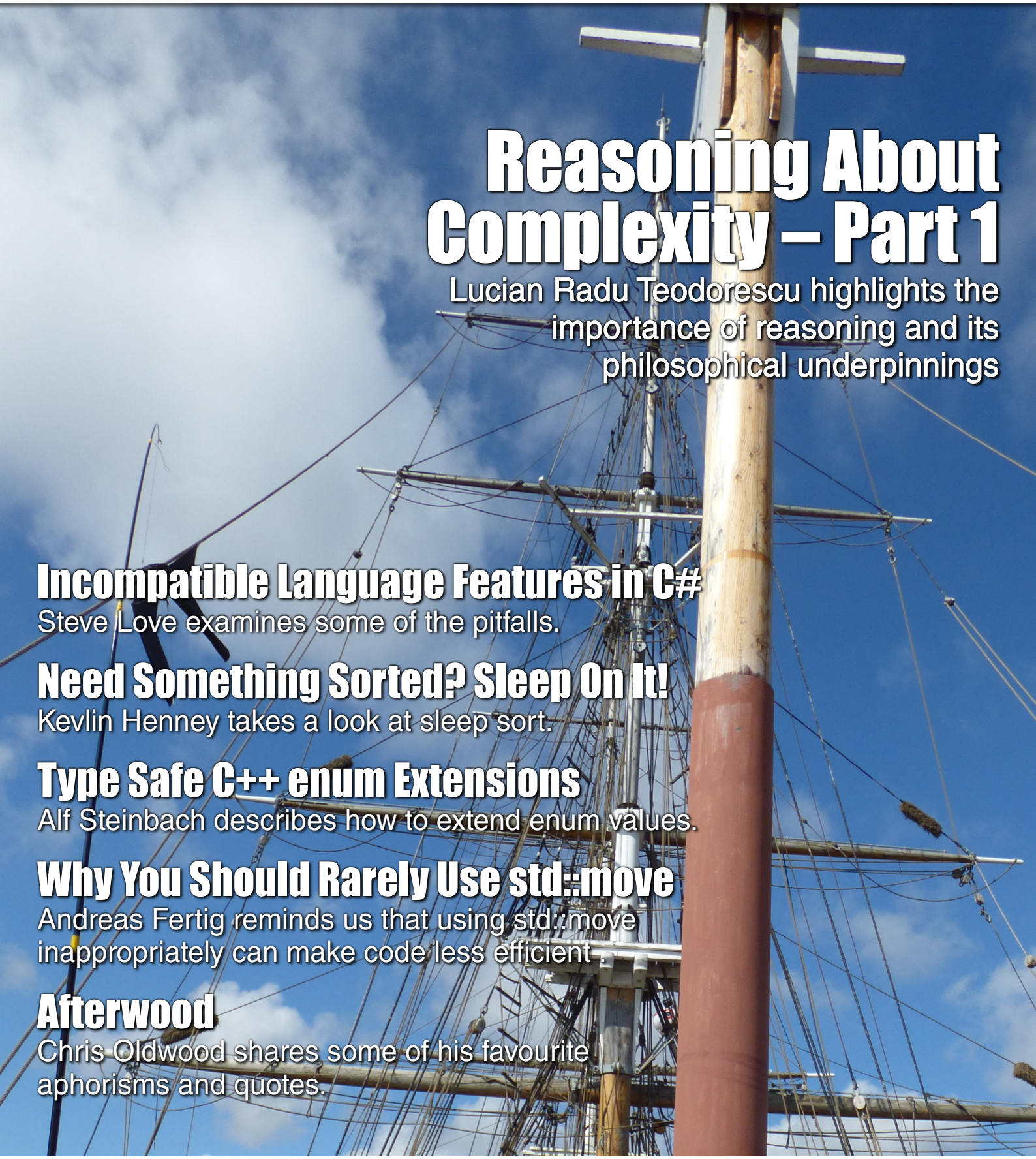
overload 175

JUNE 2023 £4.50



Reasoning About Complexity – Part 1

Lucian Radu Teodorescu highlights the importance of reasoning and its philosophical underpinnings

Incompatible Language Features in C#

Steve Love examines some of the pitfalls.

Need Something Sorted? Sleep On It!

Kevlin Henney takes a look at sleep sort.

Type Safe C++ enum Extensions

Alf Steinbach describes how to extend enum values.

Why You Should Rarely Use std::move

Andreas Fertig reminds us that using std::move inappropriately can make code less efficient.

Afterwood

Chris Oldwood shares some of his favourite aphorisms and quotes.

A magazine of ACCU

ISSN: 1354-3172

June 2023

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Tim Peck. Replica 'famine ship', Dublin, Ireland.

Copy deadlines

All articles intended for publication in *Overload* 176 should be submitted by 1st July 2023 and those for *Overload* 177 by 1st September 2023.

The ACCU

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

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

Production and Productivity

How can you increase your productivity? Frances Buontempo discovers that ChatGPT might not help.

I recently returned from this year's ACCU conference, which was lots of fun and gave me loads to think about. Unfortunately, this has stopped me from even considering what to write for an editorial. It's difficult to produce something to order, by a deadline, at the best of times. I would claim to have resisted the temptation to ask ChatGPT [OpenAI-1] to write an editorial for me, but am currently arguing with it about economics. I asked about production and productivity after all, so shouldn't be surprised the chat was not about coding. After much discussion on employment, economic growth and automation to replace humans, ChatGPT told me

> As technology advances, the role of software developers will evolve, but it is unlikely that machines will completely replace human coders.

Now, I mustn't waste all day arguing with an AI chat bot. Distractions can eat into our time, and there's never enough time.

How do you avoid distractions, and stay focused? I am told there are many productivity influencers 'out there'. Chris Bailey has a few talks based around his books on hyper-focus and the productivity project. He has had several million views on YouTube. I hope they weren't all by the same person, because if they were, they were clearly very distracted at the time. Apologies for flippancy. Sometimes you need to stop the distractions, but listening to YouTube on a connected device often means I spin up other things, so I only listened to a few minutes before getting distracted by something else. Being at a talk at a conference is a different experience, perhaps because I don't have my usual distraction around me. I absolutely can focus, but it's nice to allow your mind to wander as people talk. Sometimes you need space to let your mind drift. Slowing down and noticing your surroundings is grounding. You give space for ideas, and time to recall things you may have forgotten to do. We all need to rest sometimes. To an observer, this may not look very productive, but downtime matters and can make you more productive in the long run.

Some productivity hacks from so-called influencers can be interesting, but what works for one person may not work for another. I've recently heard mention of the '5am club'. Robin Sharma introduced the idea twenty or more years ago, and wrote a book with the subtitle 'Own your morning, elevate your life' in 2018. [Sharma18]. Many people seem to be trying a version of the suggested approach of getting up, far too early to my mind, and scheduling blocks of 20 minutes to do various tasks. I suspect that different approaches work for different people. There are lots of other 'productivity' blogs and posts out there. May Pang [Pang23] wrote a Medium article, explaining why she decided to defy the 5am club. She claims, "I watched with glee as my productivity continued to skyrocket with each conventional productivity rule I challenged that didn't feel right for me." Most productivity articles make big claims. She does sensibly point out circadian rhythms differ. Some people are productive early in the morning and some are not. Take anything that claims it will change your life, or solve your problems for you, with a pinch of salt.

Another fad attracting attention is ChatGPT. I lost a considerable amount of time arguing with it this morning. OpenAI's blog [OpenAI-2] tells me ChatGPT admits its mistakes. Try asking it to write C++ coroutine code for you. It did say "You are correct, sorry for my mistake" or "You're correct. I apologize for the oversight." on many occasions. It can generate code, which may not compile. It can even generate tests for its own code. Would you trust something that marked its own homework? OK, to be fair, developers often write tests for their own code, so perhaps that's the wrong question. I can imagine letting AI generate some boilerplate code for a simple task might be acceptable. You can even wire ChatGPT into your IDE, via Copilot [Dias23]. You add a comment, and a little frog icon has a 'think' then offers suggested code. If you leave the comment behind, others will know how the code was generated. I presume later versions, trained on different data, may come out with alternative suggestions, so maybe the comment will become obsolete at some point, as most comments do. Whether this counts as AI writing code for us, I'm not sure. It's doing some kind of statistical based guess. I wrote about AI code generation a while ago [Buontempo21]. I used genetic programming (GP), rather than a neural network. GP is like genetic algorithms, but uses tree structures, and so can generate ASTs and therefore programmers. Unlike ChatGPT, my code worked, though maybe Fizz Buzz is easier than C++ coroutines? If you don't know about genetic algorithms, buy my book, or read the Overload article I wrote 'How to Program Your Way Out of a Paper Bag Using Genetic Algorithms' [Buontempo13]. In some ways, using AI generated code is similar to copying code from the internet or even documentation. However code is generated, it's always worth looking at the code and seeing if it is easy to understand, in case you ever need to revisit it. At very least, check it compiles and does what you want.

Some boilerplate code is tedious to write, and it might be more productive to auto-generate such code. Kate Gregory gave the closing keynote at this year's ACCU conference. Her talk was called 'Grinding, Farming, and Alliances, How words and ideas from casual gaming can make you a better programmer'. She drew analogies between games and programming. Grinding meant tedious work, like clicking lots of buttons to get a reward, and seems very similar to writing boilerplate code, to my mind. Kate confessed to outsourcing some of the grinding work, getting a child to click buttons in a game, because the young girl loved seeing hearts materialize and float up the screen. A productivity win-win. When

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.

I did my post-doc, the lecturers often tried to outsource tedious work to a PhD student. They thereby avoided the hard grind of experiments, and the student got results they could use towards their qualification. If someone other than you might get some benefit or reward from doing a task, even if it might be a long slog, delegating might be the productive way to go. Of course, if you have lots of tasks to complete, hiring loads of minions to do the work might not be optimal. We are all aware of *The Mythical Man Month* by Fred Brooks. If you double the number of people on a project, the time to completion is unlikely to halve. He died at the end of last year, aged 91. He made "landmark contributions to computer architecture, operating systems, and software engineering" [ACM]. I believe the 8-bit 'byte' may be due to him. Sometime little things make a big difference. This seemingly small contribution allowed upper and lower case letters to be represented easily, allowing the use of computers in text processing. Whether that directly caused large language models and ChatGPT is another matter.

