over two hundred billion dollars. I suspect it's very hard to put anyone's true worth into dollars, but you can look up stock prices and that is a mind-boggling amount of money. Does wealth imply criminality? Nope. Does criminality imply wealth? Again, nope. Are any companies disobedient or insubordinate? That might imply a company has a mind of its own, which seem unlikely. I'll leave you to decide for yourself if huge wealth leaves you feeling slightly queasy though.

Should we pay our tax? Both Matthew and Mark's gospels say "Render unto Caesar the things that are Caesar's." This may well not mean you must pay tax, since there are various ways to interpret the text. Tax revolts, including at the time of Jesus, have been common throughout history. Wikipedia offers a list of what it calls tax resistance [Wikipedia-2], pointing out it has been suggested that such resistance has caused the collapse of empires. The resistances often involve refusing to pay part of a tax which is considered unfair or even immoral. The Quakers refused to pay taxes for equipping soldiers, according to the aforementioned Wikipedia site. Many other examples are listed. Other times people do obey the law, at least to the letter, if not the spirit. At one point in British history, a tax was introduced on windows; the building rather than computing kind. You still see many bricked up windows on old houses. You can't be taxed on a window that's no longer there. Insubordination? Yep, why not. This reminds me, I really should pay my self-assessment tax shortly.

We have seen that some laws are neither legal nor scientific laws, more observations of trends over time. We have observed that some guidelines are more like laws, for example strictly enforced coding standards. I mentioned Python formatters. It amuses me that Python uses whitespace rather than braces to avoid arguments about brace placement and other layout discussions, the idea being there should be one true way so we can concentrate on code. And yet, here we are, with a choice of Python formatters. The PEP8 guidelines [PEP8] point out style guides are about consistency, assuming that a lack of consistency harms readability. The guide reminds us, "A foolish consistency is the hobgoblin of little minds." To some Hobgoblin is an ale, to others a Marvel character, but the Emerson quote refers to a fairy creature that lives in the hob or fireplace. It seems Emerson, in fact, was encouraging non-conformity [Stanford] and perhaps some form of insubordination in order to become truly self-reliant. Guidelines and rules may be there to try to keep things running smoothly and to keep us safe. Sometimes mindless obedience makes things worse, though. A spot of anarchy once in a while never hurt anyone. Rules should make life better and easier; if they don't, question them. In an article in this edition, Bjarne Stroustrup says "Design and programming are human activities; forget that and all is lost". Let's not get lost.

References

[Black] Python formatter, Black – https://github.com/psf/black

[CNBC] Four – https://www.cnbc.com/2017/10/02/scott-galloway-the-four-amazon-apple-google-facebook.html

[Forbes] Riches Billionaire – https://www.forbes.com/sites/jonathanponciano/2020/08/26/worlds-richest-billionaire-jeff-bezos-first-200-billion/

[Moore] Moore's Law – https://en.wikipedia.org/wiki/Moore%27s_law

[PEP8] PEP8 Guidelines – https://www.python.org/dev/peps/pep-0008/#a-foolish-consistency-is-the-hobgoblin-of-little-minds

[Stanford] Stanford Encyclopedia of Philosophy: Ralph Waldo Emerson – https://plato.stanford.edu/entries/emerson/

[Voice] Working to rule – https://www.voicetheunion.org.uk/working-rule

[Wikipedia-1] Social contract – https://en.wikipedia.org/wiki/Social_contract

[Wikipedia-2] Tax resistence – https://en.wikipedia.org/wiki/List_of_historical_acts_of_tax_resistance

A Case Against Blind Use of C++ Parallel Algorithms

C++17 introduced parallel algorithms. Lucian Radu Teodorescu reminds us we need to think when we use them.

We live in a multicore world. The hardware free lunch is over for about 15 years [Sutter05]. We cannot rely on hardware vendors to improve the single-core performance anymore. Thus, to gain performance with hardware evolution we need to make sure that our software runs well on multicore machines. The software industry started on a trend of incorporating more and more concurrency in the applications.

As one would expect, the C++ standard has also started to provide higher level abstractions for expressing parallelism, moving beyond simple threads and synchronisation primitives. Just for the record, I don't count **`std::future`** as a high-level concurrency primitive; it tends to encourage a non-concurrent thinking, and, moreover, its main use case almost implies thread blocking. In the 2017 version of the standard, C++ introduced the so-called *parallel algorithms*. In essence, this feature offers parallel versions of the existing STL algorithms.

This article tries to cast a critical perspective on the C++ parallel algorithms, as they were introduced in C++17, and as they are currently present in C++20. While adding parallel versions to some STL algorithms is a good thing, I argue that this is not such a big advancement as one might think. Comparing the threading implications of parallel algorithms with the claims I've made in [Teodorescu20a] and [Teodorescu20b], it seems that the C++ additions only move us half-way through.

A minimal introduction into C++ parallel algorithms

To form some context for the rest of the article without spending too much time on this, let's provide an example on how to use a parallel STL algorithm.

Let's assume that we have a **`transform`** algorithm, and we want to parallelise it. For that, one should write something like the code in Listing 1. The only difference to a classic invocation of **`transform`** is the first parameter, which, in this case, tells the algorithm to use parallelisation and vectorisation.

This parameter is called *execution policy*. It tells the algorithm the type of execution that can be used for the algorithm. In the current C++20 standard there are four of these parallel policies, as explained below:

- **`seq`**: it will use the serial version of the algorithm, as if the argument was missing
- **`par`**: the algorithm can be parallelised, but not vectorised
- **`par_unseq`**: the algorithm can be parallelised and vectorised
- **`unseq`**: the algorithm can be vectorised but not parallelised (introduced only in C++20)

So, to transform an existing algorithm into a parallel (or vectorised) version, one just needs to add an argument to specify the parallel policy; the effort is minimal.

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

```
std::transform(std::execution::par_unseq,
                             // parallel policy
  in.begin(), in.end(),      // input sequence
  out.begin(),               // output sequence
  ftor);                     // transform fun
```

Listing 1

Please note that the library is allowed to completely ignore the execution policy and fall back to the serial execution. Thus, this execution policy provides just a hint for the library, or the maximum parallelisation/ vectorisation level allowed.

Most STL algorithms take the execution policy parameter and can be instructed to run in parallel. There are also some new algorithms that were added to overcome the fact that existing algorithms have constraints that forbid parallelising them, or that there are better ways to express some parallel algorithms: **`reduce`**, **`exclusive_scan`**, **`inclusive_scan`**, **`transform_reduce`**, **`transform_exclusive_scan`**, **`transform_inclusive_scane`**.

For a better introduction and explanation of C++ parallel algorithms, the reader should consult [Lelbach16] [Filipek17a] [Filipek17b] [ONeal18].

Most of our discussion will be focusing on the parallel execution (**`par`** policy), the one that aims to utilise all the cores available to increase efficiency. Vectorisation (**`unseq`** policy) will be briefly touched towards the end of the article.

Problem 1: no concurrency concerns

The first thing to notice is that it's straightforward to adapt existing algorithms and make them parallel. This probably partially explains the success of parallel algorithms (at least at the perception level).

But this ease of use also has a negative side effect. The astute reader might have noticed that we are talking about parallelism, and not about concurrency. See [Pike13] and [Teodorescu20c] for the distinction. Very short, concurrency is a design concern, while parallelism is a run-time efficiency concern.

It is ok, for limited domains, to focus more on efficiency than design, but that's typically not the case with concurrency. Unless one pays attention to concurrent design, one will get suboptimal efficiency. In other words, multicore efficiency is a global optimisation problem, not a local one.

C++ parallel algorithms don't allow a global concurrency design. It allows only local optimisations, by making some algorithm calls parallelisable.

