overload 179
FEBRUARY 2024 £4.50

Safety, Revisited

Lucian Radu Teodorescu gives an overview of last year's talks and articles about safety in C++ and presents a unified perspective.

User Stories and BDD – Part 3, Small or Far Away?

Seb Rose continues his investigation of user stories.

C++20 Concepts Applied – Safe Bitmasks Using Scoped Enums

Andreas Fertig gives a practical example where C++20's concepts can be used instead of enable_if.

Afterwood

Chris Oldwood explains why over-thinking is not over-engineering.

A magazine of ACCU
ISSN: 1354-3172

ACCU Conference 2024

17th - 20th April 2024
Workshops: 15th, 16th April

Welcome to the ACCU Annual Conference – a celebration of "Professionalism in Programming." by programmers for programmers about programming

ACCU invites everyone passionate about programming to join us in Bristol or virtually from anywhere in the world!

Beyond the core of C and C++, dive into a multitude of languages - C#, D, F#, Go, Javascript, Haskell, Java, Kotlin, Lisp, Python, Ruby, Rust, Swift, and more - at our conference. Sessions cover TDD, BDD, and expert programming practices.

4 days of the main conference with over 50 sessions.

Keynotes from: Herb Sutter, Inbal Levi, Laura Savino and Björn Fahller

2 days of workshops from:

Mateusz Pusz, Nicolai M. Josuttis, Kevlin Henney and Peter Sommerlad

Don't miss this opportunity to be part of a vibrant diverse community!

Book today to attend the ACCU Conference 2024

accuconference.org/pricing

February 2024
ISSN 1354-3172

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo from Adobe Stock Photos: a male Green Peafowl feather.

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 180 should be submitted by 1st March 2024 and those for Overload 181 by 1st May 2024.

Over-Promise, Under-Deliver

A new year can mean new beginnings. Frances Buontempo encourages us to stay motivated even if things don't work out.

'Tis the time of New Year Resolutions, new beginnings, and even an opportunity to write an editorial for the very first time. So much potential. However, I have not seized the chance, so we will have to cope without an editorial. We knew an editorial was highly unlikely, but dreaming about change or possibility can inspire and give hope, so we should be encouraged by the opportunity for a new start.

The trouble with New Year's Resolutions is we often start well then fail. Recently, I have been trying to learn bass, yet again. I managed to practise every day for over a month before Christmas, but a one night stop-over killed my streak. It is very easy to think, "I've blown it" and give up completely. That would be foolish. I managed to persuade myself I simply had a day off, and this means nothing significant about me, my determination, or my abilities. I simply cannot manage to do everything I want to do every day. It's perfectly fine to aim to do something every single day, as a motivation, but miss out once or twice. Aim high, by all means, but accept a few misses. By the same score, if I miss a note when I try to play bass, that's fine particularly because I am playing alone and no one can hear the mistakes. Even if I were playing with a band, I could continue on and forget the mistake and just about get away with it.

An occasional mistake in code might be acceptable, but there are many circumstances where a bug could be fatal. Various approaches have been adopted to avoid problems, including the UK government's 'SafeIT' research program, leading to the development of the Motor Industry Software Reliability Association (MISRA) guidelines for vehicle based software in the 1990s. The recent release of MISRA C [MISRA-1] and MISRA C++ [MISRA-2] has caused mention in a few places, but I have yet to see the full details. Though I have worked with embedded systems, they tended to be barcode scanners, or similar, rather than vehicles. The most dangerous thing I ever managed was undefined behaviour causing the laser scanner to stay on. Now, MISRA is used for more than vehicle software, and may be found in sectors ranging from NASA to medical devices. Whether the guidelines actually improve safety is hard to prove. To be certain you would need to clone the development team and let them code the system independently, one using the guidelines and one not. That is problematic, but then you need to test for differences in safety, which might prove even more difficult. More simply, some research on software reliability is available, for example Boogerd and Moonen conducted a case study investigating MISRA:C 2004 rule violation and actual faults [Boogerd08]. Be warned, they suggest, "Enforcing conformance to noisy rules can … increase the number of faults in the software." However, they also conclude "It is important to select accurate and applicable rules" and that doing so can make software more reliable. If used judiciously, guidelines can improve software.

There are various other guidelines for programming languages beyond MISRA. The ISO C++ Core Guidelines [ISO] and Google's C++ style guide [Google] come to mind, together with various linting tools. If you have ever started using a static analysis tool on a code base, you may have found yourself drowning in rule violations. Though Boogerd and Moonen's paper mentions frequent high levels of false positives from various tools, you might also find many true positives hiding in the output. Where do you start with a flood of warnings or problems? The tool you are using might promise to find every potential problem with your code, so you shouldn't be surprised if it seems to flag up almost every line of code. The important thing to do is apply some sense or discernment to the output. You can often silence various types of rules in a tool, making it easier to concentrate on specific types of problems. I recall being asked to conduct a Python code review a long time ago, and 'cheating' by pointing PyLint at the code. I silenced the complaints about single letter variable names, and noticed we had hidden a keyword by a bad choice of variable name. Without silencing a large number of 'problems', I would have missed a real problem. When I mentioned the name hiding in the code review, I was asked how I spotted this. I owned up and got a wry smile from the person who had written the code. We could have incorporated PyLint into the build, but didn't at the time. However, it did give us an extra way of investigating or reviewing new code.

Many tools claim to help us write better software, more quickly. Whether they do or not is another matter. Marketing often over-promises, and the goods sometimes under-deliver. If a tool provides something useful and doesn't cost the earth that might just be good enough. You don't have to fix every potential problem immediately. It is sometimes said that perfect is the enemy of good, so finding anything that helps a bit is a good thing. Aiming for perfection can make you freeze up and achieve nothing. Similarly, staring at a blank page makes it very difficult to write an editorial. Having something to start with, no matter how bad or irrelevant, gives you something to improve on. If you have broken any New Year's resolutions you made already, don't panic. Maybe you achieved something for a while, so you can be pleased with yourself and try again later. Don't forget the 80/20 rule, also called the Pareto principle. If you focus on changing 20% of what you don't like, you might end up with 80% improvement. Of course, that is hopelessly vague and doesn't specify what improvement or even change means, and the original Pareto observation related to property ownership [Wikipedia]. The salient point is that altering one thing might have many other impacts. They might even be positive. If you wait until you have listed everything that needs fixing, you are less likely to achieve anything.

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.

The ultimate overpromise may be attempting six impossible things before breakfast. The original quote is from *Through the Looking Glass* by Lewis Carroll, wherein the Queen says she has sometimes believed as many as six impossible things, rather than achieved them. However, if you were to aim for six impossible things, but only managed one, you are doing better than most people. If you don't want to believe or attempt anything impossible, you could try something possible instead. Maybe the New Year brings you a new project at work, or the chance to prepare a new talk for a conference or meetup. My ACCU conference proposal has been accepted, so I need to start writing it. Doing something new, especially if it involves something you have never done before, can be daunting. It would be much simpler to talk about techniques I already know and can rattle off without preparation; however, that would be boring. The opportunity to learn something new is exciting. I will learn about cat swarm algorithms. Yes, that is a thing: a 'cat' either traces or seeks, and algorithms can be built based on these modes to find solutions to problems [Chu06]. I hope to be able to explain this in more detail in a couple of months. Herding cats may be used as an example of something foolhardy and impossible, but implementing a cat swarm algorithm can't be that hard, surely? Watch this space.

Aiming to do something easy is lazy, whereas trying to do something you have never done before is admirable though risky. If I try this cat swarm optimisation, I might not understand it properly. I won't let fear of failure put me off though, and neither should you. ACCU members are supportive and easy to talk to, so I know I have people I can turn to if I get stuck. Perhaps you should be brave too. If you aren't going to speak at a conference, and have no intention of ever doing so, fair enough. Maybe consider writing an article instead? Or, try some of the puzzles and challenges in our member's magazine *CVu*, or write a book review. You get a free book if you get in contact with the reviews editor.[1] Push yourself outside your comfort zone this year. Like exercise in the gym, you may feel uncomfortable for a bit, but maybe you will experience some feel-good endorphins and make yourself a little stronger.

Staring at a blank page, as I said, can cause writer's block or similar. Now, AI doesn't suffer from this. Generative AI can waffle away easily enough. In fact, most AI algorithms may start by trying something at random, and then incrementally improve. Much of the agile movement encourages us to take a similar approach. Make something, a minimum viable product (MVP) for example and see how it goes. The important part is not just making a product, but observing its use. Rather than aiming for something to make money from, an MVP has "the additional criteria of being sufficient to learn about the business viability of the product." [Agile]. Trying something and using feedback to guide what happens next is an essential part of so many things, including software creation, AI, and learning in general.

Seeking feedback is one thing, but unsolicited feedback can prove incredibly unhelpful. I went for a walk round the block the other day, and noticed two ladies walking a dog. One was clearly the owner and the other a dog/dog-owner trainer. The dog's owner said "Heel!" as I went by, and I could hear the trainer saying you don't need to correct the dog unless they do the wrong thing. The dog had come to heel, but not lined up perfectly. I am not a dog owner, but I can see how being critical when something is approximately right is not the best way to encourage behaviours you want. It is difficult to say "That's really good, well done. Maybe next time, we can do even better!" each time feedback is requested and you notice something amiss. Maybe you can be more critical with some than others; however, let's all try to remember to be kind and encouraging whenever possible. If you are being drowned by negative feedback, and are not in a position to find better collaborators, then don't be shy about asking which the best parts were to remind others pointing out the good is as important as pointing out the bad.