So much for productivity. Let's think about production. As much as you prepare for something to go live, things can always go wrong. I gave a talk at the ACCU conference, which I had practised a few times. My laptop powered off live on stage and refused to start up again, until the following day. I had a copy of the talk in BitBucket, but they have recently rearranged the UI, so when I logged in on someone else's laptop, I couldn't see any of my repos. Fortunately, someone in the audience had recently had a similar problem, and managed to help. Several minutes later, I just about managed to continue where I left off, though some of the fonts weren't supported on the other laptop and I had to remember what various bits of maths on some slides should have said. I also didn't have time to build the demos I had prepared. Now, I could have had a pdf of the slides to avoid the font issue, and saved screen-casts of the demos, but whatever I had prepared would still have left open the possibility of yet more things going wrong. Sometimes you can't prepare for every possible scenario, so it's more important to be able to think on your feet and form a Plan B on the spot. Monitoring what's happening live on stage, or in production, is important. The time to recovery is often the best metric. Preparing for every eventuality is not productive. Having a backup plan or two is sensible, but you cannot fool-proof anything completely. Overthinking, whether it be of potential disasters or normal use-cases, is often a waste of time. The phrase analysis paralysis [Wikipedia] springs to mind. Chris Oldwood recently reminded me of one of his blogs about overthinking [Oldwood18]. He points out:

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

However, he also asked how much thinking is over-thinking? He points out thinking about the big picture, and trying a few thought experiments can be relatively cheap and help the system in the long run. Seeming to be productive, and bashing out Jiras or ticking things off a TODO list is all very well, but sometimes pausing for a moment is more productive than just producing a stream of code. Stopping to think about what you are producing and why, and even how, can be very useful.

Perhaps we can't quantify overthinking, but we still use the phrase. We also describe some code as over-engineered. Using factories, dependency injection frameworks and the like when a simple script would suffice certainly seems like over-engineering. I've recently been reading *The Art of Darkness: The History of Goth* by John Robb. It reminded me of several bands I had forgotten about, and I even dusted off some old vinyl and chilled over some music. John Robb describes some albums as overproduced, a phrase I had forgotten. Now, we'd never describe production code as overproduced. How do you expunge spontaneity or artistry from code? Mind you, a company may insist on certain code layouts and similar, so removing any hint of personal style. Perhaps overproduced code is a good thing? Reading code, or even a book, that switches between personal styles can be distracting. Over-production would be an entirely different matter. If you over-produce a commodity, you create too much of it, and possibly reduce its value. Can you over-produce code? Maybe, maybe not, but you can certainly over-complicate code. Writing less, but clearer, code is harder than cranking out lines and lines that work, but are hard to make sense of. I suspect AI will never generate short, clear code, unless we get better at writing our requirements. If we never read the code, this won't matter, but that's a long way off.

So, back to ChatGPT. Is it OK to use AI to write code for us? Yes, it is. It's OK to look things up on the internet or in a book if we need to, as well. If you regard AI generated code in the same vein, being careful to access what you are presented with, that's fine. Copilot and similar are often heralded as being able to help us work smarter and harder. Being able to look something up is often the smart thing to do. Whether AI really helps you be more productive is for you to decide. Don't go down the rabbit hole I fell into, trying to get ChatGPT to register it can't put `co_await` in `main`. As ChatGPT apologised and retried over and over again, I almost ended up willing it to succeed. I would have been more productive if I had written the code myself several times over, or done some housework, or written an editorial. The trouble is, feeling like you are nudging something close to success makes it very hard to stop. Gail Ollis gave Thursday's keynote at the conference about what she terms 'humaning'. She asked attendees to pop questions on a card the day before that she would try to answer. One question was why do coders get sucked into something and forget to eat or sleep. I didn't get that sucked into arguing with the AI, but I did lose some hours. Gail suggested a quest for a dopamine reward can drive us into sticking with a specific task for far too long, eventually losing the plot. We do need to eat and sleep. We're only human, after all. We can't be productive all the time, and some things never even make it to production. The important thing should be having fun, and learning from what we do. ChatGPT certainly hasn't learnt it can't put `co_await` in `main`, but I have.

References

[ACM] A.M.Turing Award to Frederick (Fred) Brooks for 'landmark contribution to computer architecture, operating systems and software engineering': https://amturing.acm.org/award_winners/brooks_1002187.cfm

[Buontempo13] Frances Buontempo (2013) 'How to Program Your Way Out of a Paper Bag Using Genetic Algorithms' Dec 2013, *Overload* 118, available at: https://accu.org/journals/overload/21/118/overload118.pdf#page=8

[Buontempo21] Frances Buontempo (2021) 'Teach Your Computer to Program Itself" in *Overload* 164, Aug 2021 https://accu.org/journals/overload/29/164/overload164.pdf#page=21

[Dias23] Chris Dias (2023) 'Visual Studio Code and GitHub Copilot' https://code.visualstudio.com/blogs/2023/03/30/vscode-copilot

[Oldwood18] Chris Oldwood (2018) 'Overthinking is not Overengineering', published 7 December 2018 at https://chrisoldwood.blogspot.com/2018/12/overthinking-is-not-overengineering.html

[OpenAI-1] ChatGPT: https://chat.openai.com/

[OpenAI-2] OpenAI: https://openai.com/blog/chatgpt

[Pang23] May Pang (2023) '4 Rebellious Productivity Rules to Declutter Your Brain' published in *Better Humans* on 17 April 2023 https://betterhumans.pub/4-rebellious-productivity-rules-to-declutter-your-brain-190d554b4230

[Sharma18] Robin Sharma *The 5 AM Club: Own Your Morning. Elevate Your Life* Harper Thorsons, 2018

[Wikipedia] 'Analysis paralysis': https://en.wikipedia.org/wiki/Analysis_paralysis

Reasoning About Complexity – Part 1

Reasoning and understanding code have fundamental roles in programming. Lucian Radu Teodorescu highlights the importance of reasoning and its philosophical underpinnings.

This is a two-part article. I wanted to write an article about complexity in software engineering (in memory of Fred Brooks, 1932–2022), but then I realised the complexity of such an endeavour. Reasoning is hard, tackling complexity is also not easy, and, moreover, the relation between reasoning and complexity in software engineering deserves a lot of attention too. Thus, there are multiple subjects that are interconnected. And covering them doesn't fit the space of a single article.

The first part is structured as an essay. We will highlight the importance of reasoning in software engineering, and how our field is close to philosophy.

The second part (to appear in the next issue of *Overload*) is structured as a play in 14 acts. Here we tackle the problem of complexity, and discuss how much it is *essential* versus how much it is *accidental*.

The two parts are deeply related. While discussing complexity, we refer to the first part in two main ways: the approach itself involves a form of *pure* reasoning, and also the object of our study, i.e., dealing with complexity, involves reasoning.

Reasoning in software engineering

How do we reason about things in software engineering? Moreover, do we even need to reason about things, and, if so, to what extent do we need to reason about them?

Software engineering is, as the name says, an engineering discipline. There are people who argue differently, but I think we have sufficient evidence to say that it is indeed an engineering discipline [Wayne21a, Wayne21b, Wayne21c, Farley21]. Like any other engineering discipline, our field needs to be based on facts, and ultimately knowledge. We should be able to conduct experiments to acquire empirical evidence.

But, the sad reality is that we have a relatively small amount of definitive empirical knowledge in software engineering [Wayne19]. Moreover, we lack this knowledge for some fundamental aspects we are using. For example, we don't have good empirical knowledge on whether object-oriented programming is better than functional programming. There are multiple reasons for this, but probably the most important one is the fact that the most important instrument in developing software is our mind; not the compilers, not the programming languages, and not even the computers. And, our mind works in mysterious ways.

The main job while programming is not writing code, but reading, reasoning and understanding code; then, after one forms a mental model of how the program should be written, the job is to translate that mental model into a concrete form that both humans and computers can understand and reason about. As Kevlin usually puts it, programming is applied epistemology [Henney19]. And epistemology is a branch of philosophy.

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

Software engineering and philosophy

If there is one book that every programmer should read, then that book is *The Mythical Man-Month* by Fred Brooks. Especially the 1995 edition, which contains the *No Silver Bullet* article, published initially in 1986. In a nutshell, Brooks argues that *software is essential complexity*[1]. Ever since I read this, many years ago, it remained with me as one of the few fundamental ideas of software engineering.

However, we cannot actually prove that software is essential complexity, we can only argue that it must be correct. The core reason for which this statement cannot be proved is because it is a metaphysical statement, and we cannot prove metaphysical statements. In the article, Brooks mentions that he is following the distinction between essential and accidental that was made by Aristotle. This distinction appears in Aristotle's 'Metaphysics' [Aristotle-1].

This definition of software implies that there is a strong connection with philosophy at the core of software engineering.

Another very influential book is *Elements of Programming* by Alex Stepanov and Paul McJones [Stepanov09]. Reading it feels very similar to reading Aristotle. It has a similar reasoning style, it uses terms like *entity*, *species*, and *genus* that Aristotle introduces [Aristotle-1], and it even gives examples with Socrates (chapter 1); later on, in chapter 5.4, it directly refers to Aristotle's 'Prior Analytics' [Aristotle-2].

The important aspect to notice about *Elements of Programming* is how the foundations of programming are exposed: through philosophical reasoning.

All these seem to converge to the idea that philosophical reasoning is somehow fundamental to software engineering. Reasoning seems far more utilised in our field compared to having controlled experiments. While we are still in an engineering discipline, the fundamentals appear to be very close to philosophy.

The extent of reasoning

Now that we've established that at the core of software engineering there needs to be philosophical reasoning, let's look at some examples. We'll use these examples, somehow similar to a qualitative study, to infer some possible characteristics of the type of reasoning we need.

Functions should be small

> *The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.*
> ~ Robert C. Martin [Martin08]

1 Brooks actually did not put it this way, I just remember it like this, as it makes the problem even more fundamental/metaphysical. Brooks actually argued that software development consists of overcoming essential difficulties and accidental difficulties, and that our main challenge comes from essential difficulties. Hope that the reader would agree with me that the two variants share the same essence. More on the two variants in the second part.

there is a fundamental limitation of the brain that makes it harder for us to reason about systems made of too many parts

This is regarded as very good advice[2]. However, the way it is explained in the book doesn't convince me. The main reason for this is that the statement is not properly argued. One can easily read this advice as "Functions should be very small because Uncle Bob said so". In fact, I got an overall impression of the *Clean Code* book [Martin08] as being perhaps *too dogmatic*. This is not something that we can put at the foundation of software engineering.

