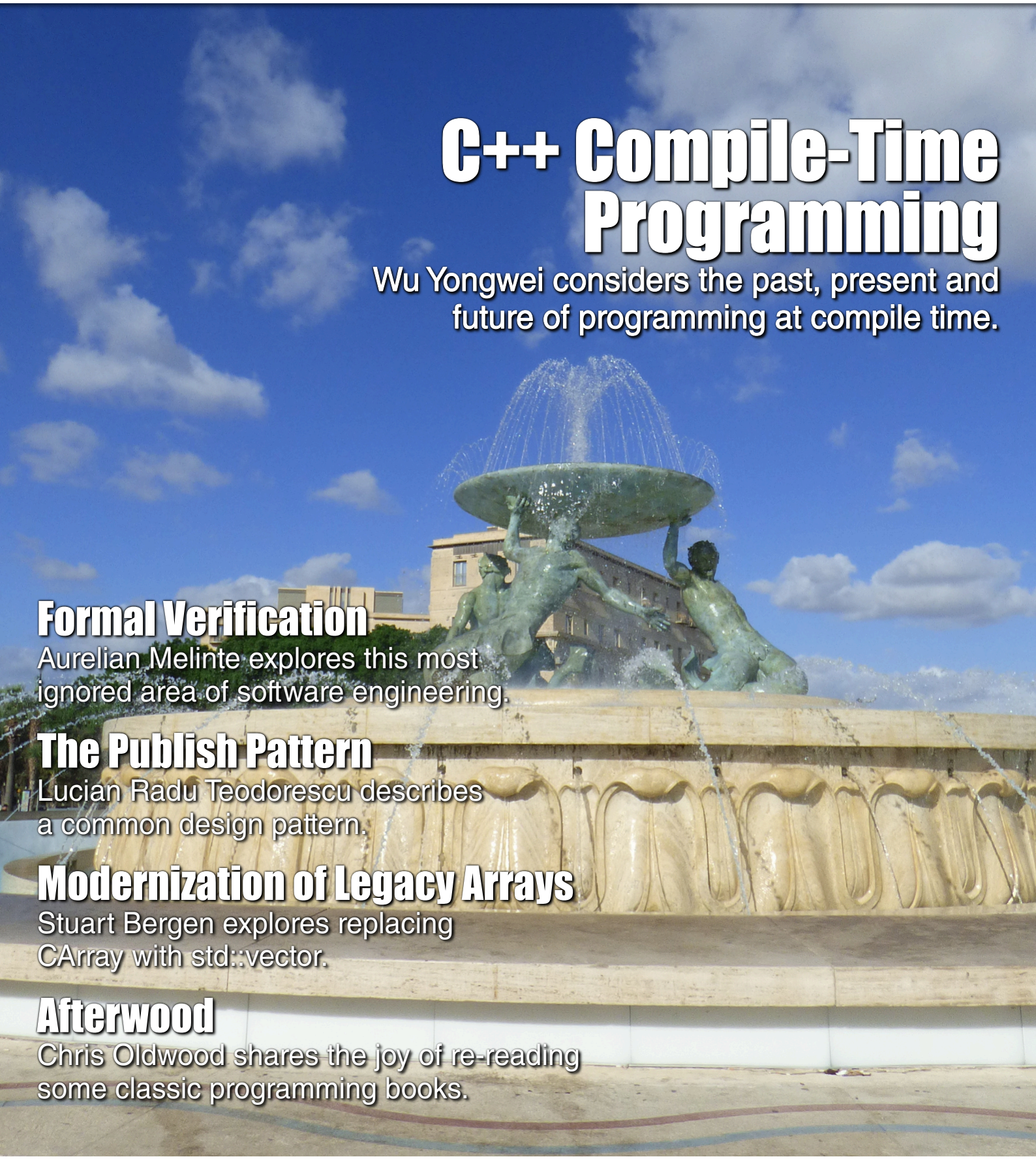
overload 183

OCTOBER 2024 £4.50



C++ Compile-Time Programming

Wu Yongwei considers the past, present and future of programming at compile time.

Formal Verification

Aurelian Melinte explores this most ignored area of software engineering.

The Publish Pattern

Lucian Radu Teodorescu describes a common design pattern.

Modernization of Legacy Arrays

Stuart Bergen explores replacing CArray with std::vector.

Afterwood

Chris Oldwood shares the joy of re-reading some classic programming books.

A magazine of ACCU

ISSN: 1354-3172

October 2024
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

Advisors
Paul Bennett
t21@angellane.org
Matthew Dodkins
matthew.dodkins@gmail.com
Paul Floyd
pjfloyd@wanadoo.fr
Jason Hearne-McGuiness
coder@hussar.me.uk
Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi
Steve Love
steve@arventech.com
Christian Meyenburg
contact@meyenburg.dev
Barry Nichols
barrydavidnichols@gmail.com
Chris Oldwood
gort@cix.co.uk
Roger Orr
rogero@howzatt.co.uk
Balog Pal
pasa@lib.hu
Jonathan Wakely
accu@kayari.org
Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Tim Peck:
Triton Fountain, Valetta, Malta.

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

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

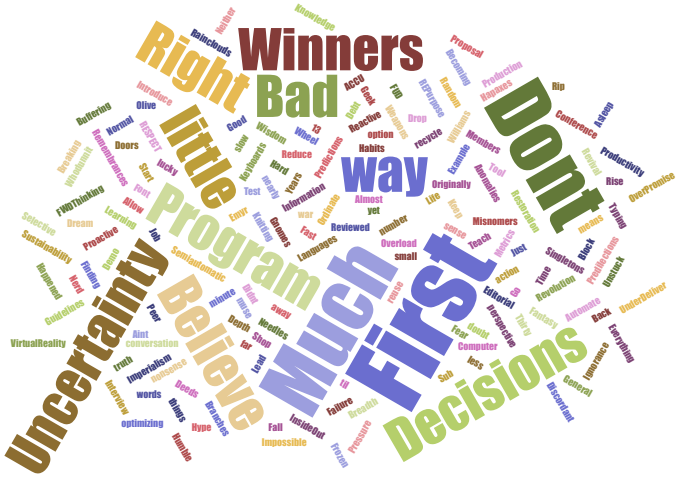
Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copy deadlines
All articles intended for publication in Overload 184 should be submitted by 1st November 2024 and those for Overload 185 by 1st January 2025.

What Does It All Mean?

Trying to make sense of things can be a challenge. Frances Buontempo reminds us something don't entirely make sense and that's OK.

Hello again. I have been very busy trying to think of editorial ideas. I even spent time looking back at my previous titles to avoid repeating myself, however most of them are silly word plays or somewhat enigmatic, so I can't tell what I said before. This time consuming activity has put me off as usual. I have written more than 70, apparently. My attempt to get ChatGPT to summarise my previous efforts wasn't much help. It said, "Her editorial tone is conversational yet insightful" which was nice, though hasn't given me the information I was after. Other forms of "AI" are available. I then wasted time generating a word cloud of the titles.



We clearly see "Program much first" in the centre.

Many people dislike word clouds, and there are more accurate ways of displaying frequencies. However, the random orientation makes you analyse the graphic differently. The Christmas special version of University Challenge has given word clouds of Christmas carols as clues, and you can manage to reform the words into a well known song. Are word clouds meaningful? No, probably not. However, they are one way to represent something.

People tend to search for patterns in things, and ascribe meaning to what they see. I suspect our brains are just wired like that. Remembering a pattern is easier than recalling a full look-up table. The act of summarizing data into information either as equations, rules or patterns is a fundamental part of how we think. We can therefore delude ourselves easily, finding patterns where there are none. This propensity probably helps to fuel conspiracy theories and might perhaps be behind recent riots in the UK. The internet is littered with various articles about this, but the BBC has a reasonable write-up [BBC-1].

Riots break out from time to time, and I recall several, including watching London burn, or at least smolder slightly, from my desk at a bank in Canary Wharf for a few days. Wikipedia lists several English riots [Wikipedia-1] in date order: 1715, 1919, 1947, 1958, 1981, 1991, 2001, 2011 and 2024.

I haven't seen all of these, of course. Now I spot a pattern. 44% end in a 1. According to Benford's law, most numbers start with a 1 [Wikipedia-2], but ending with a 1 is a different matter. Does this mean something? Probably not.

The starting months are of note too: Oct, June, August, August, April, possibly Sept (there were various riots starting in 1991 and continuing sporadically until 1992 [Libcom]), May, August, and July.

Allowing an anomaly of October, these are all when the weather tends to be better. The recent riots stopped when the rain started. Just saying!

The list is incomplete, missing the Nottingham cheese riot from the start of October 1766 [Wikipedia-3]. Further worldwide riots are listed on another page [Wikipedia-4]. Try analyzing that if you have time on your hands.

Anyway, I am not an expert on socio-political matters but I know full well my sample size is too small to be statistically significant. If you find yourself noticing a pattern, do investigate, but hold a cynical thread in mind too, questioning your assumptions and analyses. Spotting non-existent patterns isn't always problematic. Maybe you have looked at clouds and chatted with a friend about animals or other shapes you can see. This is an example of pareidolia, which Wikipedia [Wikipedia-5] describes as:

> the tendency for perception to impose a meaningful interpretation on a nebulous stimulus, usually visual, so that one detects an object, pattern, or meaning where there is none

If you ever see the face of someone famous on your toast, you probably aren't special.

A data visualization technique you might have come across before, uses faces to display data. These are called Chernoff faces ([Wikipedia-6] or the original [Chernoff73]), and use the size of various facial features, eyes, mouth, nose and so on, to graph features in data. You can easily use these for high dimensional data, and Chernoff used faces because people could spot similarities very easily. Our brains seemed to be wired to recognize facial features.

We've discussed pareidolia and people finding meaning where there is none, but we can equally miss meaning and reasons for events or outcomes

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. 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.

where there is significance. For example, people might blame their compiler when their code doesn't work. One case I came across involved a mysterious bug, where "code ended up in the wrong destructor". After a few probing questions from Yours Truly, it became apparent the basics of virtual destructors and derived classes in C++ were unknown to the complainant. I am rather pleased with myself for managing to debug code without even seeing it. Now, it is easy to get deep into a problem though and miss what's right under your nose, so maybe the other guy was tired and forgot the basics, so I might be jumping to conclusions. Nonetheless, if you go in with bias, you might fail to understand what is really happening or even see what you expect to see rather than what's actually there.

Discussing a bug without the code in front of you can be a challenge. Furthermore, words are an imprecise way to communicate, which can make discussion difficult. Some of us are more precise than others. If asked, "Do you want tea or coffee?" some readers will say "Yes." I guess you get used to how people tend to respond and might need to adapt your language to your target audience. Sometimes you think you are talking about the same thing, but later find a mismatch between the meanings or words or phrases. I may have mentioned before running a short coding practice sessions with colleagues, using the awesome cyberdojo [cyberdojo]. I paired up with a team mate, and she was very surprised to see me actually write a test first. I had said I like test-driven development several times, and used the phrase 'test first', but somehow those words meant something different to her. I learnt something and so did she, so learning as a team was a win. A much simpler example is someone going the wrong way. I told my Grandad to go left once, and he went right. The simplest solution to that problem was saying, "No, the other left." He instantly turned the other way without arguing about the ridiculous phrase I had used. Words are weird.

We use precise notation for mathematics to pin down definitions exactly. However, such notation can be difficult to read, even if you do know some maths. One of my 78 browser tabs is a *Nature* article on fast matrix multiplication with reinforcement learning [Fawzi22]. It starts with an abstract, as most academic style papers do, talking about the use of machine learning for automatic algorithm discovery. All good so far. Then we get to the main article, which gives an overview of what's coming up, as you would expect. Then the trouble starts. The first sentence tells me matrix multiplication is bilinear so can be represented by a 3D tensor and offers an illustration to help. It's taken several attempts to figure out what the figure means. So, now I will have to go back to the text and try to read this again. Sometimes pinning things down precisely makes life harder, and yet it is good to be precise sometimes.