Lots of people seize New Year, or any 'significant' date, as a time to write dimly related blogs or mail shots and this year is no exception. One that caught my eye touted the old aphorism, 'Goals without plans are just wishes.' Now, there is an element of truth to that. I can write down several goals, like continuing to play my bass. However, without specifics of what to play, practice exercises, and the like, I probably won't make much progress. Now, I don't need to plan out precisely what I plan to do when, but a handful of songs to try and scales to practise is enough to make a start. I may not have a fully formed plan, but I do have a goal. And anyway, what's wrong with wishes? I wish you all a Happy New Year, and along with the review team, am here to encourage you and maybe you will manage something you thought was impossible this year, even if you don't get around to everything you hoped.

References

[Agile] 'Minimum Viable Product (MVP)', definition in the glossary of the Agile Alliance: https://www.agilealliance.org/glossary/mvp/

[Boogerd08] C. J. Boogerd and L. Moonen (2008) 'Assessing the Value of Coding Standards: An Empirical Study', preprint of a paper published in ICSM 2008 (IEEE International Conference on Software Maintenance, 28 September–4 October 2008, available: http://resolver.tudelft.nl/uuid:646de5ba-eee8-4ec8-8bbc-2c188e1847ea

[Chu06] Shu-Chuan Chu, Pei-Wei Tsai and Jeng-Shyang Pan (2006) 'Cat Swarm Optimization', available https://www.researchgate.net/publication/221419703_Cat_Swarm_Optimization

[Google] Google C++ Style Guide: https://google.github.io/styleguide/cppguide.html

[ISO] C++ Core Guidelines (Bjarne Stroustrup and Herb Sutter, editors): https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[MISRA-1] MISRA C:2023 https://misra.org.uk/misra-c2023-released/

[MISRA-2] *MISRA C++:2023: Guidelines for the use C++:17 in critical systems* https://misra.org.uk/misra-cpp2023-released-including-hardcopy/

[Wikipedia] Pareto principle: https://en.wikipedia.org/wiki/Pareto_principle

1 See https://accu.org/menu-overviews/reviews-overview/ for details.

User Stories and BDD – Part 3, Small or Far Away?

The size of user stories is important. Seb Rose continues his investigation of user stories, considering when and how to size them.

This is the third in a series of articles digging into user stories, what they're used for, and how they interact with a BDD approach to software development. You could say that this is a story about user stories. And like every good story, there's a beginning, a middle, and an end. This article is a continuation of the middle.

Previously ...

In the last article [Rose23], we followed a user story through the process of Discovery [Nagy18]. We saw that the main purpose for a user story was to minimise waste by making decisions at the last responsible moment, that accidental discovery is an unavoidable source of waste, but can be minimised by embracing deliberate discovery. We introduced Example Mapping as a deliberate discovery technique and observed that, through discovery, stories are transformed from a placeholder for a conversation into detailed small increments. Now it's time to talk about what we mean by 'small' and why, in the context of user stories, small is beautiful.

Small and valuable

The INVEST acronym was created by Bill Wake [Wake03] and popularised by Mike Cohn. It reminds us that stories should exhibit the following characteristics:

- **I** – independent
- **N** – negotiable
- **V** – valuable
- **E** – estimable
- **S** – small
- **T** – testable

This acronym encapsulates good advice about what a user story should look like. I'll write about the other aspects of this acronym in the future, but for now I'd like to point out the tension that teams usually feel between stories being small and stories being valuable.

Value is often thought of as delivering functionality that will earn money from fee-paying customers. While that is one definition of value, it is very narrow, ignores the incremental nature of value, and encourages teams to work on larger stories than they should.

I have my own definition of value, that is much broader, and fits better with the iterative and incremental nature of agile [Rose19]:

> Value is any piece of work that increases knowledge, decreases risk, or generates useful feedback.

This definition of value allows us to work on really small stories, but it seems that most teams are resistant even though they typically accept that fast feedback is beneficial.

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. Co-author of the BDD Books series Discovery and Formulation (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

Why small is beautiful

There's really only one reason to work on smaller stories, which is reduced waste. For that to make sense we need to understand where waste occurs during software development. For the purposes of this article, I'm going to assert that the majority of waste arises from one of two sources: ignorance and disrupted flow.

Ignorance

Software development is a process of learning and we learn by getting feedback. Whether that's feedback from the customer that we have misunderstood their problem, or feedback from tests that we've not understood the limitations, it's all valuable learning. To learn faster we need to get feedback faster.

The more work we do between each piece of feedback, the more rework we'll have to do if it turns out that we've been building on misunderstandings or incorrect assumptions. If we can minimise rework, then we can minimise the risk that the work we're doing will be wasted.

And, just to be clear, there are multiple risks that learning (and hence feedback) needs to address. The bigger the story, the higher the risk that it's hiding some complexities that we're ignorant of. The earlier that we can reduce that ignorance, the sooner we can begin to decrease our exposure to risk.

Big stories mean we have to wait longer for feedback; smaller stories mean we can get our feedback more quickly.

Disrupted flow

While waste due to ignorance is usually problem-specific, the waste due to the processes that our teams follow are often systematic.

The most efficient way to work on stories is what Lean manufacturing calls single piece flow. Essentially this means that when someone starts work on a story, they can independently take it to completion. The larger the story, the more likely it is that it will contain some obstacle that interrupts its completion. When that happens, the flow of the work will be disrupted and they will generally context switch to another story. Context switching is a major source of waste.

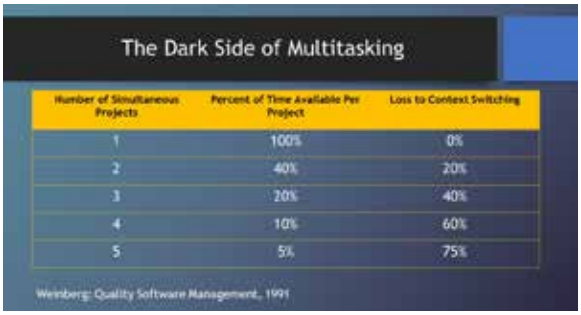

Image: Maureen Adamson [Adamson19]

Figure 1

The way you choose to slice your stories becomes a really important skill, so that you're able to gradually fade up the fidelity until you're ready to ship.

How small is small?

There's a conversation that takes place in *The Hitchhiker's Guide to the Galaxy* which neatly describes peoples' reaction to my answer [MovieQuotes]:

> *Lunkwill*: Do you...
>
> *Deep Thought*: Have an answer for you? Yes. But you're not going to like it.

As long as a story delivers value (see my definition of value above), then my answer (whether you like it or not) is: the smaller that you can make the story, the better.

The standard objection to this is that it's just not efficient to do such small pieces of work. But why is it not efficient? Read on….

Transaction cost

The challenge with delivering tiny stories is that our development processes are frequently inefficient. The cost of creating a small story and pulling it through your development pipeline is called the transaction cost. A heavyweight process with, for example, a manual release process involving management sign-off, incurs a high transaction cost for every story. At the other end of the spectrum, a fully-automated Continuous Delivery pipeline has an extremely low transaction cost. The lower you can make this cost compared to the cost of actually delivering a story's value, the more worthwhile it is for you to ship small stories often.

There are major global corporations that have focused on making their processes efficient – and they can deliver thousands of tiny stories into production every day. Most organisations still think that demonstrating a new piece of functionality every two weeks is quite advanced.

To maximise the efficiency of our development teams we should focus on improving their flow. It took Toyota years to get to where they are today, but they didn't stop making cars while they improved their production processes – and we shouldn't stop delivering software.

Before your next retrospective, read *The Bottleneck Rules* (a very short book) and think about applying some of the techniques that it describes.

Low fidelity stories

Small stories will often be low fidelity stories [Scotland09].

> Fidelity refers to the finesse of the feature, or solution – low fidelity solution will be low in things like precision, granularity, or usability, but will still [help us learn how to] solve the original problem.

The goal is to do as little work as possible to learn whether we're progressing in the right direction.

Jeff Patton [Patton08] uses the analogy of painting the Mona Lisa to demonstrate the difference between iterative and incremental development. The first story would deliver an outline of a part of the composition, with future stories

- iterating on the outline by adding colour, texture, and detail

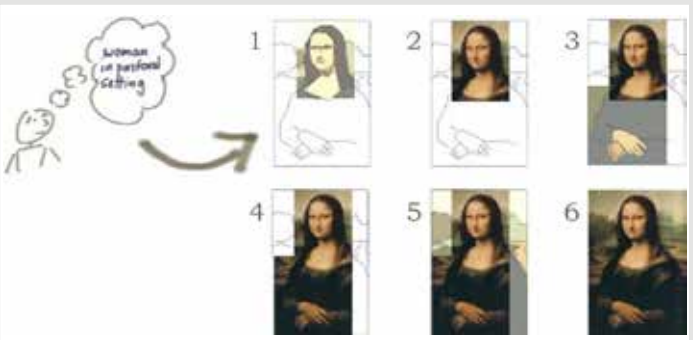

Image: Steven Thomas [Thomas12]

Figure 2

- incrementally extending the outline to cover more of the composition

Agile teams work both iteratively and incrementally, starting with low fidelity stories that allow us to learn fast and early.