If I need to reason about the size of the function, I would do it along these lines:

- the main activity in programming is reading and reasoning about code; thus, functions should be written in order to be easily reasoned about
- there is a fundamental limitation of the brain that makes it harder for us to reason about systems made of too many parts (studies show that we can hold in our mind about 7 different things at once)
- thus, in order to easily reason about functions, we should make them as small as to accommodate our limitation
- however, there is a downside to this as well: making small functions pushes the complexity outside the functions
- there is also a cost of abstraction that we need to pay – any abstraction makes the code harder to read and reason about
- thus, we should not make the functions small, but not too small to have too many extra abstractions and to push the complexity outside the functions.

This line of thought, at least the first part, can be found in [Seeman21]. The reader can see a big difference between the two stories, even if the result is (roughly) the same. One has fragile reasoning, and one has more reasoning. One doesn't touch on consequences, the other does.

As Hillel points out, the most popular paradigm in our software industry is *charisma-driven development*: we do things because high-profile speakers and authors are telling us what to do [Wayne19]. Instead of blindly following advice, we should look at the arguments behind the advice. In lack of a better empirical evidence, we should at least evaluate whether the argument makes sense, and only follow the advice if the argument is convincing.

Design choices

One thing that I've learned in my career is that there isn't a perfect solution to a problem. Every solution consists of a (potentially long) series of design choices, and each choice has advantages and disadvantages. That is, for every design choice, one can argue in favour, or against it.

Moreover, on each side, there isn't just a simple argument that one can make. Usually, we have a long chain of assumptions and inferences that lead to arguing pro or against a design choice.

The way I would like to put it is that *if one cannot argue on both sides of an argument, one must be confused*.

To choose an adequate solution for a problem, we need to properly find arguments to support it in favour of the alternative solutions. That is, we need to be able to reason about selecting a solution for our problem. Considering the fact that we constantly need to make design decisions, the process of reasoning must be constantly present in our development activities.

Thus, at higher-level, we should constantly employ reasoning.

'for' loops and reasoning frameworks

Let's now turn our attention to lower-levels and reason about reasoning code.

In a previous article 'How We (Don't) Reason About Code' [Teodorescu21] we explored a possible meaning of reasoning about code. We first tried to define a somehow formal framework for understanding what are the implications of code; later we discussed a model that also takes into consideration the previous experiences of the programmer.

In the first approach, we defined the *reasoning complexity* (or *reasoning effort*) for a code as being the difference between the post-conditions and the pre-conditions of that code. If our minds were purely mathematical, we would write lemmas for everything that's important to ensure we understand the implications. This *reasoning complexity* is equal to the number of such lemmas we need.

As an example, we tried to compute the associated complexity for using a classic `for` loop (with incrementing variable) and for using a ranged `for` loop. In our example, the classic `for` loop has a complexity of 20, compared to a complexity of 8 for the ranged `for` loop. That is, classic `for` loop is much harder to reason about than a ranged `for` loop.

The result of comparing the reasoning complexities of the two styles of `for` loops may not be that helpful. What I find significant is that we defined a framework for reasoning about code. We can take this framework and apply it to other problems; we can expand our reasoning capabilities.

In the second part of the article, we introduced another way of reasoning about code, one that is less formal and one that accepts the previous experience of the developer in considering complexity. We called this the *inference complexity*.

For this new complexity metric, we cannot calculate the complexity of the code just by looking at it. We have to factor in the experiences of its readers, which is very difficult. But, nevertheless, it still gives us a method of reasoning about code. And, especially if we put it near the previous complexity metric, it can tell us about the biases that we, the programmers, have.

2 At least the first sentence, that functions should be small. The second sentence, "smaller than that", cannot be taken seriously. It tells us that any reasonable definition of "small" that we find is not good enough; it takes away from us the possibility of defining a reasonable threshold for the size of a function.

a programming paradigm imposes a set of constraints on the possible programs that can be written, and by doing this it can provide some guarantees about the programs written in that paradigm

We can have reasoning frameworks that are more formal, and we can have reasoning frameworks that are less formal. And, it appears that both of them are useful.

Programming paradigms and guarantees

Reasoning is often hard. To make the reasoning process simpler, we typically perform it in limited bounds. That is, we are drawing some bounds in which we reason, and start from assumptions. If we were to start reasoning without any bounds, we would have to start from metaphysics in order to analyse the merits of a design choice.

Programming paradigms are often useful bounds for the process of reasoning about code. They impose certain restrictions about what can be done in a program that respects that paradigm. Let's take a couple of examples.

In structured programming, a function is an abstraction. Once we understand that abstraction, we can use the same reasoning each time the function invocation is seen. The meaning of the function doesn't change depending on the context in which it is being used. If the function can mutate just one single global variable, we don't expect it to mutate other global variables in any of the invocations.

In functional programming, all the functions are pure. If we look at a function call, we know that the function doesn't have any side effects, and is idempotent. If we call the function twice, it will have the same effect on the correctness of the program as if we would call it once. Furthermore, the output of the function is only dependent on the arguments of the function. This reduces the amount of reasoning we need to perform. If we see the same function called twice, we don't have to reason about it twice. Moreover, two functions that are not chained together cannot interfere with each-other; we can reason about them in isolation.

In my previous article, 'Value-Oriented Programming' [Teodorescu23], we discussed a new programming paradigm which focuses on value semantic and removing references as first-class entities in a language. If the programs conform to this paradigm, then the user doesn't have to reason about safety issues (they cannot appear), and all the reasoning can be done locally (no spooky action at a distance).

As discussed in this article, a programming paradigm imposes a set of constraints on the possible programs that can be written, and by doing this it can provide some guarantees about the programs written in that paradigm. These guarantees can simplify our reasoning about the code.

Adding restrictions to a language can simplify it by reducing the number of things we need to reason about. On the other hand, it may make some problems harder to solve; this can increase the amount of reasoning we need to perform. It's always a compromise.

If we generalise the notion of programming paradigm, the main takeaway of this section should be that, whenever we have to reason about a software problem, we should find the right frame that would be sufficient to contain the problem, and yet provide enough guarantees to simplify our reasoning process.

What can we reason about?

This one is easy: probably every aspect of software engineering. To make it more useful for the reader, let's provide a few examples:

- **Code**: What's the best way to reason about code? How can we easily find out the implications of a code snippet? What's the right organisation of code? Are there any paradigms that we can set to make the reasoning easier? Etc.
- **Software design**: How should we design the software? When does a top-down approach works best, and when a bottom-up works? What are the fundamental units of design? How can we measure the quality of a design? What does it take to understand a design? What's the easiest way to document a design? Etc.
- **Software architecture**: What is architectural and what is not? How do we make architecture coherent? How much time should we spend in defining the architecture upfront? How can we best document architecture? How to evolve the architecture? Etc.
- **Processes**: How many processes does a software organisation need to have? Does different processes work better than others? How much are the processes dependent on the organisation? How much can we influence the productivity of the software organisation with processes? How to tell when processes help and when they are just bureaucracy?
- **Testing**: How much testing does a software need? Do the testing needs differ (significantly) depending on the type of software? Are there any testing methods that are better than others? How to structure our tests?
- **Complexity**: How can we measure the complexity of a problem? What is essential and what is accidental? How should we reason about complexity? Etc.
- **Reasoning**: What's the extent of our reasoning, and how can we ensure that whatever we do has any connection with the reality? How can we move from pure reasoning (prone to errors) into empirical data (better matching reality)? Etc.

This article (this part and the one appearing in the next number) approaches the last two items on our list.

A critique of pure reason

The reader may infer from the above text that I'm strongly arguing to drop any empirical arguments in software engineering and start using *pure* reasoning instead. This is far from my intention.

We should use this reasoning only in limited scenarios:

- to make sense of things when we don't have empirical data, or making sense of incomplete empirical data
- to form hypothesis/models that can later be tested empirically.

Reasoning on accidental complexity is much harder, as most of the things we do in software engineering end up as accidental complexity

A great example of reasoning in the first category is Brooks' reasoning on essential and accidental complexity. We don't have empirical data to make the distinction between essential and accidental complexity; after all, what is essential complexity?

The second category is far more interesting. Let's take, for example, the *Lean Software* principles. They may have originated in practice at Toyota, but they originated in a different domain. In software, they first appeared as a conceptual framework [Poppendieck03]. It took us some time to validate this principle empirically and prove that they can lead to successful software organisations; the main results can be found in the *Accelerate* book [Forsgren18].

The book provides strong empirical data showing that certain software development practices lead to success. This is one of the few studies that we have in our field that can show empirical that certain practices are more valuable than others.

This should also be our goal with *pure* reasoning in software engineering: find models that we can later prove empirically.

Interlude (part 1)

We reached the end of the first part of our two-part article. It's time to have some partial conclusions.

The goal of the entire article was to reason about complexity. In this first part, we discussed the importance of reasoning in software engineering so that we can later apply this reasoning about complexity in the second part.

We started to argue that software engineering doesn't have enough good empirical studies to capture best practices in the field. Instead, they are replaced by some kind of reasoning. We argued then that some of the most important foundational work in software engineering draw their inspiration from philosophy. It seems that philosophical reasoning is essential to our field.

We then explored a few ways in which we can reason in software engineering. Exploring this landscape gives us an idea of what strategies we can employ when reasoning about software engineering topics. The reasoning that was used to analyse the two types of `for` loops is especially important for this article, as we would use the same pattern when reasoning about complexity.

Reasoning is good, but reasoning should not be divorced from practice. Whenever it makes sense, we should test empirically the results of our reasoning.

In the next part, we will look at complexity. We would start by defining the problem: how we are introduced to essential complexity and accidental complexity, and a possible sharp distinction between the two (essential belongs to the problem, while accidental belongs to the solution). With this sharp distinction, we go on exploring essential complexity. We define a framework to associate a number with the essential complexity; this allows us to compare the complexity of two problems. Reasoning on accidental complexity is much harder, as most of the things we do in software engineering end up as accidental complexity. This is why the first part was needed to start exploring the extent of reasoning. At the end, we try to provide another answer to Brooks' old question: can we find a ten-fold improvement in productivity for software engineering?

Read the next *Overload* issue for the continuation. ■

References

[Aristotle-1] Aristotle, 'Metaphysics' in *The Complete Works of Aristotle, Volume 2: The Revised Oxford Translation*, edited by Jonathan Barnes, Princeton University Press, 1984.

[Aristotle-2] Aristotle, 'Prior Analytics' in *The Complete Works of Aristotle, Volume 2: The Revised Oxford Translation*, edited by Jonathan Barnes, Princeton University Press, 1984

[Farley21] Dave Farley, *Modern Software Engineering: Doing What Works to Build Better Software Faster*, Addison-Wesley Professional, 2021.