Even if you don't need to be exact, knowing why things are the way they are, or a bit of history behind words or ideas can be useful, or at least interesting. Chris Oldwood shared some of his favourite quotes and aphorisms, back in 2023, and considered their origins [Oldwood23]. He began with the phrase "ship-shape and Bristol fashion" which sprouted a sidebar explaining in more detail, essentially everything being in the right place and able to deal with Bristol's tidal river and lots of mud. Francis Glassborow has also been running a 'Meaning of Words' series in *CVu*, since 2021, with many disambiguating commonly muddled words like operator and function, code and cypher and others looking in detail at one or more ideas. I am on the verge of writing another book, and my new editor asked me what a compiler is. Fortunately, part 8 of Francis' series is about 'Assemblers, Translators and Compilers Revisited' where he says [Glassborow22]:

> Fundamentally, a simple compiler converts instructions written in a high level language (with a high level of abstraction from machine level code) into a lower level language with less abstraction.

Unfortunately, I would then have to define high and low level, and abstraction. Sometimes, you have a bootstrap problem. Do your boots have straps? Some, but not all, of mine do. There's another odd phrase. Wikipedia [Wikipedia-7] suggests some possible etymologies, all referring to impossible tasks. Even if your boots have straps, you won't be able to pull yourself out of trouble using them. But you might happily bootstrap a compiler. Try explaining that to someone who doesn't know about computing.

The BBC news website, possibly inspired by Francis, has been running a series by 'The Vocabularist' for a while, and once attempted to explore the roots of the word 'computer' [BBC-2]. The article says 'Computer' comes from the Latin 'putare' which means both to think and to prune. It's a jump to get from gardening to computing, but if you ever prune shrubs or tidy a garden, you are thinking and planning, and possibly even counting, buckets of rubbish, how many hours you have spent… Counting and computing seem to be related words. Of course, a computer is one who computes, so you have another bootstrap problem. How children ever learn language amazes me.

I would end by quoting Ecclesiastes; the second verse of the first chapter

> "Meaningless! Meaningless!" says the Teacher. "Utterly meaningless! Everything is meaningless."

However, that's not a positive note to end on. Some things in themselves might be without meaning, but they can still be fun. You can describe music in technical terms, or via musical notation, but that doesn't ascribe it meaning. It's still fun, beautiful, inspiring or other word of your own choice. It's easy to get distracted but allowing your mind to wander from time to time can be a good thing. You don't need to ensure everything you do is purposeful, but do watch out for seeing patterns where there are none.

References

[BBC-1] 'Why are there riots in the UK?' posted 7 August 2024 and updated 9 August 2024 at https://www.bbc.co.uk/news/articles/ckg55we5n3xo

[BBC-2] The Vocabularist, 'What's the root of teh word computer?', posted 2 February 2016: https://www.bbc.co.uk/news/blogs-magazine-monitor-35428300

[Chernoff73] Herman Chernoff (1973) 'The Use of Faces to Represent Points in K-Dimensional Space Graphically' (PDF) *Journal of the American Statistical Association* 68 (342). American Statistical Association: 361–368

[Cyberdojo] Cyber-dojo: https://beta.cyber-dojo.org/creator/home

[Fawzi22] A. Fawzi, M. Balog, A. Huang *et al*. 'Discovering faster matrix multiplication algorithms with reinforcement learning' *Nature* 610, 47–53 (2022). https://doi.org/10.1038/s41586-022-05172-4

[Glassborow22] Francis Glassborow (2022) 'Assemblers, Translators and Compilers Revisited' in *CVu* 34.5, available at https://accu.org/journals/cvu/34/5/cvu34-5.pdf#Page=9 (You must be a member and logged in to see editions of *CVu*)

[Libcom] 'Hot time: Summer on the estates – Riots in the UK 1991–2' https://libcom.org/article/hot-time-summer-estates-riots-uk-1991-2

[Oldwood23] Chris Oldwood (2023) 'Afterwood' in *Overload* 31(175):19-20, June 2023. https://accu.org/journals/overload/31/175/overload175.pdf#page=21

[Wikipedia-1] 'England riots': https://en.wikipedia.org/wiki/England_riots

[Wikipedia-2] 'Benford's law': https://en.wikipedia.org/wiki/Benford%27s_law

[Wikipedia-3] 'Nottingham cheese riot': https://en.wikipedia.org/wiki/Nottingham_cheese_riot

[Wikipedia-4] 'List of riots': https://en.wikipedia.org/wiki/List_of_riots

[Wikipedia-5] 'Pareidolia': https://en.wikipedia.org/wiki/Pareidolia

[Wikipedia-6] 'Chernoff face': https://en.wikipedia.org/wiki/Chernoff_face

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

Formal Verification

Proving code is correct is challenging. Aurelian Melinte explores verification, the most ignored area of software engineering.

When I see code for a state machine or a protocol I know it was not formally verified (FV); that is it was not proved to operate correctly via some formal methodology. And I know why: most engineers are not even aware that FV exists. Few universities offer FV classes in their software engineering curricula. It is hard to understand why as FV started in research and academia environments in the 60s and, by the 80s, FV toolsets usable by the average CS graduate were available. Since the 80s, hardware design engineers live and die by FV but software engineers are still generally ignorant of it.

Theoreticians might cringe at this definition: by-and-large FV means proving that a verification model – an abstraction of the design to be implemented – is mathematically correct. That is, it does what it is supposed to do and does not do things it is not supposed to do. What exactly this 'does and does not' set of properties means depends of what is being modeled. In practice, FV tools will check all possible execution paths of the model for logical flaws.

Typical designs to be verified are state machines, protocols and algorithms. But the methodologies are flexible enough to model an 8-queen chess puzzle or a Fibonacci series, for instance. Accordingly, the properties to be verified can be anything one decides to be applicable to the model.

Two notes. One, FV verifies the model of a proposed design, not the actual implementation. If the model is proven to be deadlock-free then the implementation can still deadlock. But, if the implementation does not work, at least you know the design behind the implementation is sound. Two, FV will verify only what it is told to, i.e. if you want to verify it is starvation-free then you have to subject the model to that property check lest you miss that important check.

Out of the of dozens FV methodologies [Wikipedia-1], I think two are the dominant ones: Promela/spin and TLA+/PlusCal. Below is a very high-level and hopefully still useful overview of each. The overview may not be very intelligible without further individual homework but the FV topic is at least book-sized. As for the specifics of Promela or TLA+, these are also book-size.

Both methodologies share the same theoretical roots, use the same terminology and even the same linear temporal logic (LTL) [Wikipedia-2] notation. LTL expresses the desired or undesired behavior the model should have over its lifetime and allows one to express properties such as 'if X happens then Y is guaranteed to happen, and it remains set if Z does not occur'.

Properties of a model are, by and large, classified in two bins: safety (the model does not misbehave; does not do 'bad things') and liveness (the model does what is supposed to do).

Aurelian Melinte Aurelian acquired his programming addiction in late 90s as a freshly-minted hardware engineer and is not looking for a cure. He spends most of his spare time reading and exercising. Feel free to contact him at ame01@gmx.net

Typical safety properties are: the model is deadlock-free, does not trip model-specific assertions (that you have to define) and it does not end in an invalid end state. Safety is expressed either as plain assertions, as system invariants, or as LTL properties.

Typical liveness checks are:

- responsiveness: every request gets a response back.
- non-progress cycles over model's lifetime aka starvation: FV checks for conditions leading into infinite and useless runs. Think livelock as an example.
- acceptance: the model is either stuck in an undesirable state or falls into that state infinitely often. The "acceptance" property naming makes sense only when thinking of the under-the-hood verification mechanisms[Spinroot-1].

Promela

Promela/spin [Spinroot-2] originated in the 80s at the Bell Labs. It has a strong practitioners' base in academia and in the telecom companies. Since the telecoms have lost their preeminence in IT these days, that user base is probably shrinking but research in academia seems to be as healthy as ever.

By and large, the spin tool will turn the Promela model into a state-machine which will be verified for the properties you asked for in your model.

The spin tool will also warn about code that was not reached during verification: this alone is priceless as it could be a bug in the model.

With spin, some of the built-in checks are free: dead code, deadlocks, invalid end states; and some are almost-freebies: you may have to do some minimal work and ask for progress and acceptance.

Atomicity is fine-grained, at the statement level. This makes race conditions rather trivial to find. Promela is architecture agnostic, on purpose. Modeling specifically for verification with the C++ memory-model is feasible but requires specific modeling work on top of modeling for your design. In particular, the Dekker algorithm as detailed below was proven to be unsafe on a machine without atomic reads and writes [Buhr15].

Because of the fine-grained atomicity, Promela models tend to be large in a relative sense. With the models still being entirely hand-crafted, it is safe to say that the footprint is orders of magnitude lower than hardware FV models. The first line of defense is reworking the model (e.g. by reducing the number of global variables and other artifacts such as processes and channels). Then there are special model generation options to mitigate the footprint – this was clearly an issue with 90's available machines. Finally, for really large models, there is a best-effort (read: partial) verification available. I have yet to see a model too large for modern machines – the largest I know of would complete verification within minutes.

I have yet to see a model too large for modern machines – the largest I know of would complete verification within minutes.

```
#define true   1
#define false  0
#define Aturn  false
#define Bturn  true

bool x, y, t;
byte ghost;

proctype A()
{  x = true;
   t = Bturn;
   (y == false || t == Aturn);
   /* critical section */
   ghost++;
   assert(ghost == 1);
cspa: ghost--;
      x = false
}

proctype B()
{  y = true;
   t = Aturn;
   (x == false || t == Bturn);
   /* critical section */
   ghost++;
   assert(ghost == 1);
cspb: ghost--;
      y = false
}

init{  atomic { run A(); run B(); } }

ltl criticalSection {[]!(A@cspa && B@cspb)}
```

Listing 1

There are also a number of tools to extract a Promela model from an existing codebase but I have not used these [Spinroot-3].

To learn Promela, one must read Prof. Holzmann's book *The Spin Model Checker* [Holzmann03] and follow the short two-hour video class on the spinroot site. The other book one should read (this is actually another must) is Prof. Ben-Ari's book *Principles of Spin* [Ben-Ari08]. This should suffice to start the journey.

As an example, Listing 1 is a model of the Dekker algorithm [Wikipedia-3] to enforce critical sections/mutual exclusion, sourced from the Promela manual [Spinroot-4].

Easy to read even if you do not know some Promela statements can be blocking until becoming executable. Here we want to verify that the critical section is enforcing mutually exclusive access.

The last line of the model instructs the verifier what to look for. It is a system invariant expressed in the canonical LTL way which states plainly that both processes cannot be in the critical section at the same time during the lifetime of the model.

Assertions can also be used to verify various conditions in conjunction with the LTL or independently of. As shown in the example, the assertions also verify the critical sections holds.

```
--------- MODULE criticalsection5dekker ---------
EXTENDS Integers, Sequences

(*
--algorithm criticalsection5dekker
{
  \* Global variables
  variables turn = 1; wantp = FALSE;
    wantq = FALSE;

  \* The non-critical section.
  \* For checking for freedom from starvation, it
  \* is important that a process might stay in
  \* the non-critical section forever (however,
  \* each process must leave the critical
  \* section).
  \* This procedure covers both cases: finite and
  \* infinite execution of the non-critical
  \* section.
  procedure NCS()
    variable isEndless;
  {
  \* Non-deterministically choose whether the
  \* procedure will be endless or finite.
    ncs0: with (x \in {0,1}) {
      isEndless := x;
    };
    ncs1: if (isEndless = 1) {
      ncs2: while (TRUE) {
        ncs3: skip;
      }
    } else {
      ncs4: return;
    }
  }

  \* First process (name P, pid 1)
  process(P = 1) {
    p0: while (TRUE) {
      p1: call NCS(); \* non-critical section
      p2: wantp := TRUE;
      p3: while (wantq = TRUE) {
        p4: if (turn = 2) {
          p5: wantp := FALSE;
          p6: await turn = 1;
          p7: wantp := TRUE;
        };
      };
      p8: skip; \* critical section
      p9: turn := 2;
      p10: wantp := FALSE;
    }
  }
```

Listing 2

```
  \* Second process (name Q, pid 2)
  process(Q = 2) {
    q0: while (TRUE) {
      q1: call NCS(); \* non-critical section
      q2: wantq := TRUE;
      q3: while (wantp = TRUE) {
        q4: if (turn = 1) {
          q5: wantq := FALSE;
          q6: await turn = 2;
          q7: wantq := TRUE;
        };
      };
      q8: skip; \* critical section
      q9: turn := 1;
        q10: wantq := FALSE;
    }
  }
}
*)
```

Listing 2 (cont'd)

TLA+

TLA+ [Lamport] is pure mathematics describing your model. This is rarefied air that few can adapt to when models get complex. PlusCal is an added layer that describes the model in an imperative-like language that gets translated to TLA+. You will still need to be TLA+ literate but PlusCal puts you within reach of modelling properly.