Individual stories don't necessarily have to be something coherent enough to release to your users. You may need to complete several detailed small increments before you've added enough value that your users would appreciate or even notice it. The way you choose to slice your stories becomes a really important skill, so that you're able to gradually fade up the fidelity until you're ready to ship.

Story slicing

Once you accept the benefits of small stories, there's still the challenge of creating them. There are many techniques for decomposing stories into thin slices, but I'd like to share three nuggets that you should keep in mind.

Asteroids

In his book *User Story Mapping*, Jeff Patton [Patton14] draws an analogy with the old arcade game Asteroids. The decomposition of large, slow-moving rocks into tiny, fast-moving, dangerous rocks is instructive.

Figure 3

The worst technique in Asteroids is to break all the big rocks into medium rocks, and all the medium rocks into tiny rocks. The outcome, if you do this, is a screenful of fast moving, dangerous rocks that inevitably destroy your spaceship.

A similar thing happens if you try to populate your entire backlog with tiny stories. You have a backlog that's impossible to manage, full of duplicate and out-of-date stories. Instead, break off a chunk at a time and decompose that into small stories, leaving the backlog mostly full of big and medium stories.

Example mapping

Example mapping [Wynne15] is a technique that can really help slice a medium story into a set of small stories. It's well described elsewhere and provides a structured way to collaboratively analyse a problem and generate concrete examples of how the system should actually behave.

The beauty of this technique is that the example map itself visually communicates whether the story is too big AND gives us a simple mechanism to slice it up. Because the product owner is present during example mapping, this approach has the added benefit that business priorities can be taken into account at a very fine granularity.

The flowchart

Example mapping is my go-to technique when analysing and slicing a story, but it's by no means the only way to do it. Richard Lawrence created the extremely useful 'How To Split A User Story' flowchart [Lawrence18]. Print this out (on a LARGE piece of paper) and hang it on your team's wall.

Two week iterations

To learn fast and maintain flow you need stories to be significantly smaller than your iteration. For most XP and Scrum teams, this means that every story should be deliverable (from Backlog->Done) in 3 days or fewer. This is hard, but the benefits are immense.

Small or far away?

Once you set about trying to work with smaller stories, you'll probably find that, even after slicing them, they're still bigger than you would like. The challenge is that until you really dig into the details of a story, some of its complexity remains stubbornly hidden.

As Father Ted, discovered – it's all a matter of perspective. [FatherTed]■

References

[Adamson19] Maureen Adamson, 'The High Cost of Context Switching', posted 14 May 2019, https://madamsonassociates.com/blog/high-cost-of-context-switching

[FatherTed] 'Cows: Small or far away?', an extract from the *Father Ted* comedy series (Channel 4 Entertainment) on YouTube: https://www.youtube.com/watch?v=MMiKyfd6hA0

[Lawrence18] Richard Lawrence (2018) 'How to Split a User Story' (flowchart), available at: https://agileforall.com/wp-content/uploads/2020/12/Story-Splitting-Flowchart.pdf

[MovieQuotes] *The Hitchhiker's Guide to the Galaxy* (2005), quotes listed at https://www.moviequotes.com/s-movie/the-hitchhikers-guide-to-the-galaxy-1/

Image: From Agile 2018 [Rose18]

Figure 4

[Nagy18] Gaspar Nagy and Seb Rose (2018) Discovery: Explore behaviour using examples: Volume 1 (BDD Books), https://bddbooks.com/

[Patton08] Jeff Patton, 'Don't Know What I Want, But I Know How to Get It', posted on 21 January 2008 at https://jpattonassociates.com/dont_know_what_i_want/

[Patton14] Jeff Patton and Peter Economy (2014) *User Story Mapping: Discover the Whole Story, Build the Right Product*, published by O'Reilly, ISBN-13: 978-1491904909

[Rose18] Seb Rose 'How long is a piece of string?' from Agile 2018, available at https://www.agilealliance.org/resources/videos/how-long-is-a-piece-of-string/

[Rose19] Seb Rose 'User stories: from good intentions to bad advice', *Lean Agile Scotland 2019*, https://www.slideshare.net/sebrose/user-stories-from-good-intentions-to-bad-advice-lean-agile-scotland-2019

[Rose23] Seb Rose 'User Stories and BDD – Part 2, Discover' in *Overload* 178, December 2023, available at https://accu.org/journals/overload/31/178/rose/

[Scotland09] Karl Scotland 'Fidelity – The Lost Dimension of the Iron Triange' posted on 22 December 2009 at https://availagility.co.uk/2009/12/22/fidelity-the-lost-dimension-of-the-iron-triangle/

[Thomas12] Steven Thomas 'Revisiting the Iterative Incremental Mona Lisa', posted 3 December 2012, http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa

[Wake03] Bill Wake 'INVEST in Good Stories, and SMART Tasks', posted 17 August 2003 at https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

[Wynne15] Matt Wynne, 'Introducing Example Mapping', posted 8 December 2015 on Seb Rose's blog: https://cucumber.io/blog/bdd/example-mapping-introduction/

This article was published on Seb Rose's blog on 9 January 2020: https://cucumber.io/blog/bdd/user-stories-and-bdd-part-3/

C++20 Concepts Applied – Safe Bitmasks Using Scoped Enums

It can be hard to follow code using enable_if. Andreas Fertig gives a practical example where C++20's concepts can be used instead.

In 2020 I wrote an article for the German magazine *iX* called 'Scoped enums in C++' [Fertig20]. In that article, I shared an approach of using class enums as bitfields without the hassle of having to define the operators for each enum. The approach was inspired by Anthony William's post 'Using Enum Classes as Bitfields' [Williams15].

Today's article aims to bring you up to speed with the implementation in C++17 and then see how it transforms when you apply C++20 concepts to the code.

One operator for all binary operations of a kind

The idea is that the bit-operators are often used with enums to create bitmasks. Filesystem permissions are one example. Essentially you want to be able to write type-safe code like this:

```
using Filesystem::Permission;
Permission readAndWrite{
  Permission::Read | Permission::Write};
```

The enum **Permission** is a class enum, making the code type-safe. Now, all of you who once have dealt with class enums know that they come without support for operators. Which also is their strength. You can define the desired operator or operators for each enum. The issue here is that most of the code is the same. Cast the enum to the underlying type, apply the binary operation, and cast the result back to the enum type. Nothing terribly hard, but it is so annoying to repeatedly type it.

Anthony solved this by providing an operator, a function template that only gets enabled if you opt-in for a desired enum. Listing 1 is the implementation, including the definition of **Permission**.

```
template<typename T>
constexpr std::
  enable_if_t<
    std::conjunction_v<std::is_enum<T>,
      // look for enable_bitmask_operator_or
      // to  enable  this operator ❶
      std::is_same<bool,
        decltype(enable_bitmask_operator_or(
          std::declval<T>()))>>,
  T>
operator|(const T lhs, const T rhs) {
  using underlying = std::underlying_type_t<T>;
  return static_cast<T>(
    static_cast<underlying>(lhs) |
    static_cast<underlying>(rhs));
}
namespace Filesystem {
  enum class Permission : uint8_t {
    Read = 1,
    Write,
    Execute,
  };
  // Opt-in for operator| ❷
  constexpr bool
    enable_bitmask_operator_or(Permission);
} // namespace Filesystem
```

Listing 1

Neat, isn't it?

The trick part is in the template-head in ❶. The **is_same** together with **decltype** and, of course, **std::declval** checks that a function **enable_bitmask_operator_or** exists for the given enum, which I provide in ❷. Well, **enable_if**.

Let's use the code for **operator|** and see how C++20 can simplify your code.

C++20's concepts applied

The great thing about C++20s concepts is that we can eliminate the often hard-to-digest **enable_if**. Further, checking for functions' existence requires less code due to the requires-expression of concepts.

Listing 2 is the same operator using C++20s concepts instead of the **enable_if**.

I can't tell you how much I like this code. No **decltype**, no **is_same**, no **conjunction**, and no **declval**. So beautiful.

The requires-expression tries to call **enable_bitmask_operator_or** in ❶, together with the **is_enum_v**, that's all that's required in C++20.

There is one other bonus in C++20. Since you have not only **constexpr** but also **consteval** functions available, applying them in ❷ to

```
template<typename T>
requires(std::is_enum_v<T>and requires(T e) {
  // look for enable_bitmask_operator_or to
  // enable  this operator ❶
  enable_bitmask_operator_or(e);
}) constexpr auto
operator|(const T lhs, const T rhs) {
  using underlying = std::underlying_type_t<T>;
  return static_cast<T>(
    static_cast<underlying>(lhs) |
    static_cast<underlying>(rhs));
}
namespace Filesystem {
  enum class Permission : uint8_t {
    Read    = 0x01,
    Write   = 0x02,
    Execute = 0x04,
  };
  // Opt-in for operator| ❷
  consteval
    void enable_bitmask_operator_or(Permission);
} // namespace Filesystem
```

Listing 2

Andreas Fertig is a trainer and lecturer on C++11 to C++20, who presents at international conferences. Involved in the C++ standardization committee, he has published articles (for example, in *iX*) and several textbooks, most recently *Programming with C++20*. His tool – C++ Insights (https://cppinsights.io) – enables people to look behind the scenes of C++, and better understand constructs. He can be reached at contact@andreasfertig.com

`enable_bitmask_operator_or` signals a bit better that this function is for compile-time purposes only.