[Forsgren18] Nicole Forsgren, Jez Humble, Gene Kim (2018) *Accelerate: Building and Scaling High Performing Technology Organizations*, IT Revolution.

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

[Martin08] Robert C. Martin (2008) *Clean Code: A Handbook of Agile Software Craftsmanship*, Pearson.

[Poppendieck03] Mary Poppendieck, Tom Poppendieck (2003) *Lean Software Development: An Agile Toolkit*, Addison-Wesley Professional.

[Seemann21] Mark Seemann (2011) *Code That Fits in Your Head : Heuristics for Software Engineering*, Pearson.

[Stepanov09] Alexander A. Stepanov, Paul McJones (2009) *Elements of programming*, Addison-Wesley Professional.

[Teodorescu21] Lucian Radu Teodorescu, 'How We (Don't) Reason About Code', *Overload* 163, June 2021, https://accu.org/journals/overload/29/163/overload163.pdf#page=13

[Teodorescu23] Lucian Radu Teodorescu, 'Value-Oriented Programming', *Overload* 173, February 2023, https://accu.org/journals/overload/31/173/overload173.pdf#page=16

[Wayne21a] Hillel Wayne, 'Are we really engineers?', 2021, https://www.hillelwayne.com/post/are-we-really-engineers/

[Wayne21b] Hillel Wayne (2021) 'We are not special', https://www.hillelwayne.com/post/we-are-not-special/

[Wayne21c] Hillel Wayne (2021) 'What engineering can teach (and learn from) us', https://www.hillelwayne.com/post/we-are-not-special/

[Wayne19] Hillel Wayne, 'Intro to Empirical Software Engineering: What We Know We Don't Know', *GOTO 2019*, https://www.youtube.com/watch?v=WELBnE33dpY.

Incompatible Language Features in C#

Adding features to an established language can introduce sources of errors. Steve Love examines some of the pitfalls of combining positional record structs with automatic property initializers.

The C# language has undergone quite significant changes over the last three years or so. No longer tied to the (relatively infrequent) release cadence of Microsoft's Visual Studio, the C# compiler and .NET platform designers have added a host of new features, as well as tidied up some incongruities and removed some restrictions since C# v8.0 in 2019. Overall those changes make C# a more consistent language, with fewer special corner cases, and therefore easier to write and to learn, but some changes have also introduced new complexities of their own.

C# v10.0 – released with .NET 6 in 2021 – introduced two new features that were somewhat subdued in their respective announcements[1]: record structs, and automatic property initializers for value types. We'll get to automatic property initializers but first, let's have a look at why record structs were introduced.

Why record structs?

To understand record structs, you need to understand records[2]. C# v9.0 and .NET 5 added a new way of creating user-defined types: the record. Before the introduction of records, we could choose between classes and structs. A class defines a *reference type*, meaning instances live on the heap and benefit from garbage collection. Copying a reference type variable creates a new reference to the same instance on the heap. A struct defines a value type, meaning the lifetime of an instance is tied (broadly speaking) to the scope of the variable associated with it. Copying a value type variable copies the entire instance – value type variables and instances have a 1-to-1 relationship – and this has important consequences for efficiency and equality semantics.

The default behavior of `Equals` for classes performs a reference-based comparison where two variables compare equal if they refer to the same object on the heap. By contrast, `Equals` for structs performs a value-based comparison whereby two variables are equal if all their fields and properties match. Value-based equality comparisons are common for some kinds of types in many programs, but copying large struct instances – those with several fields, for example – could negatively impact a program's performance. We can override the default equality comparison for class types to perform a value-based rather than reference-based comparison, and still benefit from the reference-based copying behaviour for instances of the type.

A record is a reference type (in fact, once compiled it really is a class), and the compiler synthesizes an efficient and correct implementation of equality (along with a few other features), which represents a fairly significant saving on some boilerplate code that's surprisingly easy to get wrong. Put simply, records are reference types with *compiler generated* value-based equality behaviour.

Record structs are merely the value type equivalent of records. The compiler translates a record struct into a struct. While structs have always had value-based equality semantics, the default behaviour suffers from performance issues; specifically, the default implementation of `Equals` usually (with some exceptions) requires the use of reflection, which is not generally associated with high performance. As a result, customizing `Equals` for structs is good practice, but as with classes, there are pitfalls to avoid. The compiler generates the implementation of `Equals` for a record struct in the same way as for a record.

On the face of it, record structs were introduced simply to re-establish the symmetry between reference types and value types, but there are good reasons for choosing value types over reference types in some circumstances. In particular, structs and record structs are a good choice where instances are short-lived and an application will have large numbers of them, because value types aren't subject to garbage collection. Implementing them as reference types instead might add considerable heap memory pressure, causing extra work for the garbage collector.

Property initializers

Using *positional* syntax[3] with either records or record structs makes defining simple types compact and convenient. Here's an example of a positional record struct to represent a UK postal address:

```
public readonly record struct
  Address(string House, string PostCode);
```

The compiler translates this positional syntax to a struct with a read-only property for each positional argument (owing to the use of `readonly` in the type's declaration), and a constructor taking those parameters to initialize the properties. Since a record struct is really just a normal struct when it's compiled, a default-initialized instance will have `null` for any reference type properties. That means both properties of a default-initialized `Address` will be `null`. Classes have been able to use automatic property initializers since C# v6.0 to address problems like this by allowing automatic properties to be given a default value (the same is true for fields too, but we're only considering properties here). From C# v10.0, automatic property initialization syntax is also permitted for record structs and normal structs, shown here for the `Address` type:

```
public readonly record struct
  Address(string House, string PostCode)
{
  public string House { get; } = "";
  public string PostCode { get; } = "";
}
```

Here we define our own `House` and `PostCode` properties (inhibiting the compiler from generating them from the `Address` type's positional parameters) and use the property initializer to assign an empty string as the *default* value for each property. The intention of using the property initializers is to try to prevent `null` values for those properties when an `Address` is default-initialized, like Listing 1 (overleaf).

1 https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-10

2 https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#record-types

3 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#positional-syntax-for-property-definition

STEVE LOVE

Steve Love is a programmer who gets frustrated at having to do things twice. He can be contacted at steve@arventech.com

record structs were introduced simply to re-establish the symmetry between reference types and value types, but there are good reasons for choosing value types over reference types in some circumstances

```
var defaultAddress = new Address();

Assert.That(defaultAddress.House, Is.Not.Null);
Assert.That(defaultAddress.PostCode,
  Is.Not.Null);
```

Listing 1

The property initializers in the `Address` type are valid since C# v10.0, but unfortunately, this test doesn't pass.

Initialization order

The problem here is that positional parameters and property initializers don't mix. Property initializers are part of object construction, so the initializers are only applied when we call a constructor. In the example, the `defaultAddress` variable is default-initialized, meaning that no constructor call occurs, and thus the property initializers are never applied.

Since the compiler uses the positional parameters to generate a constructor for our record struct, if we use that constructor to create an object, the property initializers are indeed applied:

```
var address = new Address(House: "221b",
  PostCode: "NW1 6XE");

Assert.That(address.House, Is.Not.Null);
Assert.That(address.PostCode, Is.Not.Null);
```

The named arguments used to create the `address` variable aren't mandatory, but they emphasize how the arguments are applied to the positional parameters (or rather, the constructor parameters created by the compiler). This test passes, but hides a deeper problem: the property initializers have been applied to the properties, but the arguments we passed to the constructor have not! Neither of these tests pass:

```
Assert.That(address.House, Is.EqualTo("221b"));
Assert.That(address.PostCode,
  Is.EqualTo("NW1 6XE"));
```

Both properties now have the values assigned by the automatic property initializers, and so are both empty strings.

The compiler-generated constructor hasn't initialized the properties from its parameter values. Note that this behaviour applies equally to record types. The earlier problem with default initialization doesn't apply to records, which as reference types have a *default constructor* inserted by the compiler if no other constructors are defined. Since the compiler uses the positional parameters to create a constructor (called the *primary constructor*), the default constructor is inhibited, with the result that creating a new object without arguments would fail to compile.

For both records and record structs, however, the primary constructor will only use its parameter values to initialize properties generated by the compiler; if we define any property of our own, even if it has the same name as a positional parameter, it is not initialized by the primary constructor.

Did I mention that positional parameters and property initializers don't mix?

Requiring properties to be initialized

Our original problem was that the default values for the `string` properties of `Address` would be `null` in a default-initialized instance. There are a couple of ways to address this – at least for most common cases – but no perfect solutions.

Since C# v11.0 we can force the user to assign a value to a property by using the `required` keyword to modify the property definition, like this:

```
public readonly record struct Address
{
  public required string House { get; init; }
  public required string PostCode { get; init; }
}
```

Note that we've added an `init` accessor[4] for both properties, enabling *object initialization* for `Address` objects. The `init` accessor was introduced in C# v9.0 along with records. The compiler will reject the use of `required` without either a public `init` or `set` accessor, and `init` means an `Address` is immutable once it's been created.

This doesn't prevent the user from assigning `null` (although we could use the nullable reference type[5] feature available since C# v8.0 to warn them), but we no longer need property initializers. The tests in Listing 2 all pass.

Note that we're no longer using positional syntax for `Address`. We might have used a plain struct here, although using a record struct brings other benefits, but the `required` keyword means `Address` objects must be created using object initialization: the primary constructor for a positional record struct won't initialize our custom properties, which is why `Address` doesn't use the positional syntax. That's a little unfortunate,

```
var address = new Address {
  House = "",
  PostCode = ""
};

Assert.That(address.House, Is.Not.Null);
Assert.That(address.PostCode, Is.Not.Null);

address = new Address {
  House = "221b",
  PostCode = "NW1 6XE"
};

Assert.That(address.House, Is.EqualTo("221b"));
Assert.That(address.PostCode,
  Is.EqualTo("NW1 6XE"));
```

Listing 2

4 https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#init-only-setters

5 https://learn.microsoft.com/en-us/dotnet/csharp/nullable-references

```
var defaultAddress = new Address();

Assert.That(defaultAddress.House, Is.Not.Null);
Assert.That(defaultAddress.PostCode,
  Is.Not.Null);

var address = new Address(House: "221b",
  PostCode: "NW1 6XE");

Assert.That(address.House, Is.EqualTo("221b"));
Assert.That(address.PostCode,
  Is.EqualTo("NW1 6XE"));
```

Listing 3

because positional record types are very convenient for the most simple types. Luckily, we can revert to a positional record struct, and even make **Address** slightly simpler, while keeping the same guarantees we've realized by using the **required** modifier.