TLA+ got adopted by Microsoft and other software companies followed suit, so I guess the practitioner base is currently expanding.

The curricula here includes Wayne's book *Practical TLA+* [Wayne18], the training videos on Prof. Lamport's site and his book *Specifying Systems*.

The Dekker algorithm, according to a course at the Stuttgart University [Duerr15], follows. First the PlusCal (lines 29-99 in the original) that you have to write is in Listing 2.

Please note every line is labeled. That is because TLA+ has coarse-grained atomicity: the whole pan of statements between two labels is one atomic op. Since we check for race conditions here, we have to label every statement. TLA+ is also architecture agnostic.

Coarse atomicity generates much smaller models (compared to Promela). The only way I know to further reduce the size of the model is to rework it.

If you go to the full model file on github, you can see he TLA+ translation of the above PlusCal at lines 102-281 – the 'real' verification model you can write by hand if you feel brave.

Finally, the verification properties are shown in Listing 3, lines 283-306 of the original. There are no freebies with TLA+. You have to state what deadlock and starvation look like. Another (rather big) minus: no dead code warnings unless you code model-specific properties checks for it.

Summary

This article has given a quick overview of two dominant FV methodologies, Promela/spin and TLA+/PlusCal. They both have pros and cons, but can proof a design for issues. Though many programmers are unfamiliar with FV, companies do use them. For example AWS have used TLA+ succesfully [Newcombe15] and Promela was used to proof Plan9 concepts [Plan9] and mission-critical applications [Spinroot-5]. ■

```
\*** Mutual exclusion
\* For mutual exclusion, process 1 and process 2
\* must never be in the critical section at the
\* same time.
MutualExclusion == [] ~ (pc[1] = "p8" /\ pc[2]
  = "q8")

\*** Deadlock free
\* If P and Q both want to enter the critical
\* section, one of them will eventually enter the
\* critical section.
NoDeadlock == /\ pc[1] = "p2"
              /\ pc[2] = "q2"
              ~>
              \/ pc[1] = "p8"
              \/ pc[2] = "q8"

\*** Starvation free
\* If P wants to enter the critical section, P
\* will eventually enter the critical section.
\* The same must hold for Q.
NoStarvationP == (pc[1] = "p2") ~> (pc[1] = "p8")
NoStarvationQ == (pc[2] = "q2") ~> (pc[2] = "q8")
NoStarvation == /\ NoStarvationP
                /\ NoStarvationQ

\* Assume weakly fair scheduling of all commands
(* PlusCal options (wf) *)
```

LISTING 3

References

[Ben-Ari08] Mordechai Ben-Ari (2008) *Principles of the Spin Model Checker*, Springer, ISBN-13: 978-1846287695

[Buhr15] Peter Buhr, Dave Dice and Willem Hesselink (2015) 'Dekker's mutual exclusion algorithm made RW-safe', published in Concurrency and Computation, available at https://onlinelibrary.wiley.com/doi/10.1002/cpe.3659

[Duerr15] Frank Duerr (2015) criticalsection5dekker.tla (at October 2024), available at: https://github.com/duerrfk/skp/blob/master/criticalsection5dekker/criticalsection5dekker.tla

[Holzmann03] Gerard J. Holzmann (2003) *The Spin Model Checker, Primer and Reference Manual*, Addison Wesley Professional, ISBN-13: 978-0321228628

[Lamport] Leslie Lamport, the *TLA+* website, available at https://lamport.azurewebsites.net/tla/tla.html

[Newcombe15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff 'How Amazon Web Services uses formal methods', available at https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods

[Plan9] 'Spin & Plan 9', available at: https://swtch.com/spin/

[Spinroot-1] Spinroot manual, 'accept': https://spinroot.com/spin/Man/accept.html

[Spinroot-2] Spinroot website: https://spinroot.com

[Spinroot-3] Modex readme: https://spinroot.com/modex/

[Spinroot-4] Basic Spin Manual: https://spinroot.com/spin/Man/Manual.html

[Spinroot-5] 'Inspiring Applications of Spin', available at: https://spinroot.com/spin/success.html

[Wayne18] Hillel Wayne (2018) *Practical TLA+: Planning Driven Development*, Apress Berkeley, CA, ISBN-13: 978-1-4842-3828-8 (softback)/ 978-1-4842-3829-5 (ebook)

[Wikipedia-1] 'Formal verification': https://en.wikipedia.org/wiki/Formal_verification

[Wikipedia-2] 'Linear temporal logic': https://en.wikipedia.org/wiki/Linear_temporal_logic

[Wikipedia-3] 'Dekker's algorithm': https://en.wikipedia.org/wiki/Dekker's_algorithm

C++ Compile-Time Programming

Programming at compile time has been possible in C++ for a long time. Wu Yongwei considers its past, present and future.

Compile-time programming is a key feature of C++. It enables writing high-performance code often unattainable in other languages. This article explores its past, present, and future applications, highlighting the diverse possibilities in C++. We'll briefly cover template metaprogramming, constexpr, variadic templates, static reflection, and more.[1]

Introduction

Compile-time programming is vastly different from run-time programming. The code runs during compilation, but the results can be used at run time.

Some believe compile-time programming is merely a trick, unused in real-world engineering. To them, I ask: do you use the C++ Standard Library? The mainstream implementations rely heavily on various programming techniques, including compile-time programming.

'I don't write the standard library' – this might be a possible response. But consider this: the standard library is just one tool, a standard weapon. Is it enough to use only standard tools? That's the real question.

The abundance of excellent open-source libraries suggests otherwise. A skilled programmer crafts tools for themselves and their team. If your work feels tedious, perhaps it's time to build a bulldozer to tackle problems.

Compile-time programming offers a way to build such tools.

A bit of history

In traditional C++ programming, source code is compiled into executable code, which runs when executed. However, code can also be executed at compile time. This concept dates back over 30 years.

Bjarne Stroustrup proposed templates in 1988 [Stroustrup88]. It was implemented in 1989 and fully described in the 1990 *Annotated C++ Reference Manual* [Ellis90]. The initial purpose of template design was to express parameterized container classes. However, in order to implement types similar to arrays, non-type template parameters (NTTP) were introduced:

```
template<class E, int size> class buffer;
buffer<char, 1024> x;
```

Around the same time, Alex Stepanov and David Musser introduced *generic programming*. Alex realized C++ templates perfectly suited his needs and used them to implement the Standard Template Library (STL), showcasing their real-world potential.

Alex Stepanov remarked on the capabilities of the C++ language [Stroustrup94]:

> C++ is a powerful enough language – the first such language in our experience – to allow the construction of generic programming components that combine mathematical precision, beauty, and abstractness with the efficiency of non-generic hand-crafted code.

1 Links to godbolt.org are to online examples, supplementing this article.

```
template <int i> struct D {
  D(void*); operator int();
  };
template <int p, int i> struct is_prime {
  enum {
    prim = (p%i) &&
      is_prime<(i > 2 ? p : 0), i -1> :: prim };
  };
template < int i > struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i, i-1>::prim };
  void f() { D<i> d = prim; }
  };
struct is_prime<0,0> { enum {prim=1}; };
struct is_prime<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim = 1};
  void f() { D<2> d = prim; } };
#ifndef LAST
#define LAST 10
#endif

main () {
  Prime_print<LAST> a;
  }
```

Listing 1

Please allow me to put STL aside for now, and discuss a side effect of templates. In 1994, at a C++ committee meeting, the first recorded instance of templates being 'abused' for compile-time programming occurred. The power of C++ templates exceeded the expectations of both its creator and the father of the STL. Erwin Unruh demonstrated the code in Listing 1 [Unruh94]. This is no longer valid C++, and does not work with today's compilers.

A 'modern' working version is given in Listing 2 (overleaf). This code fails to compile in an intriguing way. Some compilers produce error messages like the following (see https://godbolt.org/z/f7zjM6Ysz):

```
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<17>'
…
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<13>'
…
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<11>'
…
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<7>'
…
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<5>'
…
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<3>'
…
unruh.cpp:20:19: error: no viable conversion from 'int' to 'D<2>'
```

Wu Yongwei Having been a programmer and software architect, Yongwei is currently a consultant and trainer on modern C++. He has nearly 30 years' experience in systems programming and architecture in C and C++. His focus is on the C++ language, software architecture, performance tuning, design patterns, and code reuse. He has a programming page at http://wyw.dcweb.cn/, and he can be reached at wuyongwei@gmail.com.

In 1994, at a C++ committee meeting, the first recorded instance of templates being 'abused' for compile-time programming occurred.

```
template <int p, int i> struct is_prime {
  enum {
    prim = (p==2) ||
           (p%i) && is_prime<(i>2?p:0),
                             i-1> :: prim };
};
template<>
struct is_prime<0,0> { enum {prim=1}; };
template<>
struct is_prime<0,1> { enum {prim=1}; };

template <int i> struct D { D(void*); };

template <int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i, i-1>::prim };
  void f() { D<i> d = prim ? 1 : 0; a.f();}
};
template<> struct Prime_print<1> {
  enum {prim=0};
  void f() { D<1> d = prim ? 1 : 0; };
};
int main() {
  Prime_print<18> a;
  a.f();
}
```

Listing 2

Simple examples

Factorial

The previous example was a bit weird and complex. Let's consider a simpler one – factorials. The mathematical definition of factorial can be expressed recursively as:

$$n! = n \cdot (n-1)!$$

$$0! = 1$$

It corresponds to the following C++ code:

```
template <int N>
struct factorial {
  static const int value =
    N * factorial<N - 1>::value;
};

template <>
struct factorial<0> {
  static const int value = 1;
};
```

This almost aligns perfectly.

Let's try some numbers.

- Evaluating `factorial<0>::value` yields 1.
- For `factorial<1>::value`, it computes `1 * factorial<0>::value`, resulting in 1.

```
template <bool Condition,
          typename Then, typename Else>
struct conditional;

template <typename Then, typename Else>
struct conditional<true, Then, Else> {
  using type = Then;
};
template <typename Then, typename Else>
struct conditional<false, Then, Else> {
  using type = Else;
};
```

Listing 3

- For `factorial<2>::value`, it computes `2 * factorial<1>::value`, resulting in 2.
- For `factorial<3>::value`, it computes `3 * factorial<2>::value`, resulting in 6.

And so on.

This approach resembles functional programming, which is atypical for C++:

1. A class template represents a 'function'.
2. Instantiating a template is akin to a 'function call', with a unique result and no side effects.
3. A 'variable' (the static member variable in the class template) can be assigned once and not modified later.
4. The result of instantiation is remembered during subsequent compilation, similar to memoization.
5. …

Conditionals and loops

Similarly, we can create a generic compile-time conditional construct with class templates and specialization (Listing 3). A compile-time loop is more complex (Listing 4, on opposite page).

When the result of a class template equals a similarly named result on the right side of an expression, we can express it more succinctly with inheritance (Listing 5, on opposite page).

In template metaprogramming, using direct recursion typically enhances readability. Using the `loop` construct, as shown in Listing 5, might make the result harder to understand (Listing 6, on opposite page).

Regardless of how it is written, we can obtain the result directly at compile time. For code like the following:

```
printf("%d\n", factorial<10>::value);
```

The value `3628800` will appear in the assembly code, with no trace of `factorial` at all (for details, see https://godbolt.org/z/j59ErdnTj and https://godbolt.org/z/cxdfn97x6).

In template metaprogramming, using direct recursion typically enhances readability. Using the loop construct might make the result harder to understand

```
template <bool Condition, typename Body>
struct loop_result;

template <typename Body>
struct loop_result<true, Body> {
  using type = typename loop_result<
    Body::next_type::condition,
    typename Body::next_type>::type;
};
template <typename Body>
struct loop_result<false, Body> {
  using type = typename Body::type;
};
template <typename Body>
struct loop {
  using type =
    typename loop_result<Body::condition,
                         Body>::type;
};
```

Listing 4

In the examples above, we follow a C++ standard library convention: using a class template's member type **`type`** for a result type and its static member variable **`value`** for a result value.