Considering this, things are awful from a didactical point of view: the C++ standard might teach people that one needs not to pay attention to concurrency issues; these would be magically solved by using STL. I hope that it's clear by now that this is not the case.

If we generalise a bit, we may come to the conclusion that this is the same problem that led us to bad concurrency design in the first place. Instead of recognising that concurrency needs a completely new type of design, we tried to 'patch' the old imperative and serial thinking by adding the ability

While adding parallel versions to some STL algorithms is a good thing, I argue that this is not such a big advancement as one might think

to run on multiple threads; see also Kevlin Henney's discussion on synchronisation quadrant [Henney17]. To do proper concurrency, we need to drop the old ways of writing software, and adopt a new paradigm.

But maybe the C++ committee never intended to solve the concurrency problem, they only wanted to solve some local efficiency problems, and it's just a misunderstanding of their goal. Let's tackle efficiency concerns.

Problem 2: applications have more than algorithms

Remember Amdahl's law? To have speed improvements from parallelism, one needs to have a significant part of the code that is parallelisable. In the case of Parallel STL[1], one needs to have significant parts of the application using such STL algorithms.

Let me repeat this: in order to have relevant performance benefits, the vast majority of the time needs to be spent in STL algorithms.

This is somehow hard to achieve for a lot of the applications. Not every program consists just in STL algorithm calls. Actually, many programs out there have flows that it are very hard to reduce to STL algorithms, if it's even possible. Lots of these applications can only expose control flow concurrency, making them unsuitable for using parallel algorithms.

One example that I have in mind is applications that do a lot of graph processing. For most of the cases, there aren't any STL algorithms ready to be used.

Problem 3: multiple algorithms introduce serial behaviour when combined

Let's assume that to solve a particular problem one needs to call multiple STL algorithms. If the algorithms were to interact, then they need to be called in a serial fashion; i.e., call one algorithm, then call another, etc. The scenario is illustrated in Figure 1.

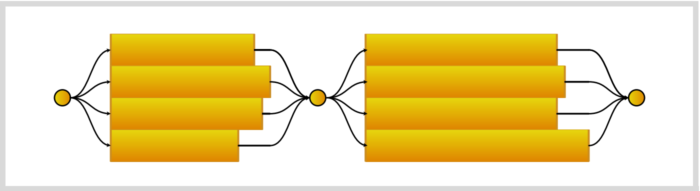


Figure 1

As one can see in the picture, there are three parts of the flow that are bound to execute serially: before the first algorithm, between the algorithm calls and at the end of the second algorithm. And, no matter how good the implementation is, there is some time spent in synchronising and starting tasks; thus, according to Amdahl's law this will put an upper bound of the performance improvement.

But this is not even the worse part. Whenever the algorithm ends, it will have to wait for all the threads to finish executing. This is a consequence of the fact that the amount of work cannot be perfectly distributed between threads, and some threads will process more than others. Typically, this will make the threads stop executing real work, reducing the parallelism capacity of the machine.

To counter this behaviour, one must do one or more of the following:

- start algorithms from multiple threads
- ensure that algorithms have continuations, avoiding serially calling algorithms
- tune the parallel algorithms to ensure that the work split if even.

The first item can be doable outside parallel STL but with no direct support. However, the current design of the standard library does not allow one to implement any of the following two items.

Problem 4: small datasets are not good for parallelisation

Considering Amdahl's law and the overhead one gets from synchronising the exit of an algorithm, let us investigate when it makes sense to parallelise an algorithm. To make it worthwhile for an algorithm to be parallelised, the execution time of the algorithm needs to be somehow large. See my analysis in [Teodorescu20a]. On the type of applications I typically work on, I would say as a rule of thumb that the algorithm needs to take more than 100 milliseconds for being worthwhile to be parallelised.

Please excuse my over-generalisation, but it doesn't seem too rewarding to try to parallelise a 100 millisecond algorithm. Obtaining a linear speedup on a 6-core machine would save 83.3 milliseconds on the completion time, and will fully utilise all the cores. Probably there are better optimisation opportunities.

For an algorithm to take a long time to execute, one of the following conditions must be true:

- the number of elements in the collection that the algorithm operates on needs to be sufficiently large
- the operation given to the algorithm needs to take a long time.

While the second condition can be true in many applications, the first one (numerous elements) is typically not true.

Before we move on, I would beg the reader to consider most of the applications that one worked on. How many of them had algorithms operating on too many elements (e.g, millions of elements), and how many of them had algorithms that are given operations that are long to execute (more than 100 milliseconds per functor call). I bet that there aren't that many applications, especially in the first case.

Now, I've been saying *numerous elements*, but I haven't been too precise. How many elements count as *numerous*?

To give an order of magnitude for this, let's take the examples from [Filipek17b]. The first example considers a simple `transform` operation, with the functor just doubling the value received as a parameter; the algorithm is run over a vector of `double` numbers. For 100k elements, on his more powerful machine (i7 8700, 6 cores), the execution times goes down from 0.524 to 0.359 seconds. That's just a 1.46x improvement for 6 cores. For this type of operation, it's not worth parallelising it.

1. I'm using *Parallel STL* interchangeably with *C++ parallel algorithms*.

After this test, Bartlomiej writes [Filipek17b]:

> As you see on the faster machine, you need like 1 million elements to start seeing some performance gains. On the other hand, on my notebook, all parallel implementations were slower.

Let's consider his *real world* example: computing the Fresnel transform on pairs of position and normal vectors. On this example, it takes 100k elements to have the transformation go from 1.697 to 0.283 (that's a 5.996x improvement; this is good).

Here, we need 100k elements for the performance improvement to be in the order of hundreds of milliseconds.

But, maybe Bartlomiej was having some odd and biased tests. Let's also look at the benchmark that Billy O'Neal made [ONeal18]. After all, Billy implemented parallel algorithms in Microsoft Visual Studio's standard library, so he must know better. His article presents multiple benchmarks on a **`sort`** algorithm. And in all the cases he uses 1 million (with all times reported for release configuration less than 100 ms).

So, as a rule-of-thumb, to see significant performance improvements from parallelising STL algorithms, one needs to have containers with a rough order of magnitude of 1 million elements. From my experience, this doesn't happen too often.

Problem 5: cannot tune the algorithms

Although this problem applies to all the algorithms, let's take the **`sort`** algorithm as a running example.

Sorting 100 integers is faster with a linear algorithm than with a parallel sorting algorithm. So, the algorithms typically have a cutoff point; if the number of elements is below this cutoff point, then the algorithm simply calls the serial version.

In the parallel sort implementation of Intel TBB, this cutoff point is 500 elements. In GCC implementation of parallel algorithms the cutoff point is still 500 elements. My own concore library [concore] also has a cutoff of 500 elements for the sort implementation. This is a common pattern.

But it's not the same thing to sort integers, to sort strings or to sort some complex objects. The cutoff points need to be different. But the C++ standard doesn't allow any tuning of the algorithms.

That is, either the cutoff point is too small for sorting integers, or too large to sort complex objects in parallel. One size does not fit all.

Similarly, let's assume a simple **`for_each`** operation. For some cases, if the algorithm needs to call a heavy function, it's ok to have each element mapped into a task. On the other hand, if the transformation is simple (i.e., an arithmetic operation) then, creating one task per element will be bad for performance. Thus, algorithms may need tuning to accommodate different input patterns.

Listing 2 presents a small snippet using concore library [concore] showing how one can set hints to parallel algorithms.

As mentioned above, the current C++ standard doesn't allow any tuning of the algorithms. This leads to suboptimal performance.