The syntaxy solution

Prior to C# v10.0, defining a parameterless constructor for a value type wasn't allowed. Constructing an instance of a struct type without arguments *always* used the built-in default-initialization, which *always* sets its fields (including property backing fields) to a pattern of all-zero bits – essentially, either **0** or **null**, depending on the field's type.

Since C# v10.0, user-defined parameterless constructors[6] for either structs or record structs are allowed, and we can use this facility to achieve the outcomes needed here: an instance created using **new** but with no arguments has non-**null** values for the properties, while keeping the convenience of the positional syntax to properly initialize properties with those values.

We don't need property initializers, and our record struct representation of **Address** actually becomes a little simpler:

```
public readonly record struct
  Address(string House, string PostCode)
{
  public Address() : this("", "")
  {
  }
}
```

Here we're defining our own parameterless constructor which uses *constructor forwarding* to invoke the compiler-generated primary constructor with the default, non-**null**, values as the arguments. The syntax used here is somewhat arcane in that we're forwarding to an *invisible* constructor, but it is arguably less surprising than the alternatives we've already explored. The compiler synthesizes the properties based on the positional parameters, and those properties are correctly initialized by the primary constructor which is directly invoked by our parameterless constructor with the required default values for those properties. The tests in Listing 3 all pass.

We still can't prevent a default-initialized[7] **Address**, such as **default(Address)**, or the elements of an array of **Address** objects, which will both still have **null** for their properties; such instances will *always* be default-initialized, and it's not possible to change or prevent that behaviour.

6 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct#struct-initialization-and-default-values

7 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/default

More on required properties

Disallowing automatic property initializers for structs (record structs were added at the same time this restriction was lifted) was a frequent source of friction, so removing the restriction is beneficial, but needs to be used with care. We've not explored all the complexities here but the take-away is that mixing positional record types and automatic property initializers will give you a headache. You have been warned!

The **required** keyword in C# v11.0 certainly has its uses, but it doesn't play well with constructors. Consider this record:

```
public sealed record class Address
{
  public Address(string house, string postcode)
    => (House) = (house);

  public required string House { get; init; }
  public required string PostCode { get; init; }
}

var address = new Address("221b", "NW1 6XE");
```

The creation of the **address** variable here doesn't compile. Because the properties are marked **required**, we *must* set them in an object initializer (using braces **{ }**) or add the **[SetsRequiredMembers]** attribute to the constructor.

Adding the attribute to the constructor satisfies the compiler, but it's not foolproof, as shown in Listing 4.

We must import the **System.Diagnostics.CodeAnalysis** namespace in order to use **[SetsRequiredMembers]**, but even though we've applied that attribute to the constructor here, the compiler still can't catch the fact that the constructor does not initialize all the required properties. This test fails because the **PostCode** property isn't initialized in the constructor.

Closing thoughts

Language design is undoubtedly hard, and adding new features to any non-trivial language can bring unforeseen consequences. C# may consider itself to be "simple", but the truth is that usefulness almost always involves complexity. New features can interact with long-established semantics in … interesting ways. In this article we've examined how just some of the many new features in C# are intricately entwined with each other, and with features that have been part of the C# language from the very beginning. ■

```
using System.Diagnostics.CodeAnalysis;

public sealed record class Address
{
  [SetsRequiredMembers]
  public Address(string house, string postcode)
    => (House) = (house);

  public required string House { get; init; }
  public required string PostCode { get; init; }
}

var address = new Address("221b", "NW1 6XE");

Assert.That(address.PostCode, Is.Not.Null);
```

Listing 4

Need Something Sorted? Sleep On It!

Sorting algorithms have been thoroughly studied. Kevlin Henney takes an unexpected paradigm journey into sleep sort.

A decade ago, I first presented a lightning talk entitled 'Cool Code'. This short talk evolved into a full talk whose iterations I presented over the next half decade. The focus? Code that, for some reason or other, can be considered cool. For example, code that has played a significant role in historical events, such as the source for the Apollo Guidance Computer [Apollo]. Or code that is audacious – if not seemingly impossible – given its constraints, such as David Horne's 1K chess [Frogley01]. There is code that is both simple and profound, such as Peter Norvig's fits-on-a-slide spelling corrector [Norvig16]. And code that demonstrates ingenuity and humour, such as Yusuke Endoh's Qlobe [Endoh10].

Leaving aside its content for a moment, one of the most interesting things about the talk was its stone-soup nature [Wikipedia-1]. Whenever and wherever I gave it, I would usually receive at least one suggestion for more code to include. This drove the talk's evolution over the years. Summed across all its variations, there's probably a half-day's worth of material in total.

The first full-length version of the talk I presented was at JavaZone 2011 [Henney11]. Afterwards, someone came up to me and asked, "Have you come across sleep sort?" I hadn't. "You should look it up. I think you might enjoy it." I did. And I did.

Bourne to sleep

Possibly the only good thing to ever come out of 4chan, sleep sort was created by an anonymous user in 2011 [Sleep]. The original code was written in Bash, but it also runs as a traditional Bourne shell script.

Here is a version with the called function renamed to something more meaningful (it was `f` in the original post):

```
function sleeper() {
  sleep $1
  echo $1
}
while [ -n "$1" ]
do
  sleeper $1 &
  shift
done
wait
```

We could condense this code by inlining the `sleeper` function at its point of use:

```
while [ -n "$1" ]
do
  (sleep $1; echo $1) &
  shift
done
wait
```

I have a slight preference for the shorter version but, in this case, keeping the longer form gives a structure we can preserve more easily across implementation variations. It also offers more real estate for supplementary explanation (see Listing 1).

Assuming this is in a script file named sleepsort, you can run it as follows:

```
function sleeper() {# define a function that...
  sleep $1          # sleeps for duration of its
                    # argument in seconds
  echo $1           # prints its argument to the
                    # console
}
while [ -n "$1" ]   # while the script's lead
                    # argument is not empty
do
  sleeper $1 &      # launch a sleeper with the
                    # lead argument
  shift             # shift argument list,
                    # so $2 becomes $1, etc.
done
wait                # wait for all launched
                    # processes to complete
```

Listing 1

```
sleepsort 3 1 4 1 5 9
```

With the following result:

```
1
1
3
4
5
9
```

The first two lines arrive after 1 second, the next after 3 seconds, the next after 4 seconds, etc.

Bonkers, brilliant and definitely NSFW.

Reductio ad absurdum

Sleep sort presents itself as an O(n) sorting algorithm for non-negative integers. It's also a lot of fun. Sleep sort lets us defamiliarise a hackneyed topic – sorting – and see it anew: sorting as an arrangement in time rather than space. We can learn things from it, from the mechanics of time expressed in code to styles of composition in different paradigms and languages. It lets us explore assumptions as we uncover solution possibilities and brush up against their boundaries.

If you want to extend the sortable domain into negative numbers, either you must add a bias to each value of at least the magnitude of the most negative value, or you need to find yourself a source of reverse entropy (e.g., thiotimoline [Wikipedia-2]). Although sources of entropy are common in modern CPUs, reverse entropy sources will not available in the foreseeable (or unforeseeable) future.

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

Python's global interpreter lock (GIL) and casually modifiable underlying object model make it almost uniquely unsuited to pre-emptive threading

In theory, this algorithm can work for floating-point numbers… in theory. In practice, scheduler limitations and variability and the cost of process creation makes sorting of floats unreliable the closer the values are to one another.

Which brings us to performance. Sleep sort is not just fun; it's educational.

That the algorithm appears to take linear time, i.e., the number of steps taken to sort is proportional to the number of items in the input, is both a point of interest and a source of distraction. Big O notation can be considered the GDP of performance. It is a singular observation that, while not necessarily false, may distract you from other measures that matter. There's more to algorithms and their (un)suitability than just finding the abstracted proportionality of steps to input size.

At first sight, the sorting of each value appears to be done without reference to or co-ordination through any shared state – not even the other values to be sorted. Viewed classically, the only shared resource is the standard output. But look more closely and it becomes clear that there is co-ordination and it is through a shared resource: time. Without the fork–join arrangement (from each `&` to the `wait`), this code would terminate before completion of the algorithm and, without the explicit and managed passage of time, there is no algorithm. The `wait` is a necessary feature not an optional one; for the algorithm, time is of the essence, not simply a consequence.

The specific machinery of time is not a necessary part of a sleep sort implementation, but that time has a design is, which opens up other implementation possibilities.

In a state of threadiness

We can switch from processes to threads, preserving the structure of the solution above. In C++, the `sleeper` function becomes

```
void sleeper(int value)
{
  this_thread::sleep_for(value * 1s);
  cout << value << "\n";
}
```

The move from pre-emptive multitasking processes to pre-emptive multitasking threads is, in this case, a simple one. The `sleeper` function is still a function that receives an argument and is launched from within a loop. The separateness or sharing of memory doesn't play a role in the solution. Had the process-based Bash solution used an IPC mechanism for synchronisation, such as pipes or locks, this would have had to have been translated in some way. But in the absence of any other communication, the thread-based version is essentially a transliteration of the process-based one.

This leaves the significant differences between the shell and the C++ versions as being down to language, such as syntax and typing, and the realisation of the fork–join model.

To make it a little easier on the eye, I've assumed the `std` namespace is made available, along with the `std::chrono_literals` namespace, which lets us express 1 second directly as `1s`, which would also be the value we would change to rescale the unit interval.

```
void sleepsort(const auto & values)
{
  vector<jthread> sleepers;
  for (int value: values)
      sleepers.push_back(jthread(sleeper,
                                 value));
}
```

The function receives the values to be sorted by reference, along with a `const` promise that the values will not be altered by the sorting. Type deduction via `auto` leaves the remaining type bookkeeping to the compiler, so it can be called with a `vector<int>`:

```
sleepsort(vector {3, 1, 4, 1, 5, 9});
```

Or with anything else that satisfies the expectation of the `for` loop:

```
sleepsort(list {3, 1, 4, 1, 5, 9});
```

And, for C++ compilers that also support C compound literals:

```
sleepsort((int[]) {3, 1, 4, 1, 5, 9});
```

The lifetime of `sleepers` is scope-bound to `sleepsort`. When the function completes, `sleepers` will be cleaned up automatically and, in turn, will clean up its contained values. In this case, the contained values are `jthread` instances, which are threads that join – i.e., wait for – their threads to complete before completing their own destruction. Thus, joining is an automatic end-of-life feature of the `sleepers` variable, and the `sleepsort` function will not return until all the launched threads have completed.