Prime sieve example

Functional programming in template metaprogramming

Let's return to prime numbers, without showing the result in error messages, which is not that interesting. We will compute at compile time a list of primes, which can be used at run time as well. To do this, we need some tools.

First, we need a tool to convert between *values* and *types*. The standard library provides this functionality, which looks like this:

```
template <typename T, T Val>
struct integral_constant {
  static const T value = Val;is_nonstatic
  typedef T value_type;
  typedef integral_constant type;
};
```

```
template <bool Condition, typename Body>
struct loop_result;

template <typename Body>
struct loop_result<true, Body>
  : loop_result<Body::next_type::condition,
                typename Body::next_type> {};

template <typename Body>
struct loop_result<false, Body> : Body {};

template <typename Body>
struct loop
  : loop_result<Body::condition, Body> {};
```

Listing 5

```
template <int N, int Last, int Result>
struct factorial_loop {
  static const bool condition = (N <= Last);
  using type = integral_constant<int, Result>;
  using next_type =
    factorial_loop<N + 1, Last, Result * N>;
};

template <int N>
struct factorial
  : loop<factorial_loop<1, N, 1>>::type {};
```

Listing 6

Although templates can have non-type template parameters, having a fixed type is not flexible. When the value type is not unique, representing a compile-time constant with a type is common. The class template above can represent a constant of any integer type, like **`int`**, **`size_t`**, or **`bool`**. The **`value`** member retrieves the template's value parameter, **`value_type`** retrieves the type parameter, and **`type`** points to itself.

Here are some examples:

```
// Types
integral_constant<int, 42>
integral_constant<bool, true>
// Values
integral_constant<int, 42>::value    // 42
integral_constant<bool, true>::value // true
```

Next, we need a tool similar to a list in functional programming. I'll use a functional, C++98-compatible definition:

```
struct nil {};
template <typename Head, typename Tail = nil>
struct list {};
```

Those familiar with functional programming will recognize the pattern immediately. For others, you can think of it as a singly linked list:

```
struct list {
  any head;
  list* tail{nullptr};
};
```

Our algorithm for finding prime numbers is a simple sieve, represented in Haskell as:

```
primesTo n = sieve [2..n]
             where
             sieve (x:xs) =
               x:(sieve $
                  filter (\a -> a 'mod' x /= 0)
                         xs)
             sieve [] = []
```

We aim to generate a list from 2 to *n*, then perform the following operations:

1. Take the first element.
2. Filter out elements divisible by it from the remaining list.

After seeing the power of template metaprogramming, you might naturally wonder: How much can be done with template metaprogramming?

```
template <template <typename> class Pred,
          typename List>
struct filter;

template <template <typename> class Pred,
          typename Head, typename Tail>
struct filter<Pred, list<Head, Tail>> {
  typedef typename conditional<
    Pred<Head>::value,
    list<Head,
         typename filter<Pred, Tail>::type>,
    typename filter<Pred, Tail>::type>::type
    type;
};

template <template <typename> class Pred>
struct filter<Pred, nil> {
  typedef nil type;
};
```

Listing 7

3. Recursively apply the sieve to the rest and combine the result with the first element.
4. Recursion stops at an empty list, yielding an empty list.

Currently, we don't have a 'filter' in our toolbox. It's defined in Listing 7. This template is a bit complex, so let me explain briefly.

In the first section, we declare the 'prototype' of this template. It has two template parameters: the first is a class template used as the 'predicate' for filtering, and the second is a normal type representing the list to filter.

In the second section, we implement the main 'overload' of this 'metafunction' with partial specialization. We require the second parameter to match the form `list<Head, Tail>`, naturally separating the 'head' and 'tail' of the list. We then apply the predicate to the 'head' to check the condition. If the condition is met, our result `type` is `Head` concatenated with the filtered result of `Tail`; otherwise, `Head` is discarded, and the result `type` is just the filtered result of `Tail`.

The third section is the recursion termination condition for this 'metafunction'. When we reach `nil`, i.e. the list is empty, the result `type` is also this `nil` marker, representing an empty list.

Similarly, we need a tool to generate sequences:

```
template <int First, int Last>
struct range {
  typedef list<
    integral_constant<int, First>,
    typename range<First + 1, Last>::type>
    type;
};

template <int Last>
struct range<Last, Last> {
  typedef nil type;
};
```

```
template <typename T>
struct sieve_prime;

template <typename Head, typename Tail>
struct sieve_prime<list<Head, Tail>> {
  template <typename T>
  struct is_not_divisible
    : integral_constant<
        bool, (T::value % Head::value) != 0> {};

  typedef list<
    Head,
    typename sieve_prime<typename filter<
      is_not_divisible, Tail>::type>::type>
    type;
};

template <>
struct sieve_prime<nil> {
  typedef nil type;
};
```

Listing 8

We can now write down the metaprogramming algorithm to find primes (Listing 8).

If we want the code to compiler under C++98, the intuitive alias templates can't be used. But we can simulate generating the final result type with inheritance:

```
template <int N>
struct primes_to
  : sieve_prime<
      typename range<2, N + 1>::type>::type {};
```

You can output or further process this result type as needed. More details are available in my online code at https://godbolt.org/z/xE7MKca4s.

After seeing the power of template metaprogramming, you might naturally wonder: *How much can be done with template metaprogramming?*

The answer, as you might guess, is almost *anything*. C++ templates are Turing complete [Veldhuizen03], meaning you can theoretically perform any computation through template metaprogramming.

Of course, there's always a difference between theory and practice. There are things we don't want or can't do at compile time. Even if we want to and can, some tasks aren't convenient with template metaprogramming.

Template metaprogramming doesn't allow any side effects. We can't have input/output access in template code. It can only handle types and compile-time constants. Even if you could access input/output, it wouldn't make sense: we want to display a user interface, accept user input, and read/write databases when the program is running – not when it's compiled.

As mentioned earlier, template metaprogramming is a form of 'functional' programming. I have done a lot just to generate something equivalent to the Racket/Scheme code in Listing 9 (opposite).

Besides the fact that many algorithms are better suited to imperative styles, template metaprogramming is not ideal for compile-time tasks.

```
(define (sieve-prime lst)
  (cond
    [(null? lst) '()]
    [else (let ([n (car lst)])
          (let ([is-not-divisible
                 (lambda (m)
                   (not
                    (= (remainder m n) 0)))])
          (cons n (sieve-prime
                   (filter is-not-divisible
                           (cdr lst)))))))]))

(define (primes-to n)
  (sieve-prime (range 2 (+ n 1))))
```

Listing 9

Template metaprogramming is a primitive form of functional programming, and C++ compilers aren't optimized for it. In fact, the Haskell and Scheme code I have shown runs faster than compiling the previous C++ code. Besides the fact that many algorithms are better suited to imperative styles, template metaprogramming is *not* ideal for compile-time tasks. We can easily stress C++ compilers to their limits, and complicated template metaprograms may work on one compiler and fail on another. Additionally, poorly written template code can even crash the compiler or system, as issues that occur at run time may arise at compile time now. Due to the undecidability of the halting problem, compilers just can't catch every issue in compile-time code. ☺

constexpr

Now let's discuss the new **constexpr** feature introduced in C++11, which allows compile-time programming without using template metaprogramming.

First, while the code I wrote earlier works, there is a minor syntax issue. In the following code (see https://godbolt.org/z/TW4GTKPYc):

```
template <typename T>
void print_value(const T& value)
{
  …
}
print_value(factorial<10>::value);
```

the final link step will fail, which can be frustrating. In the era of C++98, the workaround was to use **enum** instead of **static const int**, an inelegant hack. The proper 'modern C++' solution is **constexpr**. Apart from the C++17 improvement on definition rules, a **constexpr** 'variable' explicitly indicates a compile-time constant. This is a key use of **constexpr** (https://godbolt.org/z/c3Gh13eWb).

Another key use of **constexpr** is in **constexpr** functions. Here is a recursive version that works in C++11:

```
constexpr int factorial(int n)
{
  return n == 0 ? 1 : n * factorial(n - 1);
}
```

In C++14, one can write a more conventional iterative version (see https://godbolt.org/z/3PE43PTn4):

```
constexpr int factorial(int n)
{
  int result = 1;
  for (int i = 2; i <= n; ++i)
    result *= i;
  return result;
}
```

C++17 further loosened restrictions, allowing many functions, including most member functions of **array**, to be marked **constexpr** for compile-time use. I would like to emphasize that **array** is an important compile-time tool, as we can get the size of this container and access all its elements at compile time.

Now we can really try the standard sieve algorithm:

```
template <int N>
constexpr auto sieve_prime()
{
  array<bool, N + 1> sieve{};
  for (int i = 2; i <= N; ++i)
    sieve[i] = true;
  for (int p = 2; p * p <= N; p++)
    if (sieve[p])
      for (int i = p * p; i <= N; i += p)
        sieve[i] = false;
  return sieve;
}
```

This function generates a sieve. To find the number of primes up to **N**, we need to count them. In C++20, we can use the standard **count** algorithm, but, in earlier versions, we need to implement it ourselves:

```
template <size_t N>
constexpr size_t
prime_count(const array<bool, N>& sieve)
{
  size_t count = 0;
  for (size_t i = 2; i < sieve.size(); ++i)
    if (sieve[i])
      ++count;
  return count;
}
```

Converting the final result into an **array** is now simple:

```
template <int N>
constexpr auto get_prime_array()
{
  constexpr auto sieve = sieve_prime<N>();
  array<int, prime_count(sieve)> result{};
  for (size_t i = 2, j = 0; i < sieve.size();
       ++i)
    if (sieve[i]) {
      result[j] = i;
      ++j;
    }
  return result;
}
```

See also https://godbolt.org/z/hjezWEf7v.

...regardless of the number or types of arguments, the compiler can expand and usually inline them. This simplifies repetitive code and offers new automation possibilities...

For almost any C++ programmer, this kind of code is likely easier to understand than template metaprogramming. It is also easier for compilers. MSVC and Apple Clang failed on my template metaprogramming code for calculating primes with an `N` of 1000, while GCC took over a second to compile it. In contrast, the above code handles `N` of 10000 happily across all three compilers. GCC compiles it in just 0.7 seconds for `N` at 10000. Compiling the template code for `N` at 10000 with GCC takes over two minutes and uses several gigabytes of memory. The time complexity of template metaprogramming is not linear.

The only 'unnatural' part is that `N` cannot be passed as a regular function parameter, but only as a template parameter. Even in `constexpr` or `consteval` (C++20) functions, function parameters aren't considered compile-time constants and can't be used where compile-time constants are required.

For compile-time evaluation, the `array` variable must be initialized immediately upon declaration to avoid indeterminate values. As of C++20, this requirement is relaxed; the array still can't contain indeterminate values, but you can declare it without `{}` and initialize it later.

C++20 brings significant improvements for compile-time programming, such as:

1. Using `vector` and `string` at compile time
2. Using strings and custom-type objects as template parameters

These features offer greater flexibility, removing the need to pass lengths as template parameters. However, there is a big limitation: the `vector` or `string` results can't be directly used at run time. I won't go into detail, but interested readers can check out my online code example at https://godbolt.org/z/6c833fE4r.

Variadic templates

Another major improvement in compile-time programming is variadic templates, introduced in C++11 and enhanced in later versions.

Variadic templates have two main uses:

- Forwarding a variable number of arguments to other functions, often with forwarding references
- Iterating over arguments using recursion or fold expressions

The second use is particularly important for compile-time programming. Here is a simple example to check if any provided arguments are null pointers:

```
template <typename... Args>
constexpr bool is_any_null(const Args&... args)
{
  return (... || (args == nullptr));
}
```

The significance of this approach is that, regardless of the number or types of arguments, the compiler can expand and usually inline them. This simplifies repetitive code and offers new automation possibilities.

```
template <typename T, typename F, size_t... Is>
constexpr void
for_each_impl(T&& obj, F&& f,
              std::index_sequence<Is...>)
{
  using DT = std::decay_t<T>;
  (void(std::forward<F>(f)(
     typename DT::template _field<T, Is>(obj)
       .name(),
     typename DT::template _field<T, Is>(obj)
       .value())),
   ...);
}
template <
  typename T, typename F,
  std::enable_if_t<
    is_reflected_struct_v<std::decay_t<T>>,
    int> = 0>
constexpr void for_each(T&& obj, F&& f)
{
  using DT = std::decay_t<T>;
  for_each_impl(
    std::forward<T>(obj), std::forward<F>(f),
    std::make_index_sequence<DT::_size>{});
}
```

Listing 10

By using macro techniques, we can inject metadata into `structs`. Then, with fold expressions and compile-time programming, we can achieve static reflection capabilities (see Mozi [Mozi] for a complete implementation). For instance, Listing 10 is a generic function to iterate over `struct` fields.