C++23: The small pearl

One more thing. You have C++23 available now[1]. There is one change you can now make to simplify the code even more. C++23 offers you `std::to_underlying` for converting a class enum value to a value of its underlying type. The function is located in `<utility>`.

Applying this to the example leads to the code in Listing 3.

Not only does `std::to_underlying` remove redundant and boring code you had to write before C++23 but, in my opinion, the utility function makes the code more readable as well. ■

```
template<typename T>
requires(std::is_enum_v<T>and requires(T e) {
  enable_bitmask_operator_or(e);
}) constexpr auto
operator|(const T lhs, const T rhs)
{
  return static_cast<T>(std::to_underlying(lhs) |
                        std::to_underlying(rhs));
}
```

Listing 3

References

[Fertig20] Andreas Fertig, 'Scoped Enums in C++11' in iX, accessible from https://www.heise.de/select/ix/2020/7/2006907575811763393

[Williams15] Anthony Williams 'Using Enum Classes as Bitfields', posted 29 January 2015 at https://www.justsoftwaresolutions.co.uk/cplusplus/using-enum-classes-as-bitfields.html

1 Check which parts of C++23 are available on your chosen compiler at https://en.cppreference.com/w/cpp/compiler_support

This article was first published on Andreas Fertig's blog on 2 January 2024: https://andreasfertig.blog/2024/01/cpp20-concepts-applied/

And the winners are...

In *Overload* 178 and *CVu* 35.6, we invited you to vote for your favourite articles both in *Overload* and *CVu* (which is our sister publication for members). The results are in.

Overload

Winner

Floating-Point Comparison
(Paul Floyd, in *Overload* 173)

Runners-up

Use SIMD: Save the Planet
(Andrew Drakeford, in *Overload* 178)

Live and Let Die
(Martin Janzen, in *Overload* 177)

CVu

Winner

Are the Old Ways Sometimes the Best? (Roger Orr, in CVu 35.4)

Runner-up

Care About the Code
(Pete Goodliffe, in *CVu* 35.2)

Thank you to everyone who took the time to vote, and to those who wrote the articles. We can't offer a prize – just the mention here.

A number of other writers got a vote, so if you wrote something for us, someone probably thoroughly enjoyed what you had to say.

If you're reading this online, the article titles link to the articles. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones.

If you're not a member yet, why not join?

Safety, Revisited

Last year saw a proliferation of talks and articles about safety in C++. Lucian Radu Teodorescu gives an overview of these and presents a unified perspective on safety.

Safety was a hot topic in 2023 for the C++ community. Leading experts took clear positions on its significance in the context of C++ and, in general, system programming languages. They explored various aspects, including general safety principles, functional safety, memory safety, as well as the intersections between safety and security, and safety and correctness. Many of these discussions were influenced by recent reports [NSA22, CR23, WH23a, EC22, CISA23a, CISA23b], that strongly criticised memory-unsafe languages.

In this context, it's logical to revisit the primary safety discussions from last year and piece together a comprehensive understanding of safety in the context of C++. While experts may find common ground, they also have differing opinions. However, it's likely that the nuances expressed by the authors hold greater significance than mere agreements or disagreements.

In this article, we will examine key C++ conference talks with a primary focus on safety, along with a brief mention of relevant podcasts. Our selection is limited to talks and podcasts from 2023. Subsequently, we will consolidate the insights and viewpoints of various authors into a unified perspective on safety in system languages.

Safety in 2023: a brief retrospective

Sean Parent, All the Safeties

In his presentation at *C++ now* [Parent23a], Sean Parent presents the reasons why it's important to discuss safety in the C++ world, tries to define safety, argues that the C++ model needs to improve to achieve safety, and looks at a possible future of software development. This same talk was later delivered as a keynote at *C++ on Sea* [Parent23b].

Sean argues the importance of safety by surveying a few recent US and EU reports which have begun to recognise safety as a major concern [NSA22, CR23, WH23a, EC22]. There are a few takeaways from these reports. Firstly, they identify memory safety as a paramount issue. The NSA report [NSA22], for instance, cites a Microsoft study noting that "70 percent of their vulnerabilities were due to memory safety issues". Secondly, they highlight the inherent safety risks in C and C++ languages, advocating for the adoption of memory-safe languages. Lastly, these documents suggest a paradigm shift in liability towards software vendors. Under this framework, vendors may face accountability for damages resulting from safety lapses in their software.

Building on the reports that underscore the significance of safety, Sean delves into deciphering the meaning of 'safety' in the context of software development. After evaluating several inadequate definitions, he adopts a framework conceptualised by Leslie Lamport [Lamport77]. The idea is to express the correctness of the program in terms of two types of properties: safety properties and liveness properties. The safety properties describe what cannot happen, while the liveness properties indicate what needs to happen.

As highlighted in other talks that Sean gave (see, for example, 'Exceptions the Other Way Round' [Parent22]), safety composes. If all the operations in a program are safe, then the program is also safe (if preconditions are not violated). Correctness, on the other hand, doesn't compose like safety. This is why safety is a (easily) solvable problem.

Sean further elaborates on what constitutes memory safety in a language. Citing 'The Meaning of Memory Safety' [Amorim18], he argues that memory safety is what the authors of the paper call *the frame rule*. This rule is equivalent to the *Law of Exclusivity*, coined by John McCall [McCall17]. Sean then explains why C++ can never be a safe language. In general, any language that allows undefined behaviour is an unsafe language.

Herb Sutter, Fill in the blank: _____ for C++

In his keynote at *C++ now*, 'Fill in the blank: _____ for C++' [Sutter23a], Herb Sutter presents the latest developments in his cppfront project, envisioned as a successor to C++. He presented this talk again, with minor variations, at *CppCon 2023* under the title 'Cooperative C++ Evolution: Toward a Typescript for C++' [Sutter23b]. A primary objective of this new language is to significantly enhance safety. Herb sets an ambitious goal of improving safety by 50 times compared to C++. His plan for achieving this goal is to have bounds and null checking by default, guaranteed initialisation before use, and other smaller safety improvements (contracts, default const, no pointer arithmetic, etc).

The features that Herb presents in this talk are fairly small in terms of safety improvements that they bring (especially compared to his previous talk [Sutter22]). However, I included Herb's keynote because he advocates a gradual approach to safety, and provides a clear measurement for achieving the goal. One might say that this approach is more pragmatic.

As an interesting observation, Herb has two different approaches to two important features of his language: safety and compatibility. While he advocates a gradual adoption for safety, he advocates an all-or-nothing approach to compatibility (a good successor needs to be fully compatible with the previous language from day one).

Bob Steagall, Coding for Safety, Security, and Sustainability (panel discussion)

Safety was an important topic at *C++ now*, and the conference organised a panel with JF Bastien, Chandler Carruth, Daisy Hollman, Lisa Lippincott, Sean Parent and Herb Sutter [Steagall23].

The panellists disagreed on a definition of safety, and they disagreed on the relation between safety and security. But, apart from that, there seemed to be a consensus on multiple topics: it's difficult to express safe/correct code in C++, safety is important to the future of C++, safety and performance are not incompatible, the C++ experts need to consider more the opinion of security experts, the programmers also have responsibilities

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

the resistance within the C++ community towards memory safety mirrors the initial reluctance of the automotive industry to adopt seatbelts

for delivering safe and secure code (not just managers) and that regulation of the industry is likely imminent.

One important point that Daisy puts forward is that there shouldn't be a single answer for safety. She points out that the HPC community is not particularly interested in safety and security, and they focus solely on performance.

Chandler Carruth, Carbon's Successor Strategy: from C++ interop to memory safety

In his presentation at *C++ now 2023*, titled 'Carbon's Successor Strategy: from C++ interop to memory safety' [Carruth23], Chandler Carruth delved into the ongoing evolution of the Carbon language. As memory safety is a key objective for the Carbon language, a significant portion of his talk also addressed the strategy of handling safety by the Carbon community.

In his talk, Chandler offers a different type of definition for safety, starting from bugs. Safety, according to Chandler, is the guarantees that the program provides in the face of bugs. According to Chandler, safety is not a binary state; rather, it can exist in varying degrees. Chandler defines *memory safety* as a mechanism that "limits program behaviour to only read or write intended memory, even in the face of bugs". Then he goes on to make an important clarification: we may not want the entire language to be memory safe, but we may want a subset of the language to be memory safe. This subset should serve as a practical default, with unsafe constructs being the exception. Additionally, there should be a clear and auditable demarcation between safe and unsafe elements in the language. Intriguingly, Chandler does not deem data-race safety as a strict requirement for this safe subset, although he acknowledges it as an admirable objective to strive for.