Other notes

Compile time regressions

All the algorithms in the standard library are heavily templatised, and thus so are the parallel versions of the algorithms. They all need to be implemented in C++ headers. But, implementing these parallel versions is far more complicated than implementing the serial versions. Thus, the parallel versions of the algorithms add significant compile-time overhead.

```
concore::partition_hints hints;
hints.granularity_ = 100;
  // cannot process less than 100 elements in
  //a single task
concore::conc_for(itBegin, itEnd, operFtor,
  hints);
```

Listing 2

See [Rodgers18] for some GCC implementation notes, mentioning this problem.

This is not a problem to be taken lightly. It seems that it takes more and more to compile C++ programs. In a lot of code-bases that I've seen the compilation cycles are extremely demoralising; not to mention how bad it is to apply Test-Driven Development in projects that take forever to compile.

Luckily for us, there are a few tricks that library implementers can employ to save part of this problem. For example, GCC will not compile the parallel versions of the library if **`<execution>`** is not included [Rodgers18].

More work to finish faster

This might surprise some readers, but the parallel versions of some algorithms need more work than their serial counterpart to achieve the same results. That is, we do more work in the hope that parallelisation will make the algorithms finish faster. It's a tradeoff between throughput and latency. **`exclusive_scan`** and **`inclusive_scan`** are prime examples of such algorithms.

This is not a weakness of the C++ parallel algorithms, but a fundamental limitation of these parallel algorithms.

Quick change of policy is an advantage

The C++ algorithms are easily turned on by changing (or adding) a parameter to the function call. So, an easy switch can improve the performance of some applications. This is a good thing.

Vector level parallelism is OK

All the discussion so far was focused on parallel algorithms that perform work on multiple cores. I.e., the **`par`** execution policy. Let's briefly turn our attention to the vectorisation (**`unseq`** policy).

If one cannot achieve good parallelism without having a more global perspective, to efficiently use vectorisation, one typically must focus on the local computations. This local focus makes it perfect for vectorisation to be applied at the STL algorithms level.

This can potentially unlock a larger portion of the computation power available on modern computers [Parent16].

Conclusions

C++ parallel algorithms bring vectorisation to common algorithms, without requiring a lot of effort from the user. This is great! C++ parallel algorithms also bring parallel versions of the algorithms without requiring a lot of effort from the user. However, this is not that great. It is a tool that can be useful in some cases, but it's not great.

First, there is a central design issue. Moving to multicore programming requires a somehow-global concurrency design. C++ parallel algorithms don't offer any support for this. Moreover, the standard sends a bad signal to all the C++ programmers that somehow parallelism can be achieved without concurrent thinking. I can't stress enough how bad is this.

As applications have more than just algorithms, we argue that using C++ parallel algorithms is not enough to achieve good speedups to most of the applications.

Then, we walk over some other problems, related to lower-level performance issues: multiple algorithms are serialised, typical small datasets make performance gains small, impossibility of tuning the algorithms. So, besides the high-level concurrency design issue, we have efficiency hurdles at low-level too.

It is true that it's easy for a user to quickly change the policy of the STL algorithms and maybe get some performance benefits. But my focus in this article is on the empty half of the glass. I'm arguing that the benefits are not as big as one could obtain with a proper concurrency design. In some limited cases (i.e., many elements, or functors that are too complex) one might get some speedups for one's algorithms. But even in these cases, the costs of spiking into using multiple cores may have an overall negative performance costs.

All these make C++ parallel algorithms not such a great addition in the concurrency toolkit. They are ok, they are needed, but it's still not enough. I would argue that basic executors support would have been a better addition to the standard. But the executors didn't make it into the 2020 standard. So, let's wait to see what the future will reserve us. ■

References

[concore] Lucian Radu Teodorescu, *Concore library*, https://github.com/lucteo/concore

[Filipek17a] Bartlomiej Filipek, *C++17 in details: Parallel Algorithms*, 2017, https://www.bfilipek.com/2017/08/cpp17-details-parallel.html

[Filipek17b] Bartlomiej Filipek, *The Amazing Performance of C++17 Parallel Algorithms, is it Possible?*, 2017, https://www.bfilipek.com/2017/08/cpp17-details-parallel.html

[Henney17] Kevlin Henney, *Thinking Outside the Synchronisation Quadrant*, ACCU 2017 conference, 2017, https://www.youtube.com/watch?v=UJrmee7o68A

[Lelbach16] Bryce Adelstein Lelbach, *C++ Parallel Algorithms and Beyond*, CppCon 2016, 2016, https://www.youtube.com/watch?v=Vck6kzWjY88

[ONeal18] Billy O'Neal, *Using C++17 Parallel Algorithms for Better Performance*, Microsoft C++ Team Blog, 2018, https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/

[Parent16] Sean Parent, *Better Code: Concurrency*, code::dive 2016 conference, 2016, https://www.youtube.com/watch?v=QIHy8pXbneI

[Pike13] Rob Pike, *Concurrency Is Not Parallelism*, https://www.youtube.com/watch?v=cN_DpYBzKso

[Rodgers18] Thomas Rodgers, *Bringing C++ 17 Parallel Algorithms to a Standard Library Near You*, CppCon 2018, 2018, https://www.youtube.com/watch?v=-KT8gaojHUU

[Sutter05] Herb Sutter, 'The free lunch is over: A fundamental turn toward concurrency in software', *Dr. Dobb's journal*, 2005

[Teodorescu20a] Lucian Radu Teodorescu, 'Refocusing Amdahl's Law', *Overload 157*, June 2020

[Teodorescu20b] Lucian Radu Teodorescu, 'The Global Lockdown of Locks', *Overload 158*, August 2020

[Teodorescu20c] Lucian Radu Teodorescu, 'Concurrency Design Patterns', *Overload 159*, October 2020

We invited you to vote for your favourite articles of 2020 in both *Overload* and *CVu* (our sister publication for members). The results have now been counted.

And the winners are…

CVu

What Is Your Number?

By Simon Sebright
CVu 32.5, November 2020

Overload

C++ Modules: A Brief Tour

By Nathan Sidwell
Overload 159, October 2020

The runners-up are:

From *CVu*:

- Adding Python 3 Compatibility to Python 2 Code
  By Silas S. Brown, *CVu* 32.1 (March 2020)
- The Ethical Programmer
  By Pete Goodliffe, *CVu* 32.1 (March 2020)
- Diving into the ACCU Website
  By Matthew Jones, *CVu* 32.2 (May 2020)
- Expect the Unexpected (Part 1)
  By Pete Goodliffe, *CVu* 32.2. (May 2020)
- The Standards Report
  By Guy Davidson, *CVu* 32.4 (September 2020)

From *Overload*:

- Afterwood: Assume Failure By Default
  By Chris Oldwood, *Overload* 159 (October 2020)

That we have such a large number of articles sharing the runner-up position in *CVu* is both a testament to the high quality of the articles in the journal and evidence of the wide-ranging interests of our readers. Just in case you haven't read them, each of the article titles is a link to that article on our website. You must be logged in to access the articles from *CVu*.

C++ – an Invisible Foundation of Everything

What is C++ and why do people still use it? Bjarne Stroustrup provides a short note answering these questions.

I am often asked variations of the questions 'What is C++?' and 'Is C++ still used anywhere?' My answers tend to be detailed, focused on the long term, and slightly philosophical, rather than simple, fashionable, and concrete. This note attempts a brief answer. It presents C++ as 'a stable and evolving tool for building complex systems that require efficient use of hardware'. Brief answers are necessarily lacking in depth, subtlety, and detail – for detailed and reasoned explanations backed by concrete examples, see 'References and resources' on page 10. This note is mostly direct or paraphrased quotes from those sources.