With tools like `for_each`, implementing more features becomes easy. For example, the function in Listing 11 can be used to print all fields in a `struct`. (See https://godbolt.org/z/saejMW6Pq.)

```
template <typename T>
void print(const T& obj,
           std::ostream& os = std::cout,
           const char* fieldName = "",
           int depth = 0)
{
  if constexpr (is_reflected_struct_v<T>) {
    os << indent(depth) << fieldName
       << (*fieldName ? ": {\n" : "{\n");
    for_each(
      obj, [depth, &os](const char* fieldName,
                        const auto& value) {
        print(value, os, fieldName, depth + 1);
      });
    os << indent(depth) << "}"
       << (depth == 0 ? "\n" : ",\n");
  } else {
    os << indent(depth) << fieldName << ": "
       << obj << ",\n";
  }
}
```

Listing 11

Writing such code isn't easy, but the result is impressive, with performance far exceeding reflection features in languages like Java.

This function uses the `for_each` function, generic lambda expressions, and compile-time conditional statements (`if constexpr`) to 'iterate' over different fields of a reflected struct, which can have different types. It can correctly handle nested reflected `struct`s.

With a similar method, we can generically implement comparison, copying, and other functions for reflected `struct`s. Writing such code isn't easy, but the result is impressive, with performance far exceeding reflection features in languages like Java. This is because the compiler can statically expand the `struct` when compiling functions like `for_each`, making the generated code equivalent to manually written code that processes each field individually!

Static reflection under standardization

The code works, but we must use special macros to define `struct`s for it to function. With standardized static reflection, we'd be able to write a generic `print` function template (see Listing 12) without special definition forms or the `for_each` function (using the syntax from P2996 [P2996r5]).

While there is an experimental implementation for P2996 [clang], an online implementation of an early proposal, P2320 [P2320r0], is conveniently available in Compiler Explorer. And I'll use it to demonstrate `print` actually works at https://cppx.godbolt.org/z/c7a4jfo74.

Here are some key points (notice that the syntax is subject to changes):

- `^T` is the proposed reflection syntax to get a compile-time reflection object.

```
template <typename T>
void print(const T& obj,
           ostream& os = cout,
           const char* name = "", int depth = 0)
{
  if constexpr (is_class_v<T>) {
    os << indent(depth) << name
       << (*name ? ": {\n" : "{\n");
    template for (constexpr meta::info member :
        meta::nonstatic_data_members_of(^T)) {
      print(obj.[:member:], os,
            meta::name_of(member),
            depth + 1);
    }
    os << indent(depth) << "}"
       << (depth == 0 ? "\n" : ",\n");
  } else {
    os << indent(depth) << name << ": " << obj
       << ",\n";
  }
}
```

Listing 12

- `[:expr:]` reverses the reflection, converting it back to a C++ type or expression; `[:^T:]` gets us back `T`.
- `template for` is a compile-time loop for iterating over objects during compilation [P1306R2], eliminating the need for generic lambdas and `for_each`.
- The `std::meta` namespace provides tools for compile-time processing:
  - `info` is a general reflection object.
  - `members_of` retrieves all members of a type.
  - `nonstatic_data_members_of` extracts non-static data members.
  - `name_of` gets the member's name.

Put together, the outcome is:

- For class types (including `struct`s), it outputs a left brace, recursively calls `print` for all members, and outputs a right brace.
- Otherwise, it simply outputs the field name and the content of non-static data members.

Unfortunately, this will only be available in C++26 (if not later). ■

References

[clang] clang-p2996: https://github.com/bloomberg/clang-p2996

[Ellis90] Margaret A. Ellis and Bjarne Stroustrup (1990) *The Annotated C++ Reference Manual*, Addison-Wesley.

[Mozi] Mozi: https://github.com/adah1972/mozi

[P1306R2] Andrew Sutton et al., 'Expansion statements (revision 2)', May 2024, http://wg21.link/p1306r2

[P2320r0] Andrew Sutton et al., 'The Syntax of Static Reflection', 2021, http://wg21.link/p2320r0

[P2996r5] Wyatt Childers et al., 'Reflection for C++26' (revision 5), August 2024, http://wg21.link/p2996r5

[Stroustrup88] Bjarne Stroustrup, 'Parameterized Types for C++', October 1988, https://www.usenix.org/legacy/publications/compsystems/1989/win_stroustrup.pdf

[Stroustrup94] Bjarne Stroustrup (1994) *The Design and Evolution of C++*, Addison-Wesley.

[Unruh94] Erwin Unruh, 'Primzahlen', 1994. http://www.erwin-unruh.de/primorig.html

[Veldhuizen03] Todd L. Veldhuizen, 'C++ Templates are Turing Complete', 2003, https://www.researchgate.net/publication/2475343_C_Templates_are_Turing_Complete

accu.org

accu.org

accu.org

accu.org

accu.org

The Publish Pattern

How do you minimise locking between producers and consumers? Lucian Radu Teodorescu describes a common, but currently undocumented, design pattern.

Design patterns can help us reason about code. They are like algorithms that are vaguely defined in the code. Once we recognise a pattern, we can easily draw conclusions about the behaviour of the code without looking at all the parts. Patterns also help us when designing software; they are known solutions to common problems.[1]

In this article, we describe a concurrency pattern that can't be found directly in any listing of concurrency patterns, and yet, it appears (in one way or another) in many codebases. It is useful when we have producers and consumers that run continuously, and we want to minimise the locking between them.

Problem description

Let's say we have an open-world game. As the player walks through the world, we load the data corresponding to the regions around the player. We have two types of workloads in our scenario: one for loading the data and another for displaying the loaded data. For the sake of our discussion, let's say that each of these two activities is bound to a thread.

The problem we are trying to solve is how to structure the passing of data (let's call this *document*) from the loading thread (producer) to the rendering thread (consumer). This is analogous to the classical producer-consumer problem [Wikipedia-1], but it has some interesting twists. Let's try to outline the requirements of the problem:

- (R1) The producer is constantly producing new versions of the *document*.
- (R2) The consumer is constantly consuming document data.
- (R3) The consumer will use the latest version of the document.

Please note that we are discussing multiple versions of the document. In our example, the loading thread will produce different documents depending on the position of the player, and the rendering thread will display the latest document, corresponding to the player's most recent position.

A naïve implementation for this problem might look like the code in Listing 1. Here, we have two functions, `produce()` and `consume()`, that are called multiple times from the appropriate threads. They both work on a shared document, `current_document`, so they need something to protect concurrent access to this object. We use a mutex to ensure that we are not reading the document while modifying it.[2]

```
struct document_t {...};

void produce_unprotected(document_t& d);
void consume_unprotected(const document_t& d);

document_t current_document;
std::mutex document_bottleneck;

void produce() {
  std::lock_guard<std::mutex>
    lock{document_bottleneck};
  produce_unprotected(current_document);
}
void consume() {
  std::lock_guard<std::mutex>
    lock{document_bottleneck};
  consume_unprotected(current_document);
}
```

Listing 1

This implementation meets our requirements, but it likely has performance issues. If producing the document takes a long time (which is often the case when loading data from disk), then rendering might be blocked until the operation completes. This is completely unacceptable for this type of application. Moreover, considering the potential usage of the data, this is also suboptimal, as we are often loading data that is far from the player, thereby blocking rendering for data that we might not even need.

This means we need to add a new requirement:

- (R4) Consuming the data should not be delayed by delays in producing the data.

With this new requirement, the naïve implementation from Listing 1 no longer works. Let's explore how we can improve it.

The Publish Pattern

We need to find a way to decouple the production and consumption of the document. The consumer needs to use the latest version of the *published* document but doesn't necessarily need to wait for the producer to complete if production is currently happening. Similarly, the production of the document doesn't need to wait until the latest version of the document is consumed. This is the essence of what I call *the Publish Pattern*.

Similar to the mutex implementation, we still have three parts:

- The latest *published* document
- A way to update the published document (without blocking the consumers)
- A way to consume the published document (without preventing the publishing of new versions of the document)

The code that implements this might resemble what is shown in Listing 2 (overleaf), and the definition of the `published` template may look like the one in Listing 3. The interaction between the producer and the consumer occurs through a `published_document` object that is

1 When we talk about design patterns, we often emphasise their importance for design, while not giving enough credit to how we read and reason about the code. As we reason about the code more often than we design it, maybe we should shift the focus. Just a thought…

2 Every mutex is a bottleneck; by design, it prevents a thread that has resources from doing meaningful work.

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

We need to find a way to decouple the production and consumption of the document... This is the essence of what I call the Publish Pattern.

```
struct document_t {...};

void produce_unprotected(document_t& d);
void consume_unprotected(const document_t& d);

published<document_t> published_document;

void produce() {
  // Make a copy of the currently published
  // document.
  document_t new_version(
    *published_document.get());
  // Perform the needed updates.
  produce_unprotected(new_version);
  // Publish the new version.
  published_document.publish(
    std::move(new_version));
}
void consume() {
  auto current = published_document.get();
  consume_unprotected(*current);
  // The current document is kept alive until
  // the end of the function.
}
```

Listing 2

thread-safe. The producer takes the current document, makes a copy of it, and then edits this copy; when the new document is ready, it is published to the shared `published_document` object. The consumer simply takes the latest version from `published_document` (which is a `std::shared_ptr<const document_t>`) and uses the object freely.

The `published` template provides two basic thread-safe operations on an inner `std::shared_ptr<const T>` object. The first operation is to update the content of this object, and the second is to simply return a copy of the shared pointer.

The key element is the internal type: `std::shared_ptr<const T>`. Because the inner object of the shared pointer is `const`, nobody can change it after it's created.

This means that copies of the `std::shared_ptr<const T>` object can be passed around and safely shared across threads; we don't have race conditions on the inner object.

When we want to update the document, we update just the master copy of the object, the one embedded in the `published<T>` class; we do this under a lock to prevent copies being made while we update the master copy. All the copies we've already made are still valid, and they don't become invalidated by updating the master copy.

This means that the consumers, on whichever threads they run, can continue to use their versions of the document regardless of how many times we publish a new document. Thus, we fulfil all four requirements we had.

```
template <typename T>
class published {
public:
  void publish(T&& doc) {
    std::lock_guard<std::mutex>
      lock(small_bottleneck_);
    published_document_ =
      std::make_shared<const T>(std::move(doc));
  }

  std::shared_ptr<const T> get() {
    std::lock_guard<std::mutex>
      lock(small_bottleneck_);
    return published_document_;
  }
private:
  std::shared_ptr<const T> published_document_;
  std::mutex small_bottleneck_;
};
```

Listing 3

Analysis

Performance

For most problems, using this pattern is relatively cheap. We have just 2, potentially 3, heavy operations:

- a memory allocation each time we publish a new document
- small synchronisation when we publish and when we get the latest version of the document
- (potentially) copying of the document

If the producing and the consuming operation are expensive (which is the main use case for this pattern), then a memory allocation for publishing a new document should be a very small cost.

Similarly, the contention on accessing the latest published document inside the `published<T>` class should be very small. We are doing very little inside the lock, and the lock is taken infrequently (twice while producing and once while consuming). Depending on the platform, the lock can be improved by using a spin-lock, but I would argue that for most cases, this won't be necessary; anyways, we should measure first.

Finally, we are making a copy of the document. Depending on the document type, this may be an expensive operation. If the documents are hard to copy, there may be ways of improving the copying type. We will provide some tips later.

Thread safety

An instance of `published<T>` can be safely shared across threads and called from multiple threads. The content of the `std::shared_ptr<const T>` objects obtained from a `published<T>` can also be freely used across different threads. The inner objects are marked as `const`, so we can only have read-only access

to them. In the absence of a writer to this object, there is no data race when accessing it.

This immediately implies that we can have multiple consumers running at the same time. They can all view the same document (i.e., the latest version), or they may be viewing different versions of the document if there was at least one `publish` call during the execution of a consumer job.