Fearless symmetry

Python's global interpreter lock (GIL) and casually modifiable underlying object model make it almost uniquely unsuited to pre-emptive threading. That does not, however, mean that it cannot express concurrency conveniently. Originally with generator functions, and more recently and explicitly with async functions, Python supports coroutines. Donald Knuth states:

> Subroutines are special cases of more general program components, called coroutines. In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines. [Knuth05]

Coroutines were invented in the late 1950s, with the word itself coined in 1958 by Melvin Conway [Conway] (yes, that Conway) . They were an influential feature across many architectures, paradigms and languages. Their popularity waned to the point of disappearance in the 1980s, although co-operative multitasking became a common feature of runtime environments (e.g., Mac OS and Windows). Over the last decade, this classic of the procedural paradigm has enjoyed new popularity, with many languages and libraries adopting coroutines or coroutine-like constructs.

> The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously. [Conway63]

Coroutines offer a constrained concurrency model, one based on single-threaded execution of concurrently available re-entrant code, rather than truly concurrent execution. For sleep sort that is enough.

```
async def sleepsort(values):
  async def sleeper(value):
      await sleep(value)
      print(value)
  await wait(
    [sleeper(value) for value in values])
```

Structurally, there is little difference between this and the pre-emptive solutions, with all of them sharing the same underlying fork–sleep–join anatomy.

The **sleepsort** function expects **value**s to be iterable, such as a list or a tuple. The values can be non-negative floats, not just integers, for which the underlying single-threaded nature of coroutines offers a better ordering guarantee than would be delivered for the equivalent pre-emptive execution.

The **sleeper** function is nested as a private detail within the **sleepsort** function. The **sleeper**-launching loop is expressed as a comprehension whose resulting list holds the awaitable coroutine instances that are run and blocked on by **asyncio.wait**, thereby collapsing the whole fork–join into a single **await** statement.

An alternative expression of this is to gather together all of the tasks that must be run concurrently, in this case unpacking the list of coroutines to **asyncio.gather**, which expects each awaitable object to be a separate argument:

```
async def sleepsort(values):
  async def sleeper(value):
      await sleep(value)
      print(value)
  await gather(
    *[sleeper(value) for value in values])
```

In either case, the **sleepsort** coroutine must be launched explicitly as a coroutine:

```
run(sleepsort([3, 1, 4, 1, 5, 9])
```

The coroutine-ness, and therefore the **run**, can be further wrapped inside an ordinary function, if you prefer.

It's about time

If we look closely at what we've been trying to do, we'll see we've been working around the intrinsic structure of the solution rather than expressing it directly. Consider, for a moment, what the essence of sleep sort is: time-based execution. What have the three solutions shown so far done? They have launched separate paths of execution that have immediately been suspended so as to prevent that very execution. The terms and conditions of suspension have been time-based, but the constructs that shape the code have not been. We've been faking timers.

In the words of Morpheus [YouTube], "Stop trying to hit me and hit me."

Time can be considered a source of events, as can I/O and user interaction. Event-driven control flow offers an alternative way of structuring applications to the the explicit top–down control flow most commonly used to express algorithms. This inversion of control activates code in response to events rather than embedding blocks or polls into the code to stem the flow of control. Such opposite framing has many design consequences.

Although they both express themselves through asynchrony, threads and event-driven code mix poorly. It is generally better to choose one approach and stick with it. Bringing these two world views together (correctly) under the same architectural roof demands a clear head, a clear separation of responsibilities and, unfortunately, a clear increase in complexity. For example, the POSA2 [Buschmann00] full write-up of the Reactor pattern, which is an event-loop based dispatch pattern, is 36 pages long; the Proactor pattern, which mixes in asynchronous mechanisms for its event-handling, took 46 pages. Although we managed to condense them to three pages each in POSA4 [Buschmann07], I recall that for much of the draft Proactor was hitting the four-page mark. We also ranked the patterns differently: Reactor was considered more mature and less tricky than Proactor.

Tempting as it is to mess around with timer events in GUI frameworks or with POSIX timers in C for an illustrative example, life is too short. The corresponding JavaScript/HTML5 implementation, however, is not:

```
const sleepsort = values =>
  values.forEach(
    value => setTimeout(
      () => document.writeln(value),
         value * 1000))
```

To run, this needs to be embedded into HTML as a **<script>** – timers are part of the HTML5 standard but not the ECMAScript standard.

```
sleepsort([3, 1, 4, 1, 5, 9])
```

The **sleepsort** function can be refactored to more closely resemble the separations in the previous three versions:

```
const sleeper = value =>
  setTimeout(() =>
    document.writeln(value), value * 1000)
const sleepsort = values =>
  values.forEach(sleeper)
```

The **sleepsort** function still iterates through each value to be sorted, but rather than launching a **sleeper**, it simply calls it as an ordinary function that then registers a lambda expression as a callback after the appropriately scaled number of milliseconds.

In this case, no join boundary is necessary to wait on all the timers expiring because timers and their effect persist in their enclosing web-page environment.

Trigger warning

Time-triggered systems are a particular class of event-driven architecture that offer another variation on the idea of organising application execution in terms of tasks and time.

Rather than regarding timers as plural and first class, the progress of time is uniquely marked out by a single recurring interrupt. This heartbeat is played against a schedule of tasks, a queue of work or a set of pending notifications from external or internal events. The tick of the timer marks out time slices into which tasks are placed rather than associating tasks with timers, threads or other directly asynchronous constructs. Therefore, in contrast to our usual conception of event-driven systems, time-triggered systems have only one event they respond to directly: there is only quantised, periodic time. If there is something to do at that moment, it is done. The constraint that must be respected to make this architecture work is that tasks undertaken in any time slice must fit within that time slice – there is no concurrent execution or arrhythmia.

Time-triggered systems have a more even and predictable behaviour that makes them attractive in safety-critical environments. They impose a regular and sequential structure on time that ensures conflicting events are neither lost nor experience priority inversion in their handling.

> It is the need to deal with the simultaneous occurrence of more than one event that both adds to the system complexity and reduces the ability to predict the behaviour of an event-triggered system under all circumstances. By contrast, in a time-triggered embedded application, the designer is able to ensure that only single events must be handled at a time, in a carefully controlled sequence. [Pont01]

Although normally associated with C and embedded systems, we can illustrate the approach using JavaScript and HTML5's **setInterval** function (see Listing 2, overleaf).

```
const sleepsort = values => {
  const sleepers = {}
  values.forEach(value => sleepers[value]
    = sleepers[value] + 1 || 1)
  let ticks = 0
  const tick = () => {
    for (let count = sleepers[ticks];
         count > 0; --count)
      document.writeln(ticks)
    ++ticks
  }
  setInterval(tick, 1000)
}
```

Listing 2

I've separated out the roles in this code to be a little more explicit, but it is already clear that the shape of this solution has little in common with the previous four solutions. It is more complex, involving an additional intermediate data structure, **sleepers**, that represents the task table (i.e., number of items of a particular value to be sorted) with respect to the time slice. For the task in hand it is – compared to the other examples – overkill (but keep in mind that we are dealing with sleep sort, so all excess is relative…).

The first tick is counted as 0 and there is an initial offset of 1000 milliseconds before tick 0 drops. The interval timer will run forever (well, until the page is closed), but by decrementing an initial sleeper count the code could be tweaked to cancel the timer when all the sleepers have expired.

We have journeyed through time to a solution that has laid out the sorting structure spatially in a table. The solution is now, in truth, data driven and is simply dressed up to behave like sleep sort. Time is now consumed in a more conventional algorithmic form – albeit in a rather elaborate way – and the defining temporal separation of sleep sort has been rotated into spatial separation through data structure.

Timing out

I previously noted [Henney20] that 'Looking at something from a different point of view can reveal a hidden side.' And that is true of this essay, whose subtitle describes *an unexpected journey*.

I originally had it in mind to show three variants of sleep sort: the simplified shell version I normally use in talks, a multithreaded version and then one other. Deciding on coroutines as the third variant made it clear to me that, nice as that troika was for illustrating sleep sort, excluding an event-driven solution might feel like a glaring omission. And once I started down that path, a time-triggered solution seemed like another natural inclusion. Three becomes four becomes five and, before you know it, what started as a simple examination of sleep sort becomes an exploration of execution paradigms and architectural decisions.

Of course, there are many more paradigms and variations that could be explored and lessons learned from this fun example. For example, although the shell version was based on OS processes, there are other process-based approaches such as actors and CSP that could be explored. Writing, however, shares an important attribute with algorithms: there should be a stopping condition. In this case, we'll stop at five.

The purpose here is neither the search for an ideal sorting algorithm nor the ideal way of implementing a non-ideal sort. It is to take something that is familiar and commodified and dull and find within it something fun and unexpected and surprising – sorting without sorting – and use it as a way to pry open different approaches to organising execution with respect to the flow and structure of time. How code and data are organised with respect to time is architectural, and changing your model of time has consequences we can appreciate and apply beyond sleep sort.

Grace Hopper once observed [QuoteInvestigator]:

> Humans are allergic to change. They love to say, 'We've always done it this way.' I try to fight that. That's why I have a clock on my wall that runs counter-clockwise.

Grace Hopper is why I have a clock on my wall that does the same.

The value of the exercise comes from approaching sorting, concurrency and events through ostranenie [PennState] rather than habit, as something unfamiliar and new.