The Carbon migration strategy that Chandler presented is a step-by-step process. Initially, the transition involves moving from unsafe C++ code to Carbon, potentially utilising some unsafe constructs. Subsequently, the strategy shifts towards adopting a safe subset of Carbon. This phased approach breaks down the transition from unsafe to safe code into more manageable steps, enabling an incremental migration process.

Throughout his talk, Chandler implicitly advocates the viewpoint that memory safety ought to be a fundamental expectation in programming languages. He suggests that software engineers should have the right to demand safety guarantees in their work.

JF Bastien, Safety & Security: the future of C++

Right from the outset of his *C++ Now* keynote [Bastien23], JF Bastien presents a compelling argument: safety and security represent existential threats to C++. Software is central in modern society, and safety issues can have serious consequences, potentially even leading to loss of life. To reinforce his point, JF cites an extensive array of reports and articles, stressing the message that the C++ community cannot afford to neglect safety [NSA22, CR23, Gaynor18, Black21, Dhalla23, Claburn23, CISA23a, CISA23b].

JF draws a striking analogy in his talk: he compares programming in C++ to driving without seatbelts. He points out that the resistance within the C++ community towards memory safety mirrors the initial reluctance of the automotive industry to adopt seatbelts. His vision is for safe programming languages to become as universally accepted and life-saving as seatbelts. This perspective gradually evolves into an ethical argument. JF suggests that to truly adhere to our duty of avoiding harm, it's essential to take the necessary steps to mitigate safety issues in programming as much as possible.

Later on, JF argues that we don't have a common understanding of *safety* and attempts to provide a definition for what safety means; for him, safety is about *type safety*, *memory safety*, *thread safety* and *functional safety*.

Type safety "prevents type errors" ("attempts to perform operations on values that are not of the appropriate data type"). He references Robin Milner's famous quote "Well-typed programs cannot go wrong" [Milner78] to indicate the importance of type safety. Turning the attention to C++, he argues that it's really hard in C++ to follow all the best practices regarding type safety; he also argues that it's difficult to guarantee the absence of undefined behaviour, so C++ cannot be considered a type-safe language.

For memory safety, JF references 'The Meaning of Memory Safety' [Amorim18], and defines memory safety as the absence of use-after-free and out-of-bounds accesses (he doesn't include use of uninitialised values). To the question whether C++ has memory safety, the answer is no, C++ is not there yet, but JF points out a few alternatives how C++ can get memory safety; it's worth noting that each of the approaches has trade-offs.

Thread safety is defined as the absence of data races. Like with the other types of safety, C++ doesn't have a way of guaranteeing the lack of data races.

Regarding functional safety, he defines functional safety as "the systematic process used to ensure that failure doesn't occur". While the programming part of this is important, functional safety extends beyond it, to processes and people; having a "safe culture" (where the boss can hear bad news) is also important for functional safety. JF additionally argues that security is also needed to achieve functional safety.

In the rest of the talk, JF discusses the distinction between the two adversaries behind safety and security: stochastic vs smart adversaries, and how the smart adversary may have a wide range of resources. He discusses many nuances of preventing and mitigating these types of issues. Towards the end of the talk the subject of possible regulations is tackled; based on recent reports [OpenSSF22, WH23b, Hubert23], JF believes that our field will soon be regulated.

Andreas Weis, Safety-First: Understanding How To Develop Safety-critical Software

At *C++ now*, Andreas Weis talks about safety from a slightly different perspective, focusing on software development for safety-critical domains like the automotive industry. He delved into topics such as functional

The guidelines can help users write safer code, while the profiles can enforce the guidelines with appropriate tooling (static analysers).

safety, existing regulations, processes or multiple methods of achieving safety [Weis23].

Andreas starts by defining safety. His definition of safety is also inspired from Leslie Lamport [Lamport83]: "something bad does not happen". As examples, Andreas gives partial correctness ("the program does not produce the wrong answer"), mutual exclusion and deadlock freedom. He also defines safety from the perspective of Functional Safety, as defined by ISO 26262:2018 [ISO26262]. The ISO standard defines *safety* as being the "absence of unreasonable risk", and *unreasonable risk* as "risk judged to be unacceptable in a certain context according to valid societal moral concepts", and *risk* as the "combination of the probability of occurrence of harm and the severity of that harm". Crucial to this definition are the notions of *unreasonable* and the probability factor in risk assessment. The ISO processes require a thorough risk evaluation to determine the significance of each risk.

Another important point that Andreas draws attention to is the fact that preventing a safety fault is just one way of dealing with the fault. There are other ways to deal with the fault (control the impact, designing fault-tolerant system, increasing controllability, etc.)

Andreas also explains that defining *intended functionality* for a system is important; there may not be universal guarantees for a system. He also briefly attempts at providing a distinction and commonalities between safety and security.

Much of Andreas's talk was dedicated to discussing the processes mandated by ISO for addressing safety concerns. Pertinent to our discussion are the sections where these processes dictate coding requirements. While the standards mention some specific items, they primarily mandate companies to develop coding standards that address various safety concerns.

Bjarne Stroustrup: Approaching C++ Safely

In the *Core C++* opening keynote, 'Approaching C++ safety' [Stroustrup23a], Bjarne Stroustrup presented a blend between his ideas and standards committee on approaching safety in C++ (referencing [P2759R1, P2739R0, P2816R0, P2687R0, P2410R0]). He tries to capture the many nuances of safety, discusses the evolution of C++ towards safety and also a possible future for C++ regarding safety. He delivers roughly the same talk as a *CppCon* keynote in October 2023 [Stroustrup23a].

From the beginning of the talk, Bjarne argues that safety is not just one thing, it's actually a set of other things; he lists some of the things that safety means but fails to define any of the terms discussed. He acknowledges that the recent NSA report [NSA22] is a cause of concern and that C++ can be massively improved in terms of safety. The approach that Bjarne suggests is relying on guidelines and tooling; this puts most of the responsibility on the users, and not on the committee members. Notably, his response to criticism, evident in the first keynote [Stroustrup23a], was marked by a notably sharp tone.

A major part of the talk discusses the evolution of C++ and how, during the decades, it improved safety compared to C. Most of the ideas were also presented in different forms by Bjarne at different conferences before this 'safety crisis'.

After discussing the evolution of C++ so far, the talk goes to discuss the C++ core guidelines (present) and the safety profiles (future). The guidelines can help users write safer code, while the profiles can enforce the guidelines with appropriate tooling (static analysers). Using profiles will allow gradual improvement in safety.

Throughout the talk, the audience can hear Bjarne say that some problems are hard, and for some of them we may not get any static check soon. The references that are made in the talk [P2759R1, P2739R0, P2816R0, P2687R0, P2410R0] do not offer clear guarantees that all the safety issues of C++ will be tackled. The references slide included a 2015 paper with the note "we didn't start yesterday," underscoring the slow pace of safety improvements. This leaves an impression that fully resolving C++'s safety issues is likely to be a prolonged endeavour.

Timur Doumler, C++ and Safety

Timur Doumler gave a talk called 'C++ and Safety' both at *C++ on Sea* [Doumler23a] and at *CppNorth* [Doumler23b], explaining his perspective on safety. While his approach is similar to what others have said before at *C++ now* 2023, he has some new takes on safety and C++, more specifically on the importance of safety in C++.

In the first part of the talk, he gives a taxonomy around safety, touching functional safety, language safety, correctness (total and partial), and on the relation between undefined behaviour and safety. He focuses on language safety; he defines a program as *language safe* if it has no undefined behaviour, and considers a programming language *safe* if it cannot express programs that are not *language safe*.

He explores different types of safety issues (type safety, bounds safety, lifetime safety, initialisation safety, thread safety, arithmetic safety and definition safety). Furthermore, he provides examples and discusses them in terms of trade-offs. A language can ban undefined behaviour, but, he argues, that would have other negative consequences and would make some old programs not work anymore.

Towards the end of the talk, Timur starts to draw some conclusions. First, C++ has too much undefined behaviour, and that would make it impossible for C++ to become a safe language. The industry has developed tools and practices to make this a smaller problem. His second assertion is that compromising on performance might pose a greater threat to C++ than compromising on safety. This leads to his third conclusion: C++ is not doomed if it fails to become a memory-safe language.

To back up his claims on completely fixing C++ safety issues, he presents some data. First, he looks at the number of vulnerabilities per language: while C and C++ are often quoted together as having a memory safety issue, there is a large gap between the two languages. From the total vulnerabilities looked at, 46.9% are in C, while only 5.23% are in C++.

Other languages, like PHP, Java, JavaScript and Python have more vulnerabilities than C++; Java, considered a safe language, has an 11.4% share of vulnerabilities, which is twice as much as C++.

Then, he presents the results of a survey that he ran to determine the importance of safety for C++ users. The main conclusion is that "today, C++ developers generally do not perceive undefined behaviour as a business-critical problem".

Robert Seacord, Safety and Security: The Future of C and C++

Another important talk related to safety was Robert Seacord's keynote at *NDC TechTown*, entitled 'Safety and Security: The Future of C and C++' [Seacord23]. The talk was based on Bastien's talk 'Safety & Security: the future of C++' [Bastien23] and most of the ideas are repeated. In addition to the ideas presented in Bastien's talk, Robert, being the convenor for the ISO standardisation working group for the C programming language, added content to also cover C, not just C++.