Strictly speaking, from a thread safety perspective, it's also acceptable to have multiple producers running at the same time. They may overwrite the documents that the other producers generate, but this doesn't pose a thread safety issue.

This approach manages to retain thread safety while dramatically reducing the scope of the locks.

Functionality

Let's now analyse the proposed pattern from a functional point of view. One can easily see that requirements (R1) and (R2) are fulfilled. Nothing prevents the two threads from continuously producing and consuming documents. Requirement (R4) is also satisfied, as the consumers are not affected by the duration of the producer operation. If producing the document takes longer, consumers will see new versions of the document less frequently, but they are not prevented from running at the same pace.

Requirement (R3) is slightly more interesting. When a consumer starts its job, the first thing it does is retrieve the latest published version of the document. From this perspective, we can say that (R3) is satisfied. But this is not the complete story. During the consumer's work, the producer may publish a new version of the document; it's even possible for more than one version of the document to be published. That is, the fact that when the consumer job started, it had the latest version of the published document doesn't mean that by the time the job is finished, it still has the latest version of the document. This is an important functional aspect that the user needs to account for.

On the other hand, this technique ensures that once we start consuming a version of the document, the inner parts of the document don't change. We get 100% consistency when consuming the document. This technique linearises the producer and consumer operations [Wikipedia-2].

From a safety perspective, we can have multiple producers running at the same time. But from a functional perspective, that would be a bug. The problem is that one producer may completely overwrite the changes made by another producer. There are ways to fix this, but this is outside the pattern laid out by Listing 2.

Memory consumption

One of the things that the user may need to pay attention to is memory (and, in general, resource) consumption. If we have many long-running consumers, we may end up with multiple versions of the document. If the documents consume a significant amount of memory, this might be a problem.

The Publish pattern doesn't guarantee an upper bound to the number of documents that may be in flight at a given time.

This is a downside of having the possibility of multiple independent consumers, which also run independently from the producer(s). If we had used locking, we would implicitly guarantee that there is only one version of the document available.

Variants and improvements

The term *the Publish pattern* is something I coined to describe a behaviour that I implemented to solve a problem similar to the one described here (at that time, I was working on navigation software). The idea seemed to be applicable in multiple contexts; indeed, since then, I've applied it multiple times in various contexts. I took inspiration for this technique from a presentation by Herb Sutter on implementing atomic linked lists, in which he advocated for `std::atomic<std::shared_ptr<T>>` [Sutter14].

In essence, this pattern is a form of double-buffering or, more generally, multiple-buffering [Wikipedia-3]. One can use the Publish pattern to implement double-buffering; conversely, one can use double-buffering techniques in places where the Publish pattern could be applied. Compared to the typical double-buffering technique, I appreciate the simplicity and generality of the Publish pattern.

From a different perspective, we can consider this to be a variant of the read-copy-update (RCU) synchronisation mechanism [Wikipedia-4]. While RCU is a low-level technique, the Publish pattern is something that we can apply to complex documents.

Handling multiple producers

As previously mentioned, the code from Listing 2 doesn't work if we have multiple producers running at the same time. One of the producers might overwrite the data produced by the others.

To solve this, we may employ an optimistic locking scheme [Wikipedia-5]. We publish a new version of the document only if a new version hasn't been published while we were computing the new version to publish. Otherwise, we can retry the publish operation.

This can be implemented with code similar to Listing 4. When trying to publish, we check if nothing else has been published in the meantime; if we have the same document, we can publish directly. Otherwise, we get the latest version of the document, reapply the updates, and retry.

One downside of this algorithm is that we don't have an upper bound on how many times we can retry publishing a document. If applying some updates takes a long time, we may constantly get new versions of the document published in the meantime, which would keep the retry process running endlessly. As the reader might have guessed, there are techniques for detecting long retry chains and failing the entire update after a certain number of retries. One can imagine more complex strategies for dealing with these cases, but that is outside the scope of this article.

Merging the latest version of the document with the changes local to the producer may not be a trivial task, but it's doable.

The bottom line is that the Publish pattern can be easily adapted to work in contexts where multiple producers are needed.

Monotonic updates

My friend, Dimi Racordon, wanted to speed up the Hylo compiler [Hylo] by introducing concurrency into the type checker. Oversimplifying, the activity of a type checker can be seen as building a graph (for the most part, it's a tree) and evaluating the validity of each node. Different declarations and type evaluations will typically generate different nodes in the graph. Many nodes in the graph can be processed in parallel, even if sometimes they need to intersect (e.g., different declarations depend on the same type).

The implementation that Dimi chose for speeding up the type checker closely resembles the Publish pattern with multiple producers. A producer type-checks a set of nodes, and after a while, publishes the results obtained so far, so that others can see the progress. However, in her case, the way the document is updated has a special property: all the updates are monotonic. That is, once we add some information to the document, we can never remove it.

Having guaranteed monotonic updates may make merging the documents much easier. All the updates to the documents are additions, so merging involves placing our additions on top of those in the base document.

This strategy may allow us to break the producer's job into two parts: determining additions and committing the additions. This may look like the code from Listing 5. This approach reduces the lifetime of the current version of the document; after all, the main activity is producing the updates. This implies that the chances of other documents being produced while we apply the updates are reduced. In this way, we reduce the overall time needed to publish a new version of the document.

```
template <typename T> class published {
public:
  bool try_publish(std::shared_ptr<const T>& old,
  T&& doc) {
    std::lock_guard<std::mutex>
      lock(small_bottleneck_);
    if (published_document_ != old) {
      // tell the caller which is the current
      // document
      old = published_document_;
      return false;
    }
    published_document_ =
      std::make_shared<const T>(std::move(doc));
    return true;
  }
  ...
};

void apply_updates(const document_t& base,
  document_t& doc);

void produce_with_retry() {
  auto current = published_document.get();
  // Make a copy of the currently published
  // document.
  document_t new_version(*current);
  // Perform the needed updates.
  produce_unprotected(new_version);
  // Publish the new version.
  while (!published_document.try_publish(current,
         std::move(new_version))) {
    // 'current' is now the latest version;
    // reapply the updates, and retry.
    apply_updates(*current, new_version);
  }
}
```

Listing 4

Large documents

We mentioned earlier that the Publish pattern may not be very efficient when dealing with large documents, as producing a new version of the document requires copying it. The copying process can take significant time and may lead to high memory consumption.

Two techniques come to mind that we can apply to improve this:

- Persistent data structures [Wikipedia-6]
- Using aggregation instead of composition for document subparts and sharing the subparts between document versions.

In the first case, persistent data structures allow us to reuse the previous version of the document while adding new information to it (or removing information from it). For example, adding an element to the front of a list doesn't change any of the elements already existing. Thus, we can make the new list share data with the previous list. The act of copying the document is cheap, as the subparts of the document are not actually copied. Tree structures are also good examples of persistent data structures.

The second technique depends on the type of document, so let's describe it with an example. Let's say that the document is defined as

```
using document_t =
  std::unordered_map<location_t, tile_data_t>
```

(we load a tile's worth of data for the positions around the player). Let's also assume that, once tile data is loaded for a location, it will never change. Copying `tile_data_t` objects might be expensive, so copying the entire document is expensive. However, if we change the document to be

```
std::unordered_map<location_t,
  std::shared_ptr<const tile_data_t>>
```

then copying the document is cheap. We are not copying the tile data; we are just copying the map. Multiple document versions will share the same tile data subobjects.

```
void produce_monotonic() {
  // Produce the additions
  auto additions = produce_additions();
  // Publish a new version of the document.
  auto current = published_document.get();
  while (true) {
    document_t new_version(*current);
    update_document(new_version, additions);
    if (published_document.try_publish(current,
        std::move(new_version)))
      break;
  }
}
```

Listing 5

Conclusions

The Publish pattern can be a valuable tool in one's toolkit. It can be used in many scenarios where we want to reduce contention between threads and have a producer-consumer type of problem.

The pattern works well when there are multiple consumers of the document. In its basic form, the pattern works with only one producer, but it can be easily extended to accommodate multiple producers.

The pattern is easy to implement and can be very efficient, depending on the problem. However, like any software solution, it has downsides; in this case, extra memory allocations and copying the document. Being aware of these downsides helps users tailor the pattern to the problem at hand.

I hope you find this pattern as useful as I did. Please let me know about your experience with it. ■

References

[Hylo] The Hylo Programming Language, https://www.hylo-lang.org/

[Sutter14] Herb Sutter, Lock-Free Programming (or, Juggling Razor Blades), Part II, CppCon 2014, https://youtu.be/CmxkPChOcvw

[Wikipedia-1] 'Producer–consumer problem': https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem, accessed August 2024.

[Wikipedia-2] 'Linearizability': https://en.wikipedia.org/wiki/Linearizability, accessed August 2024.

[Wikipedia-3] 'Multiple buffering': https://en.wikipedia.org/wiki/Multiple_buffering, accessed August 2024.

[Wikipedia-4] 'Read-copy-update': https://en.wikipedia.org/wiki/Read-copy-update, accessed August 2024.

[Wikipedia-5] 'Optimistic concurrency control': https://en.wikipedia.org/wiki/Optimistic_concurrency_control, accessed August 2024.

[Wikipedia-6] 'Persistent data structure': https://en.wikipedia.org/wiki/Persistent_data_structure, accessed August 2024.

Modernization of Legacy Arrays: Replacing CArray with std::vector

Many codebases still use mature libraries, such as Microsoft Foundation Classes. Stuart Bergen explains how and why he moved to using modern standard C++ tools instead.

Our scientific desktop application relies on the Microsoft Foundation Class (MFC) library for both user interface (UI) and computational tasks. MFC, introduced in 1992, encapsulates portions of the Windows API within object-oriented C++ classes and is currently maintained [Wikipedia-1]. Despite its age, our users enjoy the interactive, straightforward and visually appealing experience that MFC provides. As a result, there has been little incentive to replace complex UI-related code with alternatives.

Computational tasks using the MFC library show signs of aging compared to standard C++ containers. The primary MFC container class, **`CArray`**, closely resembles **`vector`** in functionality [MS-1], [CPP-1]. Both offer C-like arrays that dynamically grow and shrink, reserve contiguous space on the heap, and use zero-based indexing. However, a key difference is that **`CArray`** lacks iterator support. We previously used **`boost::iterator_facade`** [Abrahams06] to supplement basic iterator support, but this only goes so far. The codebase was updated to prepare for future development and embrace modern practices, including the adoption of C++20 ranges. This effort resulted in simpler, more familiar code, performance and scalability improvements, and an enhanced debugging experience.

In this article, methods to replace the MFC container class **`CArray`** with **`vector`** are proposed. These practical techniques emerged from a real-world refactoring effort on a commercial software project. First, we present class method conversion techniques suitable for direct substitution. In cases where direct substitutions are not feasible, standalone replacement functions are offered. Next, we consider the array index, public inheritance of the **`CArray`** class, array resize, array length, and MFC use of **`CArray`**. Finally, we suggest further modernizations and draw conclusions.

Class method conversions

The modernization effort mainly focused on replacing **`CArray`** class methods and operators with their **`vector`** equivalents. Table 1 (overleaf) lists **`vector`** replacements for the public **`CArray`** class interface. The table served as a helpful reference during the refactoring effort. Template arguments are omitted for brevity. *Notation*: **`a`** and **`b`** represent **`CArray`**/**`vector`** objects, **`i`** represents an index, **`n`** and **`m`** represent array lengths or element counts, **`val`** represents an array value, and *italicized* text represents standalone replacement functions. Full method signatures can be found in the **`CArray`** and **`vector`** references [MS-1], [CPP-1]. The **`CArray`** class possesses only a default constructor; its copy constructor and copy assignment operator are deleted.

Standalone replacement functions in Listing 1 (C++17 compiler required) serve as direct replacements when no **`vector`** equivalent is available.