Overview

This note consists of:

- *Aims and means* – the high-level aims of C++'s design and its role in systems
- *Use* – a few examples of uses of C++ focusing on its foundational uses
- *Evolution* – the evolutionary strategy for developing C++ based on feedback
- *Guarantees, Language, and Guidelines* – the strategy for simultaneously achieving evolution, stability, expressiveness, and complete type-and-resource safety
- *People* – a reminder of the role of people in software development
- *References and resources* – an annotated list of references that can lead to a deeper understanding of C++
- *Appendix* – a very brief summary of C++'s key properties and features

Aims and means

C++ was designed to solve a problem. That problem required management of significant complexity and direct manipulation of hardware. My initial ideals for C++ included

- **The efficiency of C for low-level tasks**
- **Simula's strict and extensible type system**

What I did not like included

- **C's lack of enforcement of its type system**
- **Simula's non-uniform treatment of built-in types and user-defined types** (classes)
- **Simula's relatively poor performance**
- **Both languages' lack of parameterized types** (what later became templates)

Bjarne Stroustrup Bjarne is the designer and original implementer of C++. To make C++ a stable and up-to-date base for real-world software development, he has stuck with its ISO standards effort for almost 30 years (so far). You can contact him via his website: www.stroustrup.com.

This set off a decades-long quest to simultaneously achieve

- **Expressive code**
- **Complete type-and-resource safety**
- **Optimal performance**

I did not want a specialized tool just for my specific problem (support for building a distributed system), but a generalization to solve a large class of problems:

- **C++ is a tool for building complex systems that require efficient use of hardware**

That's a distillation of my initial – and current – aims for C++. Suitably fleshed out with details and implications, this explains much about modern C++. That statement is not a snappy slogan of the form ***C++ is an <<adjective>> language*** but I have never found a sufficiently accurate and descriptive adjective for that. Shifting the focus from language use to language technicalities, we can say:

- **C++ is a general-purpose language for the definition and use of light-weight abstractions**

That leaves the definition of 'general-purpose', 'light-weight', and 'abstraction' open to debate. In C++ terms, I am primarily thinking about classes, templates, and concepts; about expressiveness and efficient use of time and space.

To elaborate a bit further:

- **C++ supports building resource-constrained applications and software infrastructure**
- **C++ supports large-scale software development**
- **C++ supports completely type-and-resource-safe code**

Technically, C++ rests on two pillars:

- **A direct map to hardware**
- **Zero-overhead abstraction in production code**

By 'zero-overhead', I mean that roughly equivalent functionality of a language feature or library component cannot by expressed with less overhead in C or C++:

- **What you don't use, you don't pay for** (aka 'no distributed fat')
- **What you do use, you couldn't hand-code any better** (e.g., dynamic dispatch)

It does not mean that for a more-specific need you can't write more efficient code (say in assembler).

Use

Many well-known applications/systems are written in C++ (e.g., Google search, most browsers, Word, the Mars Rovers, Maya). All systems need to use hardware and large systems must manage complexity. Supporting those fundamental needs has allowed C++ to prosper over decades :

- **C++ is an invisible foundation of everything**

C++ was designed to solve a problem. That problem required management of significant complexity and direct manipulation of hardware

'Everything' is obviously a bit of an exaggeration, but even systems without a line of C++ tend to depend on systems written in C++. 'Everything' is a good first approximation.

Foundational uses of C++ are typically invisible, often even to programmers of systems relying on C++: to be usable by many, a complex system must protect its users from most complexities. For example, when I send a message, I don't want to know about message protocols, transmission systems, signal processing, task scheduling, processor design, or provisioning. Thus, we find C++ in virtual machines (HotSpot, V8), numerics (Eigen, ROOT), AI/ML (TensorFlow, PyTorch), graphics and animation (Adobe, SideFx), communications (Ericsson, Huawei, Nokia), database systems (Mongo, MySQL), finance (Morgan Stanley, Bloomberg), game engines (Unity, Unreal), vehicles (Tesla, BMW, Aurora), CAD/CAM (Dassault, Autodesk), aerospace (Space-X, Lockheed Martin), microelectronics (ARM, Intel, Nvidia), transport (CSX, Maersk), biology and medicine (protein folding, DNA sequencing, tomography, medical monitoring), embedded systems (too many to mention), and much more that we never see and typically don't think of – often in the form of libraries and toolkits usable from many languages. C++ is also key in components and implementations of many different programming languages (GCC, LLVM).

We also find C++ in 'everyday' applications, such as coffee machines and pig-farm management. However, the role as a foundation for systems, tools, and libraries has critical implications for C++'s design, use, and further evolution.

Evolution

Since its inception, C++ has been evolving. That reflects both necessity and an early deliberate choice:

- **No language is perfect for everything and for everybody** (that includes C++)
- **The world changes** (e.g., there were no mobile apps until about 2005)
- **We change** (e.g., few industrial programmers appreciated generic programming in 1985)

Thus

- **C++ must evolve to meet changing requirements and uses**
- **Design decisions must be guided by real-world use** – all good engineering relies on feedback

To evolve, C++ must

- **Offer stability** – organizations that deliver and maintain systems lasting for decades can't constantly rewrite their systems to keep up with incompatible changes to their foundations.
- **Be viable at all times** – must be effective for problems in its domain at all times; you can't take a 'gap year' from improving the language and its implementation.
- **Be directed by a set of ideals** – to remain coherent, the development of language features must be guided by a framework of principles and long-term aims.

Why continue to evolve after years of success? There never was a shortage of people who would prefer to stay with C or move to one of the latest fashionable languages. People can to do exactly that if it makes sense to them, but

- **C++ is a good solution to a wide range of problems**
- **There are hundreds of billions of lines of working C++ code 'out there'**
- **There are millions of C++ programmers**

It takes significant time for a language to mature to be adequate for a range of uses far beyond the understanding of its original designers. Some design tensions are inherent

- **Every successful language will eventually face the problem of evolution vs. stability**
- **Every general-purpose language must serve both (relative) novices and seasoned experts**

Successful language design – like all successful engineering – requires good fundamental ideas and a careful balancing of constraints. Optimizing for just a single desirable property can offer advantages for one application area for one moment of time, but eventually the result dies for lack of adaptability. By now, C++ has survived for 40 years by carefully balancing concerns, learning from experience, and avoiding chasing fashions.

- **A general-purpose language must maintain a careful balance of user needs**

Essential concerns that must be balanced include:

- **simplicity, expressiveness, safety, run-time performance, support for tool building, ease of teaching, maintainability, composability of software from different sources, compilation speed, predictability of response, portability, portability of performance, and stability**

'Simplicity' refers to how ideas are expressed in source code, 'expressiveness' determines the range of uses, 'safety' to type safety and absence of resource leaks, and 'predictability' is essential for many embedded systems.

Guarantees, language, and guidelines

C++ is complicated, but people don't just want a simpler language, they also want improvements and stability:

- **Simplify C++**
- **Add these new features**
- **Don't break my code**

These are reasonable requests so we need a way out of this 'trilemma'. We cannot simplify the language without breaking billions of lines of code and seriously disrupt millions of users. However, we can dramatically simplify the use of C++:

- **Keep simple tasks simple**
- **Ensure that nothing essential is impossible or unreasonably expensive**

To do that

- **Provide simpler alternatives for simple uses**
- **Provide simplifying generalizations**
- **Provide alternatives to error-prone or slow features**

Often, a significant improvement involves a combination of those three.

- **Design C++ code to be tunable**

A high-level abstraction presents a simple, safe, and general interface to users. When needed, a user – not just a language implementer – can provide an alternative implementation or an improved solution. This can sometimes lead to orders-of-magnitude performance improvements and/or enhanced functionality. By using lower-level or alternative abstractions, we can eventually get to use the hardware directly, sometimes even to directly access special-purpose hardware (e.g., GPUs or FPGAs).