Gabor Horvath, Lifetime Safety in C++: Past, Present and Future

In his *CppCon* talk [Horvath23], Gabor Horvath discusses some possible approaches to improve lifetime safety. He is deliberately not explaining once more why safety is important and briefly mentions a few C++now talks [Bastien23, Parent23a, Steagall23, Weis23] and a few reports [NSA22, CR23].

What is interesting in Gabor's talk is the distinction between *safe by construction* and *opportunistic bug finding* (Gabor also has a third category named *hybrid approach*, but I failed to understand what's the difference between this one and *opportunistic bug finding*). In a *safe by construction* language, the expressible programs are guaranteed to be safe. This may reject safe programs if the compiler cannot reason that they are safe; often, for this reason, the language allows escape hatches. On the other hand, in an *opportunistic bug-finding* approach, the language allows all the programs whether they are safe or not; then, language warnings, static analysers or other tools might identify safety issues. In this model, we incrementally move to safer code as the tools suggest safer constructs. The major downside is that there will be unsafe programs that we won't be able to detect.

Gabor spends a fair amount of time discussing recent improvements in C++ (or the MSVC/clang compiler) for lifetime safety. What I found interesting in this section is not necessarily the recent improvements (although they are great), but the many ways in which we can write unsafe code. The takeaway I have is that the more features we add to C++, the more possibilities of expressing unsafe code we have, making it harder for people to reason about them.

Podcasts

In terms of podcasts, the safety theme appeared on many episodes. Out of all these episodes, I've selected a few where safety plays a central role: *CppCast*'s 'Safety Critical C++ with Andreas Weis' [CppCast356], *CppCast*'s 'Safety, Security and Modern C++ with Bjarne Stroustrup' [CppCast365] and *ADSP*'s 'Rust & Safety at Adobe with Sean Parent' [ADSP160]. They are all worth listening to.

Putting it all together

From correctness to safety

Let's assume that we have a complete specification for a program we want to build. We consider a program to be *functionally correct* if for every input given to the program, it produces an output, and that output satisfies the specification; this is sometimes referred to as *total correctness*. A program is deemed *partially correct* if, for every input, should the program produce an output, this output must conform to the specification. Note that partial correctness does not guarantee termination, which makes it easier to reason about.

Ideally, programs would be *functionally correct*. However, ensuring this for most programs is not feasible. In fact, even achieving *partial correctness* is a challenge for many programs. Furthermore, defining a complete specification for a problem is often as complex as implementing the solution itself. Therefore, having completely correct programs is not practically achievable. But this does not mean we should resign ourselves to letting our programs behave unpredictably. We must constrain them to ensure only a reasonable set of outcomes are possible.

Consider a self-driving car that needs to travel from point A to point B. We cannot guarantee the car will complete its journey since it might break down. However, we aim to ensure that, under normal operating conditions, the car doesn't, for instance, continuously accelerate uncontrollably or start driving off-road. This leads us to express guarantees in terms of 'X shouldn't happen'.

Sean [Parent23a] references a paper by Leslie Lamport [Lamport77] which suggests breaking down correctness into two types of properties: *safety properties* (what must not happen) and *liveness properties* (what must happen). Thus, our first definition of safety is that aspect of correctness concerned with what must not happen. In the example above, not continuously accelerating and not driving off-road are safety properties.

Because there are potentially an infinite amount of negative properties, having a clear definition of safety is not possible. We might have different types of safeties, and we should always keep in mind the goals of our programs. As Sean and Timur argue, safety is just an illusion [Parent23a, Doumler23a]. There are always limitations to what safety properties can express.

All safety properties should, in one way or another, contain the condition 'if operating under intended usage parameters'. For example, there is no guarantee that a software can make if the hardware misbehaves. For example, a car may continuously accelerate if the cosmic rays make the output of the hardware to continuously accelerate (ignoring any input from the software); or, a car may drive off-road if it's teleported outside the road while having high speed. Due to practical reasons, we should always assume our safety properties are qualified to exclude unintended usage behaviours.

If we want safety properties (that are not defined in probabilistic terms) to always hold, then, this definition of safety excludes gradual safety adoption that Herb was advocating for.

Once we address the issue of intended usage, we must consider the implications of any safety property for a program. The presence of even a single non-probabilistic safety property implies that undefined behaviour in our programs is unacceptable. Undefined behaviour means anything can happen, including violations of the safety property. Therefore, discussing safety in a system where undefined behaviour is possible is fundamentally flawed.

Functional safety

There is another path that leads to defining safety, more specifically called *functional safety*, coming from regulated industries, like automotive. Andreas and JF provide a good overview of functional safety [Weis23, Bastien23]. I will slightly alter the definition to make it more general.

We will define *harm* as physical, moral or financial injury or damage to individuals or companies. In automotive, this is typically defined as "physical injury or damage to the health of persons" [Weis23]. If a software leads to moral injuries or leads to customers losing money, by our definition, this will be called *harm*. Following Andreas, we will then define *risk* as being the "probability of the occurrence of harm weighted by the severity", and *unreasonable risk* as the risk that is "unacceptable according to societal moral concepts". This leads us to define *safety* of a system as the process of ensuring the absence of unreasonable risk.

These are several points to notice about this way of defining safety. First, we define safety at the system level; that means that we might have unsafe components in the system, if, overall, the system is safe. Then, we are talking about processes; this implies that C code, which theoretically contains undefined behaviour, can be rendered safe by applying processes

that (probabilistically) ensure that undefined behaviour does not occur in practice. Finally, this definition uses the probability of harm occurrence and "societal moral concepts", rendering the entire definition subjective.

The subjectivity of this definition of safety makes it harder to work with in practice, and especially if we want to apply safety at the programming language level. Andreas outlines a thorough process by which car manufacturers can certify their systems for functional safety, but this approach may be too heavyweight.

On the other hand, the main thing that I like about this form of defining safety is that it revolves around the *why?* question. It tells us why it's important to have safety guarantees, and lets us choose which guarantees we should have, and allows us to prioritise safety guarantees. If, with our previous definition of safety, we are allowed to select any negative property as being part of safety, this definition encourages us to consider the important properties.

The reader should note that this definition of safety is equivalent to the first definition of safety if we properly express the probabilities and the 'societal moral concepts' into the negative properties.

Security

While there is general consensus that safety and security are interrelated, the nature of their relationship is a subject of debate among C++ experts [Steagall23]. By the two definitions we've listed above, security needs to be a part of safety.

For simplicity, let's define security as the protection of software systems from malicious attacks that may result in unauthorised information disclosure or any other damage to a software system. This definition of security is a property of the software system of things that are not allowed to happen; thus security is part of safety.

Coming from the second definition of safety, we can say that a security attack is producing harm, so preventing this harm is part of safety.

To classify security as a subset of safety, the system's specifications must align with safety principles. For instance, if a program's specifications permit unauthenticated access to data, this could create a conflict between safety and security. However, if both safety and security are defined solely in the context of the program's specifications, then there should be no discrepancy between the two.

Another important discussion point that also applies to safety is the above *intended use*. Sometimes, security issues don't stem directly from the software itself, but from the surrounding system. For example, a security breach might be feasible due to insufficient security at the hardware level. Such scenarios fall outside the scope of the safety properties that can be ascribed to the software system itself. However, if the software in question amplifies the damage produced (compared to that is reasonably feasible), the software may still be considered unsafe.

Regulation and ethical perspective

Starting from the recent reports on safety and security [NSA22, CR23, WH23a, EC22, CISA23a, CISA23b], many authors we cited here believe that our software industry needs to be regulated.

Consider the scenario where someone buys a phone that fails to function properly; typically, the customer is entitled to return the phone and receive a replacement, depending on the country's consumer protection laws. However, if a software update bricks the phone, the software company is often not obligated to rectify the issue, as noted in a story from Conor Hoekstra [ADSP160]. This situation seems unjust, particularly given the ubiquity of software and its increasing significance. Therefore, it is anticipated that the software industry will eventually be held accountable for its failures.

Indeed, the *National Cybersecurity Strategy* document issued by the White House [WH23a] has, among others, the following strategic objectives: "Hold the Stewards of Our Data Accountable", "Shift Liability for Insecure Software Products and Services", and "Leverage Federal Procurement to Improve Accountability". In Europe, the European Commission produced a proposal [EC22] that states:

> It is necessary to improve the functioning of the internal market by laying down a uniform legal framework for essential cybersecurity requirements for placing products with digital elements on the Union market.

The same issue can also be seen from an ethical perspective. Buggy software can produce harm (physical, moral, or financial). The engineers who produced and/or allowed those bugs are morally responsible for the harm they produce. And we can conclude – according to our second definition of safety, i.e., the bugs that create harm are safety issues – that engineers are morally responsible for safety issues.

Safety for programming languages

It's not obvious how the notion of safety applied to systems or to programs applies to programming languages (we will mainly focus here on system programming languages). The fact that a software system must not exhibit a specific problem doesn't mean that the language should prevent this problem or that there can't be sub-systems that have this problem. For example, the software system may detect faults and control their impact.