> Defamiliarization or ostranenie is the artistic technique of presenting to audiences common things in an unfamiliar or strange way so they could gain new perspectives and see the world differently. [Wikipedia-3]

Whether a difference in perspective will shower you with IQ points [Wikiquote] is an open question, but it can grant you greater knowledge and insight and both satisfy and fuel a curiosity you may not have realised you had. ■

References

[Apollo] Apollo 11 Source Code on Google Code: http://lambda-the-ultimate.org/node/3522

[Buschmann00] Frank Buschmann, Hans Rohnert, Douglas C. Schmidt and Michael Stal (2000) *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, published by Wiley & Sons

[Buschmann07] Frank Buschmann, Kevlin Henney and Douglas C. Schmidt (2007) *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, published by Wiley & Sons

[Conway] Conway's Law: https://www.melconway.com/Home/Conways_Law.html

[Conway63] Melvin E. Conway (1963) 'Design of a Separable Transition-Diagram Compiler', published in Communications of the ACM, Volume 6:7, July 1963, available at: https://www.melconway.com/Home/pdf/compiler.pdf

[Endoh10] Yusuke Endoh 'The Qlobe', published on 5 September 2010 at http://mamememo.blogspot.com/2010/09/qlobe.html

[Frogley01] Thaddaeus Frogley (2001) 'The greatest program ever written', available at: https://thad.frogley.info/archive/the_greatest_program.html

[Henney11] Kevlin Henney (2011) 'Cool Code', presented at JavaZone, Norway: https://vimeo.com/28772428

[Henney20] Kevlin Henney 'Out of Control', published on 17 December 2020 at https://kevlinhenney.medium.com/out-of-control-97ed6efa2818

[Knuth05] Donald E. Knuth (2005) *The Art of Computer Programming, Volume 1*, published by Addison Wesley Professional

[Norvig16] Peter Norvig 'How to Write a Spelling Corrector', published at https://norvig.com/spell-correct.html from Feb 2007 to August 2016

[PennState] Word of the Week: https://sites.psu.edu/kielarpassionblog2/2016/02/04/ostranenie/

[Pont01] Michael Pont (2001) *Patterns for Time-Triggered Embedded Systems*, published by Addison Wesley

[QuoteInvestigator] Most Dangerous Phrase: We've Always Done It That Way: https://quoteinvestigator.com/2014/11/27/always-done/

[Sleep] Genius sorting algorithm: Sleep sort: https://devrant.com/rants/445614/found-on-my-universitys-computing-soc-page

[Wikipedia-1] Stone Soup: https://en.wikipedia.org/wiki/Stone_Soup

[Wikipedia-2] Thiotimoline: https://en.wikipedia.org/wiki/Thiotimoline

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

[Wikiquote] Alan Kay: https://en.wikiquote.org/wiki/Alan_Kay

[YouTube] Extract from *The Matrix*: https://www.youtube.com/watch?v=5mdy8bFiyzY

This article was published on Kevlin Henney's blog in May 2021, and is available at https://kevlinhenney.medium.com/need-something-sorted-sleep-on-it-11fdf8453914

Type Safe C++ enum Extensions

Is it possible to extend a value type in C++? Alf Steinbach describes how to extend enum values.

Consider if an **enum** like the following,

```
enum class Suit{
  spades, hearts, diamonds, clubs };
```

could be *extended* like

```
enum class Suit_with_joker extends Suit {
  joker };
```

where

- **Suit_with_joker** has all the enumerators of **Suit** plus the **joker** enumerator; and
- enumerators introduced in **Suit_with_joker** get integer values following those of **Suit**; and
- any **Suit** value *is* also a **Suit_with_joker** value.

This would be an example of what I'll call a **value type extension**.

The apparently backwards *is-a* relationship in the last point, where any value of the original type *is-a* value of the derived type, is characteristic of value type extensions.

C++20 totally lacks support for value type extensions, of **enum** types or other types.

```
struct Suit
{
  int value;

  constexpr explicit Suit( const int v )
    : value( v ) {}

    static const Suit spades;
    static const Suit hearts;
};
inline constexpr Suit Suit::spades = Suit( 0 );
inline constexpr Suit Suit::hearts = Suit( 1 );

struct Suit_with_joker: Suit
{
  constexpr explicit
    Suit_with_joker( const int v ): Suit( v ) {}
  static const Suit_with_joker  joker;
};

inline constexpr Suit_with_joker
  Suit_with_joker::joker = Suit_with_joker( 4 );

auto main() -> int
{
  (void) Suit_with_joker::hearts;
      // OK, has inherited the "enumerators".
  Suit_with_joker s1  = Suit::hearts;
      //! C. error, wrong way is-a relationship.
  Suit  s2  = Suit_with_joker::joker;
      //! No c. error, but should be error.
}
```

Listing 1

Value type 'is-a' versus class inheritance 'is-a'

Direct use of class inheritance to model an **enum** extension would give an *is-a* relationship the wrong way.

As a concrete example, see Listing 1.

So, class inheritance works for picking up the base type enumerators, but it doesn't work for expressing the backwards *is-a* relationship between base value type and extended type.

A type safe model of an enum extension

Instead of providing **reference conversion** via class inheritance, a model of an **enum** extension requires **value conversion** via constructors and/or type conversion operators.

This is how e.g. **unique_ptr** works. A **unique_ptr<Derived>&** reference is *not* a **unique_ptr<Base>&** reference – there's no inheritance relationship! But a **unique_ptr<Derived>** value *converts* to a **unique_ptr<Base>** value.

When **Suit_with_joker** doesn't inherit **Suit** (since that would be the wrong way) it must inherit in the **Suit** enumerators from somewhere else. Which means that the enumerators must be defined in parallel enumerator holder classes. With no support for comparisons, data hiding etc., just implementing type safe conversion, it can go like Listing 2 (overleaf).

Compared to the hypothetical

```
enum class Suit{
  spades, hearts, diamonds, clubs };
enum class Suit_with_joker extends Suit {
  joker };
```

… this is a heck of a lot of code; language support would have been nice.

Note: the above code just exemplifies working C++ that implements a type safe enumeration type extension. It does not provide conversion from enumerator to **int**, or more generally to the underlying type. And it does not provide a way to specify the underlying type. As mentioned, it does not provide value comparison.

Suit_names and **Suit_with_joker_names** should be non-instantiable. And **Suit** and **Suit_with_joker** should ideally inherit in the **value** data member from some generic **Enumeration** class. And there are even more issues, all omitted for clarity, but all mostly trivial.

About an enum extension syntax

In the example I used the word **extends** instead of just a colon **:** as with classes, because this isn't like a class inheritance: the *is-a* relationship goes the opposite way.

Alf Steinbach is a Norwegian C++ enthusiast, currently co-admin of FB groups 'C++ Enthusiasts' and 'C++ in-practice questions (most anything!)'. He's worked as a vocational school teacher (no C++), as a college lecturer (teaching also C++, and introducing a Windows programming course), and as an IT consultant (mostly C and C++). He can be reached at alf.p.steinbach@gmail.com

And I imagine that a useful syntax would have to provide for a list of base **enum** types, not just one.

Case in point: in my own hobbyist code I've only used the above scheme once, mostly as an exploration of the issues, and then for *data stream id's* hypothetically defined like

```
enum class Input_stream_id{ in = 0 };
enum class Output_stream_id{ out = 1, err = 2 };
enum class Stream_id extends Input_stream_id,
  Output_stream_id {};
```

This supported type safety for functions taking a stream id, since a stream id argument can be limited to input (**Input_stream_id** parameter type) or output (**Output_stream_id** parameter type), and alternatively can be allowed to be any stream (**Stream_id** parameter type).

Probably the Boost Preprocessing Library (BPL) can be used to generate modeling code for enumeration type extensions. I.e. the hypothetical **enum** declarations can be replaced with actual C++ macro invocations. And possibly, as a less maintenance-friendly alternative, AI based code generation such as via ChatGPT can be used on a case by case base.

However, regarding BPL-based macros, in my experience 'smart' variadic macros lead to brittle and ungrokable code. If or when one chooses to use type safe enumeration extensions, it is perhaps better to just code it up manually, as I did for the stream id's. I believe that this is an example of a feature that would be used if it was provided by the core language, where the centralized effort provides correctness guarantees and the effort involved confers some advantage to all millions of C++ users. ■

```
struct Suit;
struct Suit_names
{
  static const Suit spades;
  static const Suit hearts;
};
struct Suit:
  Suit_names
{
  int value;
  constexpr explicit Suit( const int v )
    : value( v ) {}
};
constexpr Suit Suit_names::spades   = Suit( 0 );
constexpr Suit Suit_names::hearts   = Suit( 1 );

struct Suit_with_joker;
struct Suit_with_joker_names:
  Suit_names
{
  static const Suit_with_joker  joker;
};
struct Suit_with_joker:
  Suit_with_joker_names
{
  int value;
  constexpr explicit
    Suit_with_joker( const int v ): value( v ) {}
  constexpr Suit_with_joker( const Suit v )
    : value( v.value ) {}
};
constexpr Suit_with_joker
  Suit_with_joker_names::joker
  = Suit_with_joker( 4 );
auto main() -> int
{
  (void) Suit_with_joker::hearts;
    // OK, has inherited the "enumerators".
  Suit_with_joker s1  = Suit::hearts;
    // OK, right way is-a relationship.
  #ifdef FAIL_PLEASE
    Suit s2  = Suit_with_joker::joker;
      //! C. error, /as it should be/. :)
  #endif
}
```

Listing 2

Cartoon by Idalia Kulik (idalia.ku@hotmail.com), who also designed the ACCU Conference T-shirt and wrote about the designing experience in *CVu* 35.2.

Why You Should Only Rarely Use std::move

std::move can allow the efficient transfer of resources from object to to object. Andreas Fertig reminds us that using std::move inappropriately can make code less efficient.

In this article, I try to tackle a topic that comes up frequently in my classes: move semantics, and when to use `std::move`. I will explain to you why you should suggest `std::move` yourself (in most cases). However, move semantics is way bigger than what this article covers, so don't expect a full guide to the topic.

The example in Listing 1 is the code I used to make my point: don't use `std::move` on temporaries! Plus, in general, trust the compiler and only use `std::move` rarely. For this article, let's focus on the example code.

Here we see a, well, perfectly movable class. I left the assignment operations out. They are not relevant. Aside from the constructor and destructor, we see in ❶ the copy constructor and in ❷ the move constructor. All special members print a message to identify them when they are called.

Further down in `Use`, we see ❸, a temporary object of `S` used to initialize `obj`, also of type `S`. This is the typical situation where move semantics excels over a copy (assuming the class in question has moveable members). The output I expect, and I wanted to show my participants, is:

```
default constructor
move constructor
deconstructor
deconstructor
```

However, the resulting output was:

```
default constructor
deconstructor
```

Performance-wise, the output doesn't look bad, but it doesn't show a move construction. The question is, what is going on here?

Looking More Closely

For a deeper dive, see 'Nothing is better than copy or move' [Orr18].

C++11 introduced 'move semantics' to facilitate transferring the contents of one object to another more efficiently than creating a copy and then erasing the original. This is particularly focused on optimising the performance of temporary objects, such as when passing them into or out of a function call.