From the earliest days, a major aim for the evolution of C++ was to deliver

- **Complete type-and-resource safety**

Much of the evolution of C++ can be seen as gradually approaching that ideal, starting with adding function declarations (function prototypes) to C. By 'type safety', I mean complete static (compile-time) checks that an object is used only according to its defined type augmented by guaranteed run-time checks where static checking is infeasible (e.g., range checking). Simula offered that but at significant cost implying lack of applicability in key areas.

- **Making the type system both strict and flexible is key to correctness, safety, and performance**

Type-safety is not everything, though:

- **Correctness, safety, and performance are system properties, not just language features**
- **A type-safe program can still contain serious logic errors**
- **Test early, often, and systematically**

To simplify use, we need tools and guidelines. The *C++ Core Guidelines* (see 'References and resources' on page 10) offer rules for simple, safe, and performant use:

- **No resource leaks** (incl. no leaks of non-memory resources, such as locks and thread handles)
- **No memory corruption** (an essential pre-condition for any guarantee)
- **No garbage collector** (to avoid indirections in access, memory overheads, and collection delays)
- **No limitation of expressiveness** (compared to well-written modern C++)
- **No performance degradation** (compared to well-written modern C++)

These guarantees cannot be provided for arbitrarily complex C++ code. Therefore, the Core Guidelines include rules to ensure that static analysis can offer the needed guarantees. The guidelines are a key part of my strategy for a gradual evolution of C++:

- **Improve C++ by adding language features and libraries**
- **Maintain stability/compatibility**
- **Provide a variety of strong guarantees through selectively enforced guidelines**

The Core Guidelines are in production use, often supported by static analysis. The guidelines can be enforced by a compiler, but the aim is not to impose a single style of use on the whole C++ community. That would fail because of the widely varying needs and styles of use. By default, enforcement must be selective and optional. A separate static analyzer – usable with any ISO C++ compatible implementation – would be ideal. If a specific 'dialect' (that is, a specific set of rules and enforcement profiles) is to be enforced, it can be done through control of the build process (possibly supported by compiler options).

People

Code is written by people. A programming language is a tool, just one part of a tool chain for a technical community. This was recognized from the start. Here is the opening statement of the first edition of *The C++ Programming Language*:

> C++ is a general-purpose programming language designed to make programming more enjoyable for the serious programmer.

By 'serious programmer' I meant 'people who build systems for the use of others'. This concern for the human side of system development has also been expressed as:

- **Design and programming are human activities; forget that and all is lost**

C++ serves a huge community. To improve software, we need not just to improve the language. We must also bring the community along – supported by education, libraries, and tools. This must be done carefully because no individual can know every use of C++ or every user need.

References and resources

B. Stroustrup: 'Thriving in a crowded and changing world: C++ 2006-2020' *ACM/SIGPLAN History of Programming Languages conference, HOPL-IV*. June 2020. This is the best current description of C++'s aims, evolution, and status. At 160 pages, it is not a quick read. Available at https://dl.acm.org/doi/abs/10.1145/3386320

H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong: *DIRECTION FOR ISO C++* . WG21 P2000. 2020-07-15. Outlines the direction of C++'s evolution, co-authored and continuously updated by the ISO C++ Standard committee's Direction Group as a guide to members. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2000r2.pdf

B. Stroustrup: *The Design and Evolution of C++* Addison Wesley, ISBN 0-201-54330-3. 1994. This book contains lists of design rules for C++, some early history, and many code examples.

B. Stroustrup: *A Tour of C++* (2nd Edition) ISBN 978-0134997834. Addison-Wesley. 2018. A brief – 210 page – tour of the C++ Programming language and its standard library for experienced programmers.

B. Stroustrup: *Programming – Principles and Practice Using C++* (2nd Edition). Addison-Wesley. ISBN 978-0321992789. May 2014. A programming text book aimed at beginners who want eventually to become professionals.

The C++ Core Guidelines. A set of guidelines for safe and effective use of modern C++. Many of the guidelines are enforceable through static analysis. 2014-onwards. Available at https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

Infographic: C/C++ Facts We Learned Before Going Ahead with CLion. A 2015 report on a survey of C++ use, estimating the C++ user community to be 4.5 million strong and listing major industrial use. Today, there are more users. Available at https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/

B. Stroustrup, H. Sutter, and G. Dos Reis: 'A brief introduction to C++'s model for type- and resource-safety'. Isocpp.org. October 2015. An early summary of the aims of the core guidelines as they relate to type safety and resource safety. Available at https://www.stroustrup.com/resource-model.pdf

B. Stroustrup: How can you be so certain? P1962R0. 2019-11-18. A caution against shallow arguments for fashionable causes. Language design requires a certain amount of humility. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1962r0.pdf

B. Stroustrup: Remember the Vasa! P0977r0. 2018-03-06. A note of warning about overenthusiastic 'improvement' of the language. Available at https://www.stroustrup.com/P0977-remember-the-vasa.pdf

The C++ Foundation's Website describes the organization and progress of the standards effort. https://isocpp.org/std

www.stroustrup.com offers many of my videos, papers, interviews, and quotes, including:

- My CppCon'14 keynote: 'Make Simple Tasks Simple!' at https://www.youtube.com/watch?v=nesCaocNjtQ
- My Cppcon'17 keynote: 'Learning and Teaching Modern C++' at https://www.youtube.com/watch?v=fX2W3nNjJIo
- My Cppcon'19 Keynote: 'C++20: C++ at 40' at https://www.youtube.com/watch?v=u_ij0YNkFUs&t=235s
- Lex Fridman's 2019 'Interview with Bjarne Stroustrup' at https://www.youtube.com/watch?v=uTxRF5ag27A&t=1s

Appendix: The C++ language

The description of C++ above does not mention any language features or give any code examples. This leaves it open to serious misinterpretation. I cannot give serious examples of good code here – see 'References and resources' on page 10 – but I can summarize.

There is a reasonably stable core of ideals that guides the evolution of C++ (the references are to my 2020 'History of Programming Languages' paper):

- **A static type system with equal support for built-in types and user-defined types (§2.1)**
- **Value and reference semantics (§4.2.3)**
- **Systematic and general resource management (RAII) (§2.2)**
- **Support for efficient object-oriented programming (§2.1)**
- **Support for flexible and efficient generic programming (§10.5.1)**
- **Support for compile-time programming (§4.2.7)**
- **Direct use of machine and operating system resources (§1)**
- **Concurrency support through libraries (often implemented using intrinsics) (§4.1) (§9.4)**

Key language features with their primary intended roles:

- **Functions** – the basic way of defining a named action. Functions with different types can have the same name. The function invoked is then chosen based on the type of its arguments.
- **Overloading** – allowing semantically similar operations on different types is a key to generic programming.
- **Operator overloading** – a function can be defined to give meaning to an operator for a given set of operand types. Overloadable operators includes the usual arithmetic and logical operators plus `()` (application), `[]` (subscripting), and `->` (member selection).
- **Classes** – user-defined types that can approach built-in types for ease of use, style of use, and efficiency, while opening up a whole new world of general and application-specific types. Classes offer (optional) encapsulation without run-time cost. Class objects can be allocated on the stack, in static memory, in dynamic (heap) memory, or as members of other classes.
- **Constructors and destructors** – the key to C++'s resource management and much of its simplicity of code. A constructor can establish an invariant for a class and a destructor can release any resources an object has acquired during its lifetime. Systematic resource management using constructors and destructors is often called RAII ('Resource Acquisition Is Initialization').
- **Class hierarchies** – the ability to define one class in terms of another so that the base class can be used as an interface to derived classes or as part of the implementation of derived classes. The key to traditional object-oriented programming.
- **Virtual functions** – provide run-time type resolution within class hierarchies.
- **Templates** – allow types, functions, and aliases to be parameterized by types and values. The workhorse of C++ generic programming.
- **Concepts** – compile-time predicates on sets of types and values. Mostly used as precise specifications of a template's requirements on its parameters, thereby allowing overloading. A concept taking a single type argument is roughly equivalent to a type, except that it does not specify object layout.
- **Function objects** – objects of classes (often class templates) supporting an application operator `()`. Acts like functions but are objects that can carry state.
- **Lambdas** – a notation for defining function objects.
- **Immutability** – immutable objects can be defined. Access through pointers or references can be declared to be non-mutating.
- **Modules** – a mechanism for encapsulating a set of types, functions, and objects with a well-defined interface offering good information hiding. To use a module, you import it. A program can be composed out of modules.
- **Namespaces** – for separating major components of a program and avoiding name clashes.
- **Exceptions** – for signaling errors that cannot be handled locally. The backbone of much error handling. Exceptions are integrated with constructors and destructors to enable systematic resource management.
- **Type deduction** – to simplify notation by not requiring the programmer to repeat what the compiler already knows. Essential for generic programming and simple expression of ideas.
- **Compile-time functions** – part of comprehensive support for compile-time programming.
- **Concurrency** – lock-free programming, threads, and coroutines.
- **Parallelism** – parallel algorithms.

In addition, there is a relatively large and useful standard library and loads of other libraries. Don't try to write everything yourself in the bare language. ■

Acknowledgements

This note is mostly direct or paraphrased quotes from the referenced papers, so many thanks to the contributors to those as listed in their acknowledgement sections. Also thanks to Gabriel Dos Reis, J.C. van Winkel, Herb Sutter, J-Daniel Garcia, Roger Orr and the unnamed *Overload* reviewers who made constructive comments on earlier drafts.

Test Precisely and Concretely

Tests can hit complete coverage but fail to communicate. Kevlin Henney reminds us that assertions should be necessary, sufficient, and comprehensible.

It is important to test for the desired, essential behaviour of a unit of code, rather than for the incidental behaviour of its particular implementation. But this should not be taken or mistaken as an excuse for vague tests. Tests need to be both accurate and precise.

Something of a tried, tested, and testing classic, sorting routines offer an illustrative example – they are to computer science as fruit flies are to genetics. Implementing sorting algorithms is far from an everyday task for a programmer, commodified as they are in most language libraries, but sorting is such a familiar idea that most people believe they know what to expect from it. This casual familiarity, however, can make it harder to see past certain assumptions.

A test of sorts

When programmers are asked, 'What would you test for?', by far and away the most common response is something like, 'The result of sorting a sequence elements is a sorted sequence of elements.' As definitions go, it's perhaps a little circular, but it's not false.

So, given the following C function:

```
void sort(int values[], size_t length);
```

and some values to be sorted:

```
int values[length];
```

the expected result of sorting:

```
sort(values, length);
```

would pass the following:

```
assert(is_sorted(values, length));
```

for some appropriate definition of **`is_sorted`**.

While this is true, it is not the whole truth. First of all, what do we mean when we say the result is 'a sorted sequence of elements'? Sorted in what way? Most commonly, a sorted result goes from the lowest to the highest value, but that is an assumption worth stating explicitly. Assumptions are the hidden rocks many programs run aground on – if anything, one goal of testing and other development practices is to uncover assumptions rather than gloss over them.

So, are we saying they are sorted in ascending order? Not quite. What about duplicate values? We expect duplicate values to sort together rather than be discarded or placed elsewhere in the resulting sequence. Stated more precisely, 'The result of sorting a sequence of elements is a sequence of elements sorted in non-descending order.' *Non-descending* and *ascending* are not equivalent.

Kevlin Henney is an independent consultant, speaker, writer and trainer. His development interests include programming languages, software architecture and programming practices, with a particular emphasis on unit testing and reasoning about practices at the team level. Kevlin loves to help and inspire others, share ideas and ask questions. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*. He is also editor of *97 Things Every Programmer Should Know* and co-editor of *97 Things Every Java Programmer Should Know*.

Going to great lengths

When prompted for an even more precise condition, many programmers add that the resulting sequence should be the same length as the original. Although correct, whether or not this deserves to be tested depends largely on the programming language.

In C, for example, the length of an array cannot be changed. By definition, the array length after the call to **`sort`** will be the same as it was before the call. In contrast to the previous point about stating and asserting assumptions explicitly, this is not something you should or could write an assertion for. If you're not sure about this, consider what you might write:

```
const size_t expected = length;
sort(values, length);
assert(length == expected);
```

The only thing being tested here is that the C compiler is a working C compiler. Neither **`length`** nor **`expected`** will – or can – change in this fragment of code, so a good compiler could simply optimise this to:

```
sort(values, length);
assert(true);
```

If the goal is to test **`sort`**, this truism not particularly helpful. It is one thing to test precisely by making assumptions explicit; it is another to pursue false precision by restating defined properties of the platform.

The equivalent **`sort`** in Java would be

```
... void sort(int[] values) ...
```

And the corresponding tautologous test would be

```
final int expected = values.length;
sort(values);
assert values.length == expected;
```

Sneaking into such tests I sometimes also see assertions along the lines of

```
assert values != null;
```

If the criteria you are testing can't be falsified by you, those tests have little value – hat tip to Karl Popper:

> In so far as a scientific statement speaks about reality, it must be falsifiable: and in so far as it is not falsifiable, it does not speak about reality.

This is not to say you will never encounter compiler, library, VM, or other platform bugs, but unless you are the implementer of the compiler, library, VM, or other platform, these are outside your remit and the reality of what you are testing.

For other languages or data structure choices, that the resulting length is unchanged is a property to be asserted rather than a property that is given. For example, if we chose to use a **`List`** rather than an array in Java, its length is one of the properties that could change and would, therefore, be something to assert had remained unchanged:

```
final int expected = values.size();
sort(values);
assert values.size() == expected;
```

If you already have **code lying around** that has the **same functionality** as the functionality you want to test, you can **use it as a test oracle**

Similarly, where sorting is implemented as a pure function, so that it returns a sorted sequence as its result, leaving the original passed sequence untouched, stating that the result has the same length as the input makes the test more complete. This is the case with Python's own built-in **`sorted`** function and in functional languages. If we follow the same convention for our own **`sort`**, in Python, it would look like

```
result = sort(values)
assert len(result) == len(values)
```

And, unless we are already offered guarantees on the immutability of the argument, it makes sense to assert the original values are unchanged by taking a copy for posterity and later comparison:

```
original = values[:]
result = sort(values)
assert values == original
assert len(result) == len(values)
```

The whole truth

We've navigated the clarity of what we mean by *sorted* and questions of convention and immutability… but it's not enough.

Given the following test code:

```
original = values[:]
result = sort(values)
assert values == original
assert len(result) == len(values)
assert is_sorted(result)
```

The following implementation satisfies the postcondition of not changing its parameter and of returning a result sorted in non-descending order with same length as the original sequence:

```
def sort(values):
  return list(range(len(values)))
```

As does the following:

```
def sort(values):
  return [0] * len(values)
```

And the following:

```
def sort(values):
  return [] if len(values) == 0
    else [values[0]] * len(values)
```

Given the following sequence:

```
values = [3, 1, 4, 1, 5, 9]
```

The first example simply returns an appropriately sized list of numbers counting up from zero:

```
[0, 1, 2, 3, 4, 5]
```

The second example makes even less of an effort:

```
[0, 0, 0, 0, 0, 0]
```

The third example at least shows willing to use something more than just the length of the given argument:

```
[3, 3, 3, 3, 3, 3]
```

This last example was inspired by an error taken from production C code (fortunately caught before it was released). Rather than the contrived implementation shown here, a simple slip of a keystroke or a momentary lapse of reason led to an elaborate mechanism for populating the whole result with the first element of the given array – an *i* that should have been a *j* converted an optimal sorting algorithm into a clunky fill routine.