However, from a practical standpoint, it makes sense to add as many guarantees as possible to the language level so that we don't spend too much energy addressing safety issues at the system level. For example, it's a very hard problem to mitigate undefined behaviour from one component to make the entire system safe; after all, undefined behaviour can mean *tricking the rest of the system that everything is fine while covering a safety issue*.

While at the language level we can't guarantee all the safety properties, there are a few safety properties that we can enforce from that very level. We can enforce the absence of undefined behaviour (which includes memory safety, thread safety and arithmetic safety).

And, while we are here, I'll attempt to provide a different definition of memory safety that doesn't rely on enumerating different types of issues (type safety, bounds safety, lifetime safety, initialisation safety). We can define memory safety as the absence of undefined behaviour caused by accessing memory (for reading or writing). Other authors (for example, see [Doumler23a]) put type safety as part of memory safety. With my definition of memory safety, one can have type safety issues without having memory safety issues.

Avoiding deadlocks is another safety property that some programming languages strive for; there are solutions that provide general concurrency mechanisms while avoiding deadlocks, but these solutions are not widely deployed.

In addition to the absence of undefined behaviour, a safe programming language may also provide mechanisms for dealing with failed preconditions. When preconditions are not satisfied, there is a bug in the program. The most reasonable path forward for the software is to terminate or raise an error; the idea is to immediately get outside the area in which the bug was detected. See [Parent22] for an insightful discussion on this topic.

A safe language ensures that safety properties always hold. However, such guarantees can limit the language's expressiveness; there are safe constructs that the compiler cannot definitively verify as safe. Therefore, a common strategy in language design is to divide the language into two parts: a *safe* subset (where the compiler verifies safety properties) and an *unsafe* subset (where compiler guarantees are relaxed).

With all these, we can define a *safe programming language* as a programming language that:

- has a *safe* subset of the language that guarantees:
  - no undefined behaviour
  - type safety

- (optional) absence of deadlocks
- (optional) safe handling of precondition failure when detected

- makes the *safe* subset distinct from the rest of the language (*unsafe* subset), and that this distinction is visible and auditable

Achieving a *safe programming language* is feasible. From an ethical perspective, it is also desirable. We are beginning to see examples of C++ components that require efficiency being successfully rewritten in Rust with minimal efficiency loss. Thus, my conclusion is that a larger portion of the systems programming community will likely (and, for ethical reasons, also should) shift towards safe programming languages, whether partially or entirely.

Quo vadis, C++?

C++ has too much undefined behaviour to become a safe programming language in the foreseeable future. One way or another, all the C++ experts cited here agree on that. This means that C++ can only make partial improvements towards the direction of a safe programming language; while it may address some safety issues, it can never guarantee basic safety.

We've already seen that companies and products are seriously considering moving parts of their software away from C++ to Rust. The main question is whether C++ can do something to stop this trend, and maybe reverse it. Personally, I am not convinced that in the near future, C++ can do something to stop this trend. C++ will leak talent to other languages (currently Rust, but perhaps in the future to Cppfront, Carbon, Hylo or Swift). If the progress towards safety started in 2015 as Bjarne suggested, the last 8 years have seen very little progress in safety improvements. Even with accelerated efforts, the three-year release cycle and slow adoption of new standards will keep C++ a decade away from addressing major safety concerns.

If safety remains as critical as it is today, then C++ will bleed engineers, reducing its importance in programming language landscape. However, after some time, there will be a point in which this bleeding will not be significant any more. We still have Cobol and Fortran code being maintained, so we cannot expect C++ to simply disappear. The key questions are: how long will this transition take and how much of core C++ will remain when the bleeding is over?

The answers depend on C++'s ability to restore credibility among its core user base. Again, all things equal, I believe the transition will probably take more than a decade; during this time, C++ usage will decrease, especially for new software, potentially relegating C++ to a language for legacy code. If this happens, it could start a negative reinforcement loop, making C++ even less attractive for new projects.

A factor that could halt this process is the inability of other languages to effectively operate in the systems programming space. Rust is relatively young and not fully vetted; if it proves inefficient, this might slow or halt the migration from C++ to Rust. The stability of the language and ecosystem is another crucial factor.

One major selling point for C++ for new projects is efficiency. People are still skeptical that other languages can truly compete with C++ in that space. For example, people that need low latency (game development, trading, etc.) or are into HPC, may never be convinced to switch away from C++ because of this reason. We still don't have enough data to argue if another language that provides safety can compete with C++ in this area.

There is another reason that may slow down the transition away from C++. The inertia of some industries. Just like some industries/companies were very slow to move from C to C++, there may be companies that invested so heavily in C++ that they can't afford to switch to another language.

A more likely scenario is the segregation of C++ codebases and the rewriting of parts in other languages. Some components might continue to be written in C++, while others, where safety is more critical than peak performance, could migrate to safer languages. This is similar to the approach taken by Microsoft, Linux, and Adobe, which are starting to migrate parts of their codebase to Rust.

Assuming there is a migration away from C++, another important question is: How would migration happen? We may see multiple ways of approaching this, from migrating entire systems, to incrementally migrating components of a bigger system and even to smoother transitions similar to the ones that Chandler and Herb discuss [Carruth23, Sutter23a]. What I would envision is that soon we will see tools appearing that help migrate C++ code to safer languages.

Only time will tell how all this will evolve. While a decade may seem long, it's also just around the corner. Our focus will soon shift to other pressing topics, and before we know it, we may see a predictable safety narrative for C++ and other languages.

Reasonable safety

Safety is no longer a luxury. As the world increasingly depends on software, the importance of software safety cannot be overstated. Therefore, consumers are justified in expecting software safety to be a given. This places a responsibility on us to ensure safety is guaranteed.

To achieve safety for programs written in languages like C++, we rely on heavyweight processes. The more we shift left these guarantees to the compilers, the easier it would be for us to provide safety.

Programming language safety is not any more something difficult to achieve. We have proven experience of languages that are safe by default, i.e., that avoid undefined behaviour, while being also efficient.

Our aspiration is for all our software to be reasonable – easy to understand, reliably good, and free of surprises. We want safety to be reasonable, we want safety to be the default setting in a programming language. This should be a right, of both consumers and us, the programmers. ■

References

[ADSP160] Conor Hoekstra, Bryce Adelstein Lelbach, Sean Parent, 'Rust & Safety at Adobe with Sean Parent', *ADSP: The Podcast*, episode 160, Dec 2023, https://adspthepodcast.com/2023/12/15/Episode-160.html

[Amorim18] Arthur Azevedo de Amorim, Cătălin Hriţcu, Benjamin C. Pierce. 'The meaning of memory safety', *Principles of Security and Trust: 7th International Conference*, POST 2018, Held as part of the *European Joint Conferences on Theory and Practice of Software, ETAPS* 2018, Apr 2018.

[Bastien23] JF Bastien, 'Safety and Security: The Future of C++', *C++ now*, May 2023, https://www.youtube.com/watch?v=Gh79wcGJdTg

[Black21] Paul E. Black, Barbara Guttman, Vadim Okun, 'Guidelines on Minimum Standards for Developer Verification of Software', National Institute of Standards and Technology, NISTIR 8397, Oct 2021, https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf

[Carruth23] Chandler Carruth, 'Carbon's Successor Strategy: From C++ interop to memory safety', *C++ now*, May 2023, https://www.youtube.com/watch?v=1ZTJ9omXOQ0

[CISA23a] Cybersecurity and Infrastructure Security Agency, 'Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and -Default', Apr 2023, https://www.cisa.gov/sites/default/files/2023-04/principlesapproachesforsecurity-by-design-default5080.pdf

[CISA23b] Cybersecurity and Infrastructure Security Agency, 'Secure by Design', Apr 2023, https://www.cisa.gov/securebydesign

[Claburn23] Thomas Claburn, 'Memory safety is the new black, fashionable and fit for any occasion', *The Register*, Jan 2023, https://www.theregister.com/2023/01/26/memory_safety_mainstream/

[CppCast356] Timur Doumler, Phil Nash, Andreas Weis, 'Safety Critical C++', *CppCast*, episode 356, Mar 2023, https://cppcast.com/safety-critical-cpp/

[CppCast365] Timur Doumler, Phil Nash, Bjarne Stroustrup, 'Safety, Security and Modern C++, with Bjarne Stroustrup', *CppCast*, episode 365, Jul 2023, https://cppcast.com/safety_security_and_modern_cpp-with_bjarne_stroustrup/

[CR23] Yael Grauer (Consumer Reports), 'Future of Memory Safety: Challenges and Recommendations', Jan 2023, https:// advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report.pdf

[Dhalla23] Amira Dhalla, 'Fireside Chat: The State of Memory Safety, with Yael Grauer, Alex Gaynor, Josh Aas, *USENIX Enigma 2023*, Feb 2023, https://www.youtube.com/watch?v=b1I8qGYCx3c

[Doumler23a] Timur Doumler, 'C++ and Safety', *C++ on Sea*, Jun 2023, https://www.youtube.com/watch?v=imtpoc9jtOE

[Doumler23b] Timur Doumler, 'C++ and Safety', *CppNorth*, Jul 2023, https://www.youtube.com/watch?v=iCP2SFsBvaU

[EC22] European Commission, 'Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on horizontal cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020', Document 52022PC0454, Sep 2022, https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52022PC0454&qid=1703762955224

[Gaynor18] Alex Gaynor, 'The Internet Has a Huge C/C++ Problem and Developers Don't Want to Deal With It', *Vice*, 2018, https://www.vice.com/en/article/a3mgxb/the-internet-has-a-huge-cc-problem-and-developers-dont-want-to-deal-with-it

[Horvath23] Gabor Horvath, 'Lifetime Safety in C++: Past, Present and Future', *CppCon*, Oct 2023, https://www.youtube.com/watch?v=PTdy65m_gRE

[Hubert23] Bert Hubert, 'The EU's new Cyber Resilience Act is about to tell us how to code', Mar 2023, https://berthub.eu/articles/posts/eu-cra-secure-coding-solution/

[ISO26262] ISO, 26262:2018 'Road vehicles – functional safety', 2018.