Stuart Bergen Stuart is a software developer with a background in geophysics, finance, and communications systems. He has a PhD specializing in signal processing, and enjoys camping and skiing in the Canadian Rockies. Stuart lives in Calgary, Canada, and can be contacted at stuartbergen@shaw.ca.

```
template <typename T, typename I>
bool ValidAt(const std::vector<T>& a, I i) {
  if constexpr (std::is_unsigned_v<I>)
    return i < a.size();
  else if constexpr (std::is_signed_v<I>)
    return i >= 0 && i < a.size();
  else
    return false;
}
template <typename T>
auto Append(std::vector<T>& a,
            const std::vector<T>& b) {
  return a.insert(a.end(), b.begin(), b.end());
}
template <typename T>
auto InsertAt(std::vector<T>& a, std::size_t i,
              const T& val) {
  if (ValidAt(a, i)) return a.insert(a.begin()
      + i, val);
  a.resize(i + 1);
  a.at(i) = val;
  return a.begin() + i;
}
template <typename T>
auto SetAtGrow(std::vector<T>& a, std::size_t i,
               const T& val) {
  if (!ValidAt(a, i)) a.resize(i + 1);
  a.at(i) = val;
  return a.begin() + i;
}
template <typename T>
auto RemoveAt(std::vector<T>& a, std::size_t i,
              std::size_t n = 1) {
  return a.erase(a.begin() + i,
                 a.begin() + i + n);
}
template <typename T>
void SetSize(std::vector<T>& a, std::size_t n,
             std::size_t m) {
  a.reserve(m);
  a.resize(n);
}
template <typename T>
__int64 GetSize(const std::vector<T>& a) {
  return static_cast<__int64>(a.size());
}
```

Listing 1

Function **`ValidAt`** validates an index of any integral type (signed or unsigned). All other standalone functions employ unsigned integral types for indexing, array lengths, and element counts to avoid confusion. If necessary, these functions could be modified to support signed integral types similarly to **`ValidAt`**. The **`RemoveAt`** function does not validate or assert the array index or element count, so use it with caution. Conversely, functions **`InsertAt`** and **`SetGrowAt`** perform index validation removing the need for assertion checks.

Standalone functions return iterators corresponding to their **`vector`** methods. **`CArray`** methods return **`void`**, except for **`CArray::Append`**, which returns an index of signed integral type to the first appended

These practical techniques emerged from a real-world refactoring effort on a commercial software project

CArray	vector	Purpose
`a()`	`a()`	Default constructor
`a.Add(val)`	`a.push_back(val)`	Adds an element to the end of the array; grows the array by 1
`a.Append(b)`	*`Append(a,b)`*	Appends another array to the array; grows the array if necessary
`a.Copy(b)`	`a = b`	Copies another array to the array; grows the array if necessary
`a.ElementAt(i)` `a.GetAt(i)`	`a.at(i)`	Returns a reference to an array element at the specified index
`a.FreeExtra()`	`a.shrink_to_fit()`	Frees all unused memory above the current index upper bound
`a.GetCount()` `a.GetSize()`	`a.size()` *`GetSize(a)`*	Returns the number of elements in the array. `vector` replacements yield unsigned and signed integral types, respectively
`a.GetData()`	`a.data()`	Provides direct access via pointer to the underlying contiguous storage
`a.GetUpperBound()`	*`GetSize(a)-1`*	Returns the largest valid index. Is -1 when array is empty
`a.InsertAt(i,val)`	*`InsertAt(a,i,val)`*	Inserts an element at the specified index; grows the array as needed
`a.IsEmpty()`	`a.empty()`	Determines whether the array is empty
`a.RemoveAll()`	`a.clear()`	Removes all elements from the array
`a.RemoveAt(i,n)`	*`RemoveAt(a,i,n)`*	Removes an element (or multiple elements) at the specific index
`a.SetAt(i,val)`	`a.at(i) = val`	Sets the value for a given index; array not allowed to grow
`a.SetAtGrow(i,val)`	*`SetAtGrow(a,i,val)`*	Sets the value for a given index; grows the array if necessary
`a.SetSize(n)`	`a.resize(n)`	Sets the number of elements in the array; allocates memory if necessary
`a.SetSize(n,m)`	*`SetSize(a,n,m)`*	Sets the number of elements in the array and storage "grow by" factor; allocates memory if necessary
`a[i]`	`a[i]`	Returns a reference to an array element at the specified index

`vector` replacements for the public `CArray` class interface

Table 1

element. Function `Append` returns an iterator pointing to the first appended element.

The `GetSize` function casts the return value of `vector::size` to a 64-bit signed integral value. Its primary use case is for arithmetic operations involving signed integral types. The assumption is that the vector length will not exceed `std::numeric_limits<__int64>::max()`, which equals LLONG_MAX or 9,223,372,036,854,775,807. This is a safe assumption for our desktop application.

Array index

Both container classes assume that indexes are valid (within bounds). `CArray` utilizes a signed integral type for indexes, while `vector` uses an unsigned integral type. For our 64-bit builds, this equates to the types `long long` (or `__int64` on Windows) and `vector::size_type`/`std::size_t` (or `unsigned __int64` on Windows), respectively.

Index refactoring involves replacing the signed integral type `__int64` with the unsigned integral type `unsigned __int64`. The value ranges for these types are [-9.223E18, 9.223E18] and [0, 18.446E18], respectively [MS-2]. The unsigned type can safely represent signed integral values that are non-negative. However, it's essential to handle values less than 0 carefully to avoid the *integer wraparound* (or *overflow*) phenomenon [Wikipedia-2]. For example, when converting a signed integral value of -1 (represented by 32 bits) to an unsigned integral value, we get (2^{32} - 1 = 4,294,967,295).

Array element access without bounds checking is provided by `operator[]` for both classes. This method is commonly used, and no modifications are needed. For access with bounds checking, replace the `CArray::GetAt` method with `vector::at`. Out-of-bounds accesses with `CArray::GetAt` assert for Debug builds, while `vector::at` throws the `std::out_of_range` exception. Catch statements for `std::out_of_range` can be added in specific areas if needed, e.g., error logging.

The `CArray::GetUpperBound` method is problematic because it returns -1 for empty arrays. To avoid negative indexes, this functionality was removed from the codebase. Instead, `vector::empty` was used for empty array checks, and `vector::size` was used for upper bounds calculations. Statements of the form `a[a.GetUpperBound()]` were

When working with indexes, it's advisable to convert them to an unsigned integral type whenever possible...values remain non-negative and this helps avoid potential errors

replaced with `a.back()`. Table 1 provides a signed integral return value, to be used only if absolutely necessary.

When working with indexes, it's advisable to convert them to an unsigned integral type whenever possible. Doing so ensures that values remain non-negative and this helps avoid potential errors that can arise from mixing different types (which we explore later).

Public inheritance of CArray

The `CArray` class publicly inherits from the principal MFC base class, `CObject`, as follows:

```
template <class T, class ARG = const T&>
class CArray : public CObject
```

where `T` specifies the type of objects stored in the array, and `ARG` specifies the argument type used to access objects stored in the array. The base class `CObject` provides services such as serialization support, run-time class information, and dump diagnostics output [MS-3]. `CArray` and `CObject` have virtual destructors that permit "is-a" use cases of the form:

```
template <class T, class ARG = const T&>
class CDerived : public CArray<T, ARG>
```

which were encountered in the codebase. Since `vector` has a non-virtual destructor, it's essential to explore alternative approaches as recommended by Scott Meyers in *Effective C++* [Meyers05].

Our codebase did not use any of the `CObject` services mentioned above, permitting a straightforward refactor using public composition of the form:

```
template <class T>
struct CDerived {
  std::vector<T> v;
}
```

Code is modified by adding a `v` or `.v` to provide array access depending on the context. Some use cases arguably provide improved readability, such as transforming:

```
s += (*this)[i].GetString();
```

into:

```
s += v[i].GetString();
```

The following refactor can be used when `CObject` services are employed:

```
template <class T>
struct CDerived : public CObject {
  std::vector<T> v;
}
```

I would like to mention the StackOverflow post 'Thou shalt not inherit from `std::vector`' [StackOverflow]. It is worth reading and considering the various perspectives. There are recommendations for both public and private inheritance with caveats (no new data members), along with nuanced discussions about undefined behaviour. Public composition is a safe choice for our simple use cases [Meyers05].

Array resize

Modernizing the array resizing code involves replacing `CArray::SetSize` with `vector::resize`. From the `CArray` reference:

> Most methods that resize a `CArray` object or add elements to it use `memcpy_s` to move elements. This is a problem because `memcpy_s` is not compatible with any objects that require the constructor to be called. If the items in the `CArray` are not compatible with `memcpy_s`, you must create a new `CArray` of the appropriate size. You must then use `CArray::Copy and CArray::SetAt` to populate the new array because those methods use an assignment operator instead of `memcpy_s`.

Conversely, when `vector` reallocates it first attempts to move objects by calling the object's move constructor. If the move constructor cannot be called (as determined by the utility function `std::move_if_noexcept`), the copy constructor is invoked [CPP-2]. We encountered compilation errors when calling `vector::resize` of the form:

```
error C2280: 'BlockFile::BlockFile(const
BlockFile &)': attempting to reference a deleted
function
```

Here `BlockFile`'s copy constructor is intentionally deleted. Interestingly, switching to `vector` exposed a programming flaw in the original code. `CArray::SetSize` should not have been making copies of `BlockFile` via `memcpy_s`. We were able to precompute the number of `BlockFile` objects needed, enabling the straightforward fix:

```
std::vector<BlockFile> bFile(n);
```

which uses `BlockFile`'s default constructor and maintains the deleted copy constructor.

Array length: mixed signedness issues

Modernizing the array length reporting code involves replacing `CArray::GetSize` with `vector::size` or `GetSize`. It is recommended to use `vector::size` and `GetSize` for unsigned and signed integral types, respectively.

Many potential pitfalls arise from mixing different integral types in arithmetic and binary operations, which can result in unexpected behaviour [Wikipedia-3]. Specifically, the unmodified codebase expects indexes of signed integral type, while `vector::size` returns an unsigned integral type. According to usual arithmetic conversions, operations involving different integral types are performed using a common type [CPP-3]. In the case of signed and unsigned integral types, the unsigned integral type serves as the common type.

In most cases, indexes were always greater than 0, such as in the common `for` loop:

```
for (int i = 0; i < a.size(); i++)
```

Here, the subexpression (`i < a.size()`) works as intended, converting `i` to an unsigned integral type with no wraparound.

Now let's examine some modified statements representative of real-world conditionals found in `if`, `while`, and `for` statements, where `i` is a signed integral type that can assume negative values:

1. `bool bBelowUpper1 = (i < a.size());`
2. `bool bBelowUpper2 = (i <= a.size() - 1);`
3. `bool bAboveLimit = (a.size() > 1);`
4. `bool bInsideRange = (i >= 0 && i < a.size());`
5. `bool bOutsideRange = (i < 0 || i >= a.size());`

Line 1 fails because it exhibits wraparound when `i` is negative. The LHS of `operator<` is converted to an unsigned integral type to match the RHS.

Line 2 fails with two problems. First, the subexpression (`a.size() - 1`) exhibits wraparound when `a.size()` is 0. Second, the LHS of `operator<=` exhibits wraparound when `i` is negative for the same reason as Line 1.

Line 3 appears similar to Line 1, but it's actually fine. This is mentioned for awareness purposes, as these cases tend to look similar after a few hundred instances.

Line 4 works as expected despite wraparound when `i` is negative; the subexpression (`i >= 0`) evaluates false as intended because `i` is not converted to an unsigned integral type, while the subexpression (`i < a.size()`) evaluates false simultaneously due to wraparound. However, for positive `i` both subexpressions works as intended, functioning correctly for large arrays. In fact, this technique is employed in the signed integral branch of `ValidAt`.

Line 5 fails because the subexpression (`i >= a.size()`) evaluates true due to wraparound when `i` is negative, with `operator||` propagating the error. The main point of this discussion is to be extra careful when mixing types. It's easy to become confused with seemingly simple statements.

Switching an index's type isn't always straightforward. If you're dealing with a math-focused codebase where negative and relative indexes play a significant role (e.g., in physics simulations, time series, etc.), altering the type could impact algorithmic calculations. This change might be more complex and time-consuming than initially anticipated.

For scenarios that must honour the original signed integral intent, it is recommended to use `GetSize`:

6. `bool bBelowUpper3 = (i32 < GetSize(a));`
7. `bool bBelowUpper4 = (i64 < GetSize(a));`

Line 6 works as expected because the shorter 32-bit signed integral type on the LHS is upconverted to 64 bits to match the RHS.

Line 7 works as expected because the LHS and RHS types match.