All these implementations satisfy the spec that the result is sorted and the same length as the original, but what they let pass is also most certainly not what was intended! Although these conditions are necessary, they are not sufficient. The result is an underfitting test that only weakly models the requirement and is too permissive in letting flawed implementations through.

The full postcondition is that 'The result of sorting a sequence of elements is a sequence of the original elements sorted in non-descending order.' Once the constraint that the result must be a permutation of the original values is added, that the result length is the same as the input length comes out in the wash and doesn't need restating regardless of language or call convention.

Oracular spectacular

Are we done? Not yet.

Even stating the postcondition in the way described is not enough to give you a good test. A good test should be comprehensible and simple enough that you can readily see that it is correct (or not).

If you already have code lying around that has the same functionality as the functionality you want to test, you can use it as a test oracle. Under the same conditions, the new code should produce the same results as the old code. There are many reasons you may find yourself in this situation: the old code represents a dependency you are trying to decouple from; the new code has better performance than the old code (faster, smaller, etc.); the new code has a more appropriate API than the old code (less error-prone, more type safe, more idiomatic, etc.); you are trying something new (programming language, tools, technique, etc.) and it makes sense to use a familiar example as your testing ground.

For example, the following Python checks our **`sort`** against the built-in **`sorted`** function:

```
values = ...
original = values[:]
result = sort(values)
assert values == original
assert result == sorted(values)
```

Sometimes, however, the scaffolding we need to introduce makes the resulting test code more opaque and less accessible. For example, to test our C version of **`sort`** against C's standard **`qsort`** function, we can use the code in Listing 1 (overleaf).

Even making the narrative structure of the test case more explicit, the code still looks to be more about bookkeeping local variables than about testing **`sort`** (see Listing 2).

the auxiliary **test code** – to check that **a sequence is sorted** and that **one sequence contains a permutation of values** in another – may quite possibly **be more complex than the code under test**

```
int actual[length];
...
int expected[length];
for (size_t at = 0; at != length; ++at)
  expected[at] = actual[at];
qsort(expected, length, sizeof(int),
  compare_ints);
sort(actual, length);
for (size_t at = 0; at != length; ++at)
  assert(actual[at] == expected[at]);
```

Listing 1

```
// Given
int actual[length] = {...};
...
int expected[length];
for (size_t at = 0; at != length; ++at)
  expected[at] = actual[at];
qsort(expected, length, sizeof(int),
  compare_ints);
// When
sort(actual, length);
// Then
for (size_t at = 0; at != length; ++at)
  assert(actual[at] == expected[at]);
```

Listing 2

And **compare_ints** is ours to define, so will lie outside the test case, making the test code even harder to assimilate on reading (Listing 3).

Note that this dislocation of functionality is not the same as making test code more readable by extracting clunky bookkeeping code into intentionally named functions (Listing 4).

Refactoring to reduce bulk and raise intention is certainly a practice that should be considered (much, much more often than it is) when writing test code.

Managing expectations

One limitation of testing against an existing implementation is that it might not always be obvious what the expected result is. It is one thing to say, “The new implementation should produce the same results as the old implementation”, but quite another to make clear exactly what those results are. In the case of sorting, this is not much of an issue. In the case of something from a more negotiated domain, such as insurance quotes or delivery scheduling, it might not be clear what ‘the same results as the old implementation’ entails. The business rules, although replicated in the new implementation, may be no clearer with tests than they were without. You may have regression, but you do not necessarily have understanding.

```
int compare_ints(const void * lhs_entry,
  const void * rhs_entry)
{
  int lhs = *(const int *) lhs_entry;
  int rhs = *(const int *) rhs_entry;
  return lhs < rhs ? -1 : lhs > rhs ? 1 : 0;
}
```

Listing 3

```
// Given
int actual[length];
...
int expected[length];
presort_expected_values(expected, actual, length);
// When
sort(actual, length);
// Then
assert_equal_arrays(actual, expected, length);
```

Listing 4

The temptation is to make these rules explicit in the body of the test by formulating the postcondition of the called code and asserting its truth. As we’ve already seen in the case of something as seemingly trivial as sorting, arriving at a sound postcondition can be far from trivial. But now that we’ve figured it out, we could in principle use it in our test (Listing 5).

Where extracting **is_sorted** and **is_permutation** make the postcondition clear and the test more readable, but are left as an exercise for the reader to implement. And herein lies the problem: the auxiliary test code – to check that a sequence is sorted and that one sequence contains a permutation of values in another – may quite possibly be more complex than the code under test. Complexity is a breeding ground for bugs.

Details, details

One response to the evergreen question, “How do we know that our tests are correct?”, it to make the test code significantly simpler. Tony Hoare pointed out that

> There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies.

```
values = ...
original = values[:]
result = sort(values)
assert values == original,
  "Original list unchanged"
assert is_sorted(result), "Non-descending values"
assert is_permutation(result, original),
  "Original values preserved"
```

Listing 5

If the tests are significantly simpler than the code being tested, they are also more likely to be correct

If the tests are significantly simpler than the code being tested, they are also more likely to be correct. And when they are incorrect, the errors are easier to spot and fix.

The solution to the problem has been staring us in the face. Alfred North Whitehead observed that

> We think in generalities, but we live in detail.

Using concrete examples eliminates this accidental complexity and opportunity for accident. For example, given the following input:

```
[3, 1, 4, 1, 5, 9]
```

The result of sorting is the following:

```
[1, 1, 3, 4, 5, 9]
```

No other answer will do. And there is no need to write any auxiliary code. Extracting the constancy check as a separate test case, the test reduces to the pleasingly simple and direct

```
assert sort([3, 1, 4, 1, 5, 9]) ==
  [1, 1, 3, 4, 5, 9]
```

We are, of course, not restricted to only a single example. For each given input there is a single output, and we are free to source many inputs. This helps highlight how the balance of effort has shifted. From being distracted into a time sink by the mechanics and completeness of the auxiliary code, we can now spend time writing a variety of tests to demonstrate different properties of the code under test, such as sorting empty lists, lists of single items, lists of identical values, large lists, etc.

By being more precise and more concrete, the resulting tests will both cover more and communicate more. An understanding of postconditions can guide us in how we select our tests, or our tests can illustrate and teach us about the postconditions, but the approach no longer demands logical completeness and infallibility on our part.

A concrete conclusion

Precision matters. A test is not simply an act of confirmation; it is an act of communication. There are many assertions made in tests that, although not wrong, reflect only a vague description of what we can say about the code under test.

For example, the result of adding an item to an empty repository object is not simply that it is not empty: it is that the repository now has a single entry, and that the single item held is the item added. Two or more items would also qualify as not empty, but would be wrong. A single item of a different value would also be wrong. Another example would be that the result of adding a row to a table is not simply that the table is one row bigger: it's also that the row's key can be used to recover the row added. And so on.

Of course, not all domains have such neat mappings from input to output, but where the results are determined, the tests should be just as determined.

In being precise, however, it is easy to get lost in a level of formality that makes tests hard to work with, no matter how precise and correct they are. Concrete examples help to illustrate general cases in an accessible and unambiguous way. We can draw representative examples from the domain, bringing the test code closer to the reader, rather than the forcing the reader into the test code. ■

This article was previously published online at: https://medium.com/analytics-vidhya/test-precisely-and-concretely-810a83f6309a

A shorter version was also published as a chapter in *97 Things Every Programmer Should Know*, which is collection of short and useful tips for programmers.