[Lamport77] Leslie Lamport. 'Proving the correctness of multiprocess programs', *IEEE transactions on software engineering* 2, 1977, https://www.microsoft.com/en-us/research/publication/2016/12/Proving-the-Correctness-of-Multiprocess-Programs.pdf

[Lamport83] Leslie Lamport, 'What good is temporal logic?' *IFIP congress*, 1983, http://lamport.azurewebsites.net/pubs/what-good.pdf

[McCall17] John McCall, 'Swift ownership manifesto', 2017. https://github.com/apple/swift/blob/main/docs/OwnershipManifesto.md

[Milner78] Robin Milner, 'A theory of type polymorphism in programming', *Journal of computer and system sciences*, 1978, https://www.sciencedirect.com/science/article/pii/0022000078900144/pdf?md5=cdcf7cdb7cfd2e1e4237f4f779ca0df7&pid=1-s2.0-0022000078900144-main.pdf&_valck=1

[NSA22] National Security Agency, 'Software Memory Safety', Nov 2022, https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSISOFTWAREMEMORYSAFETY.PDF

[OpenSSF22] OpenSSF, May 2022, https://openssf.org/oss-security-mobilization-plan/

[P2410R0] Bjarne Stroustrup, 'P2410R0: Type-and-resource safety in modern C++', WG21, Jul 2021, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2410r0.pdf

[P2687R0] Bjarne Stroustrup, Gabriel Dos Reis, 'P2687R0: Design Alternatives for Type-and-Resource Safe C++', WG21, Oct 2022, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf

[P2739R0] Bjarne Stroustrup, 'P2739R0 : A call to action: Think seriously about "safety"; then do something sensible about it', WG21, Dec 2022, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2739r0.pdf

[P2759R1] H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong, 'P2759R1: DG Opinion on Safety for ISO C++', WG21, Jan 2023, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2759r1.pdf

[P2816R0] Bjarne Stroustrup, 'P2816R0: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++', WG21, Feb 2023, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2816r0.pdf

[Parent22] Sean Parent, 'Exceptions the Other Way Round', *C++ now*, May 2022, https://www.youtube.com/watch?v=mkkaAWNE-Ig

[Parent23a] Sean Parent, 'All the Safeties', *C++ now*, May 2023, https://www.youtube.com/watch?v=MO-qehjc04s

[Parent23b] Sean Parent, 'All the Safeties', C++ on Sea, Jun 2023, https://www.youtube.com/watch?v=BaUv9sgLCPc

[Seacord23] Robert Seacord, 'Safety and Security: The Future of C and C++', *NDC TechTown*, Sep 2023, https://www.youtube.com/watch?v=DRgoEKrTxXY

[Steagall23] Bob Steagall, 'Coding for Safety, Security, and Sustainability', panel discussion with JF Bastien, Chandler Carruth, Daisy Hollman, Lisa Lippincott, Sean Parent, Herb Sutter, *C++ now*, May 2023, https://www.youtube.com/watch?v=jFi5cILjbA4

[Stroustrup23a] Bjarne Stroustrup, 'Approaching C++ Safely', *Core C++*, Aug 2023, https://www.youtube.com/watch?v=eo-4ZSLn3jc

[Stroustrup23b] Bjarne Stroustrup, 'Delivering Safe C++', *CppCon*, Oct 2023, https://www.youtube.com/watch?v=I8UvQKvOSSw

[Sutter22] Herb Sutter, 'Can C++ be 10× simpler & safer…? *CppCon*, Oct 2022, https://www.youtube.com/watch?v=ELeZAKCN4tY&list=WL

[Sutter23a] Herb Sutter, 'Fill in the blank: _____ for C++', *C++ now*, May 2023, https://www.youtube.com/watch?v=fJvPBHErF2U

[Sutter23b] Herb Sutter, 'Cooperative C++ Evolution: Toward a Typescript for C++', *CppCon*, Oct 2023, https://www.youtube.com/watch?v=8U3hl8XMm8c

[Weis23] Andreas Weis, 'Safety-First: Understanding How To Develop Safety-critical Software', *C++now*, May 2023, https://www.youtube.com/watch?v=mUFRDsgjBrE

[WH23a] White House, 'National Cybersecurity Strategy', Mar 2023, https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf

[WH23b] White House, 'National Cybersecurity Strategy' (press release), Mar 2023, https://www.whitehouse.gov/briefing-room/statements-releases/2023/03/02/fact-sheet-biden-harris-administration-announces-national-cybersecurity-strategy/

Afterwood

Over-thinking is not over-engineering. Chris Oldwood presents some thought experiments to demonstrate why.

As the pendulum swings ever closer towards being leaner, and focusing on simplicity, I grow more concerned about how this is beginning to affect software architecture. By breaking our work down into ever smaller chunks and then focusing on delivering the next most valuable thing, how much of what is further down the pipeline is being factored into the design decisions we make today? And consequently, how much pain are we storing up for ourselves because we took the pejorative Big Design Up Front (BDUF) too far and ended up with No Design Up Front?

Wasteful thinking

Part of the ideas around being leaner is an attempt to reduce waste caused by speculative requirements which has led many a project in the past into a state of 'analysis paralysis' where they can't decide what to build because the goalposts keep moving or the problem is so underspecified there are simply too many options and we have to second-guess everything. By focusing on delivering something simpler, much sooner, we begin to receive an initial return on our investment earlier which helps shape the future design based on practical feedback from today, rather than guessing what we need.

When we're building those simpler features that sit nicely upon our existing foundations we have much less need to worry about the cost of rework from getting it wrong as it's unlikely to be overly expensive. But as we move from independent features to those which are based around, say, a new 'concept' or 'pillar' we should not be afraid to spend a little more time looking further down the product backlog to see how any design choices we are considering now, might play out later. Emergent Design is not a Random Walk but a set of educated guesses based on what we currently know, and strongly suspect about the near future.

Thinking to excess?

The term 'overthinking' implies that we are doing more thinking than is actually necessary; trying to fit everyone's requirements in and getting bogged down in analysis is definitely an undesirable outcome of spending too much time thinking about a problem. As a consequence, we are starting to think less and less up-front about the problems we solve to try and ensure that we only solve the problems we actually have and not the problems we think we'll have in the future.

Solving problems that we are only speculating about can lead to overengineering if they never manage to materialise or could have been solved more simply when the facts are eventually known.

But how much thinking is overthinking? If I have a feature to develop and only spend as much effort thinking as I need to solve that problem today then, by definition, any more thinking than that is 'overthinking it'. But not thinking about the wider picture is exactly what leads to the kinds of architecture and design problems that begin to hamper us later in the product's lifetime, and later on might not be measured in years but could be weeks or even days if we are looking to build a set of related features that all sit on top of a new concept or pillar.

Building the simplest thing that could possibly work does not mean being naïve about the future.

The thinking horizon

Hence, it feels to me that some amount of overthinking is necessary to ensure that we don't prematurely pessimise our solution and paint ourselves into a corner too quickly. As such, we should factor in related work further down the backlog, at least into our thoughts, to help us see the bigger picture and work out how we can shape our decisions today to ensure it biases our thinking towards our anticipated future rather than an arbitrary one.

Acting on our impulses prematurely can lead to overengineering if we implement what's in our thoughts without having a fairly solid backlog to draw on, and overengineering is wasteful. In contrast, a small amount of overthinking – thought experiments – is relatively cheap and can go towards helping to maintain the integrity of the system's architecture by narrowing the solution space to something more realistic. Very few software products have the need to scale to anything like what you read about in the technology news pages, despite what the optimists in the business might have you planning for.

The phrase 'think globally, act locally' is usually reserved for talking about the health of the planet, but I think it is fractal in nature, in that you can also apply it at the software system level too, to suggest factoring in thinking about the design and architecture of the system even though you are only implementing a feature in a small part of it.

One has to be careful quoting old adages like 'a stitch in time saves nine' or 'an ounce of prevention is worth a pound of cure' because they can send the wrong message and lead us back to where we were before – stuck for eternity in The Analysis Phase. That said, I also want us to avoid 'throwing the baby out with the bathwater' and end up forgetting exactly how much thinking is required to achieve sustained delivery in the longer term. ■

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

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

AccU 2024

Conference 17 – 20 April 2024
Pre-conference workshops 15 & 16 April 2024

REGISTRATION NOW OPEN! Visit https://accuconference.org/