MFC use of CArray

MFC uses `CArray` in a surprisingly limited capacity. We successfully eliminated `CArray` from our math-focused codebase, which employs standard MFC controls for the UI. Searching the MFC include directory for `CArray` yields 128 hits, many of which occur in protected data areas and appear implementation-specific. Nevertheless, there are some public use cases in the following classes: `CArchive`, `CBaseTabbedPane`, `CD2DGeometrySink`, numerous `CMFCRibbon*` classes, and `CTabbedPane`. You might want to reconsider replacing arrays in these cases. Alternatively, conversion methods between `CArray` and `vector` are straightforward and can be reused in testing code.

Further modernizations

Iterator difference types have potential for modernization. The difference type of an iterator [CPP-4] is a contemporary alternative to `std::ptrdiff_t` [CPP-5], allowing negative offsets. This concept applies to iterator types with defined equality. The `std::incrementable_traits` struct computes a difference type for a given type, if it exists [CPP-6].

MFC offers several ready-to-use array classes, such as `CByteArray`, `CDWordArray`, `CObArray`, `CPtrArray`, `CUIntArray`, `CWordArray`, and `CstringArray` [MS-4]. These classes have member functions similar to `CArray` and should also benefit from the proposed replacement methods.

Conclusions

The article explores modernizing legacy arrays, specifically proposing practical techniques to replace the MFC container class `CArray` with `vector`. It begins with class method conversion techniques suitable for direct substitution. When direct substitutions are not feasible, standalone replacement functions are provided. The article offers refactoring guidance for various array operations, including indexing, resizing, and length reporting. It also addresses handling situations involving the public inheritance of `CArray` and provides a description of MFC's use of `CArray`. Additionally, the article discusses working with mixed integral types (signedness) and highlights potential pitfalls with examples. Finally, the article suggests further modernizations and draws conclusions. For those dealing with MFC structures, switching to standard C++ containers like `vector` can simplify the codebase, improve performance and scalability, and enhance the debugging experience. ■

Thanks

Thank you to the anonymous reviewers for their interest and invaluable comments, which greatly improved the quality of this article.

References

[Abrahams06] David Abrahams, Jeremy Siek and Thomas Witt (2003, updated 2006) `boost::iterator_facade`: https://www.boost.org/doc/libs/1_85_0/libs/iterator/doc/iterator_facade.html

[CPP-1] `std::vector`: https://en.cppreference.com/w/cpp/container/vector

[CPP-2] `std::move_if_noexcept`: https://en.cppreference.com/w/cpp/utility/move_if_noexcept

[CPP-3] Usual arithmetic conversions: https://en.cppreference.com/w/cpp/language/usual_arithmetic_conversions

[CPP-4] Iterator library: https://en.cppreference.com/w/cpp/iterator

[CPP-5] `std::ptrdiff_t`: https://en.cppreference.com/w/cpp/types/ptrdiff_t

[CPP-6] `std::incrementable_traits`: https://en.cppreference.com/w/cpp/iterator/incrementable_traits

[Meyers05] Scott Meyers (2005) *Effective C++: 55 specific ways to improve your programs and designs*, Third Edition, Addison-Wesley Professional.

[MS-1] CArray Class: https://learn.microsoft.com/en-us/cpp/mfc/reference/carray-class

[MS-2] Data Type Ranges: https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges

[MS-3] CObject Class: https://learn.microsoft.com/en-us/cpp/mfc/reference/cobject-class

[MS-4] Ready-to-Use Array Classes: https://learn.microsoft.com/en-us/cpp/mfc/ready-to-use-array-classes

[StackOverflow] 'Thou shalt not inherit from `std::vector`': https://stackoverflow.com/questions/4353203/thou-shalt-not-inherit-from-stdvector

[Wikipedia-1] Microsoft Foundation Class Library: https://en.wikipedia.org/wiki/Microsoft_Foundation_Class_Library

[Wikipedia-2] Integer overflow: https://en.wikipedia.org/wiki/Integer_overflow

[Wikipedia-3] Signedness: https://en.wikipedia.org/wiki/Signedness

Afterwood

Many programming books are regarded as classics. Chris Oldwood shares the joy of re-reading some of them.

Now that people are beginning to venture out again, some of those meetups which have been in stasis for the past few years are starting to reappear. I continue to work almost entirely remotely after thankfully being turfed out of the office back in early 2020, and as such need to look closer to home for some in-person programming-related social interaction. Last year, somewhat desperate to get out again and meet like-minded people in-person, I attended a whole bunch of programming conferences, and that did turn out to be incredibly soul filling, but also quite an expensive exercise for a freelancer. (The bar bills were entirely self-inflicted, though.)

One meetup that resurfaced at the tail end of last year has been doing several sessions on practising TDD. Many of the attendees haven't done it before and so my presence is more for the purposes of networking and lending an extra pair of hands. Even though I started practising TDD almost 20 years ago, I wouldn't want to presume I have nothing to learn and, as it's done in pairs, I know I'll always learn something when pairing with a new partner.

I hadn't noticed until a couple of sessions in that there hadn't been any mention of books on the topic. Back in the mid-2000s, when I first became aware of TDD, there were two books which I immediately snapped up – *Test Driven Development: By Example* from Kent Beck and *Test Driven Development: A Practical Guide* from David Astels. It had been a while since pulling either book down from the bookshelf but I remember them both being very useful to me in getting a firm grounding in the practice and so I wondered if their omission was on purpose: perhaps they hadn't stood the test of time?

Hence, before opening my mouth at the next TDD oriented session, I thought I would go back and at least read Beck's book again to see if suggesting it would be useful or not. Many people prefer videos to the written word these days, so the audience for that format was likely to be small, but no video tutorials had been suggested either, so maybe there was a whole conversation about what the modern authoritative sources are? (At the time of writing there hasn't been another TDD-based session to present to, so the answer will have to wait, for now.)

Like many of Beck's books, *TDD by Example* is only a couple of hundred pages long but packed with content. The first part of the book covers the mechanics – the 'by example' bit – while the latter looks at the practice from a patterns perspective to explore the different approaches to writing tests, forces on the design, refactoring, etc. Reading through his money example was an absolute joy, far more than I remember first time around.

What I've always enjoyed about Beck's books is the commentary that goes alongside the main thread. It's in this background commentary where the real meat of the book lives. Even though the example is relatively simple, he is still forced to make numerous decisions about how to tackle the problem, such as how to break the problem down – when to go forward, sideways, or even backwards. He talks about TDD giving you courage and by seeing many of the micro-decisions he's making for each test you get to see how that's possible. One thing the book makes very clear is that the practice is not dogmatic; yes, there is a surrounding structure, but there are different paths depending on whether you already know where you are heading, e.g. Obvious Implementation vs Triangulation.

So, in summary, yes, I believe it totally stands the test of time and deserves the status of being regarded 'a classic'.

In contrast, Dave Astels book tackles the subject in reverse, i.e. concepts followed by a lengthy example. He has far more tooling-specific sections, which has dated it somewhat. However, what his book really achieved was to address the naysayers that exclaimed that you can't write a GUI test-first. Even 20 years later, I still hear that argument and modern UI frameworks are so much more amenable to being test-driven than they were at the turn of the millennium. It also helped me cement the notion of seams that Michael Feathers introduced in *Working Effectively with Legacy Code* as I tried to apply the same ideas to a legacy C++ GUI application written using an old-fashioned framework.

Should I be surprised that re-reading Beck's book was possibly more enjoyable second time around than the first? Probably not, if we consider what Italo Calvino proposed in his article 'Why Read the Classics?' [Calvino86] The article, which is a precursor to his book of the same name, lays out what he considers makes a classic text. Of course, he was talking about real literature, not books on programming, but there are certainly many parallels. Number 4 on his ever-evolving definition of a classic is:

> A classic is a book which with each re-reading offers as much of a sense of discovery as the first reading.

Likewise, Number 6 suggests that I can probably look forward to subsequent readings that will continue to bring pleasure and knowledge:

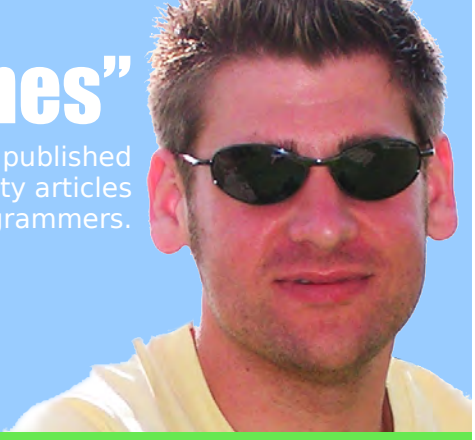
> A classic is a book which has never exhausted all it has to say to its readers.

To date my attitude to reading generally sees me only make a single pass through a book. From that point on I may refer to it again, perhaps numerous times, but usually only to remind myself of specific points, such as when citing it as part of my own writing. What I never do is go back and read it again from scratch with the intention of unearthing new perspectives. Time feels too short, and the corpus of programming works too large to dwell repeatedly on the same texts, but that approach now feels decidedly transactional. Even for some technical books I feel I'm missing out on the opportunity to take the journey again, only older and wiser, but hopefully not complacent.

References

[Calvino86] Italo Calvino (1986) 'Why read the classics?', republished 12 October 2023 and available at https://www.penguin.co.uk/articles/2023/10/why-we-read-classics-italo-calvino

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

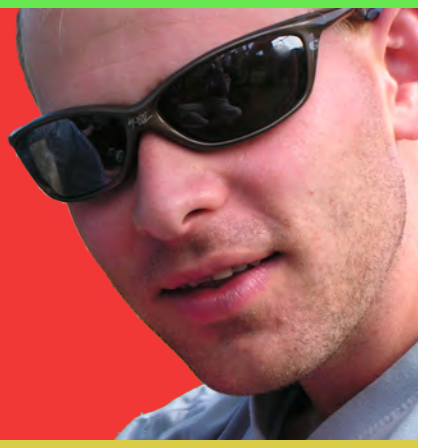


"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

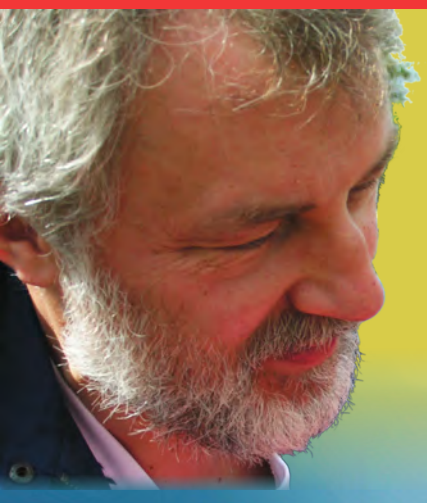
"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.



"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.



"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

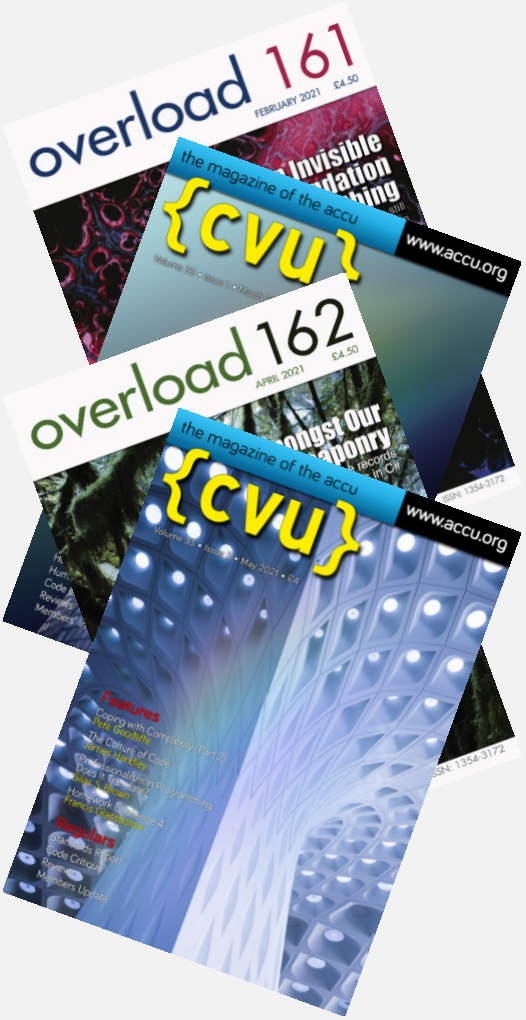
Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

Join ACCU

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

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



Join now!
Visit the website

professionalism in programming

www.accu.org