As well as contributing a chapter, Kevlin edited the book. You may recognise some of the other contributors to the book from the ACCU journals. Michael Feathers, Pete Goodliffe, Seb Rose, Allan Kelly, Giovani Asproni, Jon Jagger, Alan Griffiths, Russel Winder, Thomas Guest, Peter Sommerlad are those I spotted on a quick glance.

You would be in good company if you wrote for us, and we would help you every step of the way.

Afterwood

Think you've learnt it all? Chris Oldwood reminds us that unlearning then becomes our next problem.

After finishing my first article for *CVu* just over a decade ago, I was asked to come up with a short biography and photo to give ACCU readers a tiny insight into the author. At that point the only thing I'd ever written to describe myself was a CV for job applications but I'd guessed that wasn't really what they were looking for. Instead I had to find a way to sum myself up in just a sentence or two.

I'm a firm believer that 'context is king' and therefore I decided that to distil my essence into such a short piece I should focus on where I came from (programmatically speaking) and where I am now so the reader could extrapolate from that the kinds of projects and organisations that have shaped my programming career, and consequently my writing. Hence I arrived at the bio you now see adorning my articles to this very day (recent pandemic-related tweak notwithstanding).

During my university years and the start of my professional programming career, I saw being an '80's bedroom coder' as a badge of honour. Working at a small software house writing graphics software to run on underpowered PCs required some of the skills I had developed writing demos in assembly during my teens, such as the ability to read what the optimizing compiler had come up with and then find a way to make it work when the compiler got it badly wrong. Who knew that in the real world, though, most code is not written in assembly with speed being the only concern...

My new found interest in networking and distributed systems caused me to leave that behind and enter the corporate world in a freelance capacity. Time marched on, CPUs grew faster and ever more complex, optimizing compilers become reliable, disk and network speeds jostled for position, memory became abundant, and the claims on my CV about my knowledge of PC hardware become weaker as I slowly moved 'further up the stack'. What I once (naively) saw as the meat-and-potatoes of programming had been downgraded to 'mechanical sympathy' [MechSym].

For me any appreciation of new hardware or technology has tended to come from a single impressive moment of its application rather than a change in numbers on a data-sheet. For instance, I had a low opinion of the JVM in the early 2000s until I saw some Atari ST demos that Equinox (an old Atari demo group) had ported to run as Java applets and, while it was sluggish on the Sun JVM, they ran real-time on the Microsoft JVM. Any performance reservations of the mundane Java project I was assigned to at the time dropped away instantly. Sadly they were replaced by a more unexpected time bomb – the buggy date/time class.

My jaw dropped again some years later when I got to see the Linux kernel boot-up inside a web browser without using native code. Likewise, a DHTML version of Lemmings and the Unreal engine helped remove any other preconceptions I might have had around what can be achieved with a browser and a modern JavaScript engine.

One networking epiphany came when I had to debug an occasional crash in a C++ based service and I struggled for some time to believe what my eventual hypothesis was suggesting – that the service could send a financial trade to another machine, value it, and return and process the response on another thread before the sending thread got switched back in. As for disk I/O, which has never really been that stellar under Windows, I sported a rather large grin the first time I experienced compiling C++ code on an SSD.

More recently-ish I attended Jason McGuiness's ACCU talk about the impact of Meltdown and Spectre on high-frequency trading. Any pretence I might still have had that my mental model of what went on inside a modern CPU was readily dismissed; in my heart I probably knew that but it was still a brutal awakening after all those teenage years counting cycles. Although I still occasionally inspect modern assembly code in the debugger it's really the stack traces I've become more interested in as I try to reason about the flow rather than question the compiler's choice of instructions and sequencing to get the best out of a CPU.

Now, as I write these very words, there is a flurry of excitement about Apple's new M1 chip and how its ability to emulate a different CPU architecture faster than the real thing is an impressive achievement. For those "in the know" I'm sure it's just one more inevitable step forward, but for me it's yet another virtualization bubble burst. Even Knuth's MIX is struggling to stay relevant.

And that's one of the downsides with being a programmer as the years whizz past, there comes a point at which you find yourself spending more and more time 'unlearning'. You might say it's really just learning something new but unlearning is really about trying to forget what you learned because the game has changed and you need to catch back up with those that never learned the old ways in the first place. Unchecked, that badge of honour is slowly turning into a millstone.

Modern C++ is yet another example. I used to have a snippet of code I liked to chew over with candidates in an interview that encapsulated various idioms and pitfalls when working in C++. It was a well-honed example based on 15 years of blood sweat and tears and yet most of the discussion points are now moot as the language has changed dramatically since then due to move semantics, lambdas, range-based for loops, etc. Old habits die hard and unlearning that you shouldn't return containers by value because it now Just Works™ is another example where years of inertia can be difficult to overcome.

Luckily there are still some inalienable truths to keep me warm at night, like the speed of light limiting my ping time and giving me an excuse for why I lost at Fortnite, yet again. Having sympathy for the machine is undeniably a valuable skill but who has sympathy for the poor programmer that is forever learning and then unlearning to keep up with the march of progress in an effort to stay relevant in today's fast paced world?

Reference

[MechSym] Mechanical Sympathy: https://mechanical-sympathy.blogspot.com/2011/07/why-mechanical-sympathy.html

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the lounge below his bedroom. With no Godmanchester duck race to commentate on this year, he's been even more easily distracted by messages to 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 purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

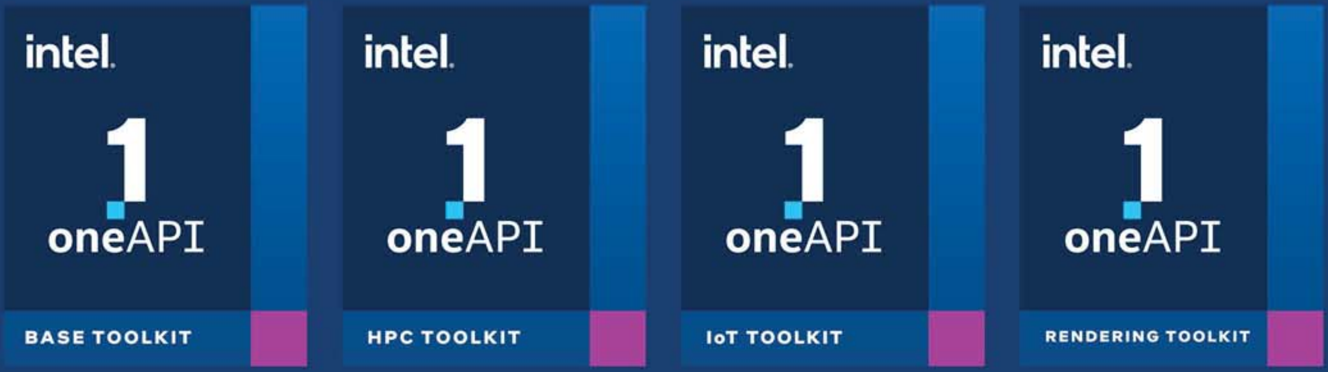
PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

intel software

oneAPI: New Era of Accelerated Computing

Take the open, productive path to accelerate cross-architecture computing using Intel® oneAPI Toolkits.



Developers, take advantage of oneAPI's unified, standards-based, cross-architecture programming model that sets you free to develop applications for your choice of architectures. Get full hardware performance using a complete set of proven tools without the limits of proprietary language lock-in.

Learn more: www.qbssoftware.com