However, in all the discussions about copying and moving, it is easy to forget that not creating an object in the first place may be even more efficient. This can be something done by design choice, or an optimisation applied during compilation. For example, introduction of a temporary object by copying can be removed; this is is called 'copy elision' in C++ and has been permitted in the language for many years.

C++17 adds some additional specification around the creation of temporary variables with the phrase 'temporary materialization'.

Rog's presentation looks at some 'worked examples' of how this behaves in practice, and some things to be aware of.

```
class S {
public:
  S() { printf("default constructor\n"); }
  ~S() { printf("deconstructor\n"); }

  // Copy constructor❶
  S(const S&) { printf("copy constructor\n"); }

  // Move constructor❷
  S(S&&) { printf("move constructor\n"); }
};
void Use()
{
  S obj{
    S{} // Creating obj with a temporary of S ❸
  };
}
```

Listing 1

This is the time to apply std::move, right?

At this point, somebody's suggestion was to add `std::move`.

```
void Use()
{
  S obj{
    // Moving the temporary into obj ❸
    std::move(S{})
  };
}
```

This change indeed leads to the desired output:

```
default constructor
move constructor
deconstructor
deconstructor
```

It looks like we just found proof that `std::move` is required all the time. The opposite is the case! `std::move` makes things worse here. To understand why, let's first talk about the C++ standard I used to compile this code.

Wait a moment!

In C++14, the output is what I showed you for both Clang and GCC. Even if we compile with `-O0` that doesn't change a thing. We need `std::move` to see that the move constructor is called. The key here is that the compiler can optimize the temporary away, resulting in only a single default construction. We shouldn't see a move here because the compiler is already able to optimize it away. The best move operation

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

will not help us here. Nothing is better than eliding a certain step. Eliding is the keyword here. To see what is going on, we need to use the `-fno-elide-constructors` flag, which Clang and GCC support.

Now the output changes. Running the initial code, without `std::move`, in C++14 mode shows the expected output:

```
default constructor
move constructor
deconstructor
deconstructor
```

If we now switch to C++17 as the standard, the output is once again:

```
default constructor
deconstructor
```

Due to the mandatory copy elision in C++17, the compiler must elide this nonsense construction even with `-fno-elide-constructors`.

However, if we apply `std::move` to the temporary copy, elision doesn't apply anymore, and we're back to seeing a move construction.

You can verify this on Compiler Explorer: godbolt.org/z/G1ebj9Yjj

The take away

That means, hands-off! Don't move temporary objects! The compiler does better without us. ■

References

[Orr18] Roger Orr, 'Nothing is better than copy or move' presentation given at ACCU 2018, available at: https://youtu.be/-dc5vqt2tgA

This article was published on Andreas Fertig's blog in February 2022 (https://andreasfertig.blog/2022/02/why-you-should-use-stdmove-only-rarely/).

Afterwood

Quotes and aphorisms are often used to emphasise a point. Chris Oldwood shares some of his favourites and considers their origins.

One of the benefits of living on a small island (Great Britain) is that you have plenty of old fashioned nautical terms to draw on when discussing matters of software development. For example, I've contributed a short segment to the Early Career's Day at the ACCU Conference in recent years and you can probably imagine my joy at realising I'd be talking about quality in software development in Bristol – home of the expression 'shipshape and Bristol fashion' – which naturally I adopted as my title.

One of my other nautical favourites that I'm accused of saying far too regularly (and causes my children to roll their eyes) is 'don't spoil the ship for a ha'p'orth of tar'. (The word ha'p'orth is a contraction of halfpennyworth.) Sadly, this gets more airing than I'd like because the quality conversation can tend occasionally towards cutting corners instead of taking that little bit of extra time to refactor more deeply or add test cases to cover the error scenarios.

There are of course plenty of nautical terms which are still in common use by normal people too and I'm not adverse to 'showing someone the ropes' or 'trying a different tack', although I don't think I could ever bring myself to 'on-board' someone (it's a phrase I'd happily give a wide berth to). Interestingly, the not too uncommon expression 'a rising tide lifts all boats', which I find particularly useful when trying express the importance of team members sharing their time and knowledge for the greater good, is believed to be a fairly modern invention.

What I like about many of these old-fashioned terms is that they add a little colour to what can be a somewhat abstract but nonetheless contentious topic. While the phrase 'best practice' gets tossed around a lot, it's debatable whether any are truly 'best' – more likely is that they are better than others in some circumstances. Hence any conversation around deciding how much effort to spend on improving matters in any given situation becomes less objective, and more subjective, and therefore largely about what your gut instinct tells you.

Once the conversation enters this abstract territory it can become (as an old work colleague once described it) a game of Top Trumps where you use quotes from various industry 'luminaries' to try and back up your side of the argument. I suspect no topic in software development has anywhere near as many sayings as those about simplicity.

For example, in languages like C# and Java it is not uncommon to be faced with a solution to a problem which is implemented in a dizzying array of interfaces and classes when a couple of extension methods could just as easily do the job. For this scenario I like to play the John Carmack card [Carmack11]:

> Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function.

As the author of such seminal games as Doom and Quake his opinion should hold a lot of sway, but if you're dealing with code from someone more classically trained you might need to draw on someone from a different era. Your deck probably holds a solid collection of quotes from the legendary Sir Tony Hoare (with the most heavily worn card likely being one on premature optimization) but I find this observation of his particularly useful for proposing further refactoring [Hoare]:

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

In extreme cases the protagonist may choose to counter with their performance card which luckily you can quickly neutralize with the aforementioned Sir Tony Hoare power card. But you may feel the need to finish off the round once and for all by hitting them with a double whammy by plucking out Hal Abelson's famous words from *Structure and Interpretation of Computer Programs* [Abelson96]:

> Programs must be written for people to read, and only incidentally for machines to execute.

For a trifecta you might consider playing Martin Fowler's variation about any fool being able to write code a computer can understand, but an ad hominem attack like this would be more Donald Trump than Top Trump, so don't.

The benefits of deleting code can never be overstated either, especially dead code and comments which provide no value, and, more importantly, code which can be further simplified by leveraging existing features of the language or standard library. For this we need to flick through the cards from our early 20th century section and draw something wonderfully profound from Antoine de Saint-Exupéry [Saint-Exupéry39]:

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Not all programming quotes can or should be weaponised though. Sometimes we can be too quick to judge the efforts of our ancestors and ascribe the actions to malice or stupidity when in fact it was neither. This quote from Gerry Weinberg is a wonderful reminder about how hindsight is 20/20 [Weinberg]:

> Things are the way they are because they got that way ... one logical step at a time.

I think we are fortunate now to be living in an age where less emphasis is being placed on talking about failure and more on using it as an opportunity to learn. The late

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

The benefits of deleting code can never be overstated either, especially dead code and comments which provide no value

Fred Brooks [Brooks] has a particularly memorable quote which I think extols that notion of continuous personal development:

> Good judgement comes from experience, and experience comes from bad judgement.

Despite being relatively new in comparison to the maritime industry we are still blessed with plenty of our own expressions to draw from. As Andrew Tanenbaum once said (sic) "the good thing about quotes is that there are so many to choose from." ■

References

[Abelson96] Harold Abelson and Gerald Jay Sussman (1996) *Structure and Interpretation of Computer Programs, 2nd Edition*, published by MIT Press.

[Brooks] Frederick (Fred) Brooks Jr (1931-2022) was an American computer scientist and software engineer, who wrote The Mythical Man Month. See https://en.wikipedia.org/wiki/Fred_Brooks

[Carmack11] John Carmack, posted 31 Mar 2011 on Twitter: https://twitter.com/ID_AA_Carmack/status/53512300451201024

[Hoare] Sir Antony Hoare, the quote is referenced in many places, including https://computerhistory.org/profile/sir-antony-hoare/

[Saint-Exupéry39] Antoine de Saint-Exupéry (1939) *Terre des Hommes*, (mostly) translated into English with the title *Wind, Sand and Stars*. See https://en.wikipedia.org/wiki/Wind,_Sand_and_Stars for an explanation of the differences.

[Weinberg] Gerald Weinberg, American computer scientist and author (1933-2018).

So, what does it mean?

The problem with expressions such as 'ship-shape and Bristol fashion' is that they aren't always as obvious as you might think to people who haven't heard them before. If you're not from the UK – or you're under 40 years of age (my guess) – you may not recognise them, or even if you do, your understanding of their meaning may be a little vague. A quick online search soon sorts it out, although you may find many suggestions of the origins of some! The current meaning is fairly standard, though, regardless of how the phrase started.

Ship-shape and Bristol-fashion: The 'ship-shape' part doesn't have so much to do with the appearance (although that may be part of it) but more a claim that everything is as it should be, in its right place (construction and contents) – ready for sea. But why Bristol-fashion? Bristol is a historic port, but isn't on the coast. Instead, it's on a tidal river. This means that any ships – sometimes full of cargo – had to be able to withstand being dumped unceremoniously on the mud when the tide went out and cope with a strong tidal flow. The strain on the construction was greater than when floating. So, if 'ship-shape' is 'ready to go', 'Bristol-fashion' is probably the 'high-quality' element. (https://wordhistories.net/2017/10/18/shipshape-bristol-fashion/)

Don't spoil the ship for a ha'p'orth of tar: Interestingly (to me, anyway) although this is 'obviously' a nautical expression, that may not actually be where it started. It's believed that 'ship' is actually 'sheep' (pronounced 'ship' in some dialects, and therefore written that way when literate non-farmers wrote it down) and 'tar' *is* tar – but was used to keep flies from sores, not to waterproof a hull. It does make sense with the commonly accepted derivation, though. (https://wordhistories.net/2017/10/13/spoil-ship-haporth-tar/)

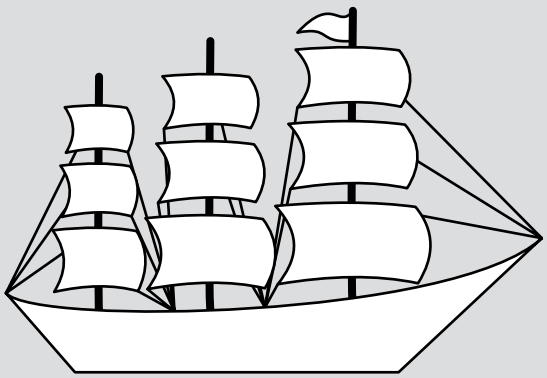
Trying to save time/effort/cost when what you're not doing is trivial in those terms but could have a devastating effect on the success of the overall project/task is the meaning, regardless of the origin.

Showing someone the ropes: The sails on large sailing ships were raised and lowered using ropes. And some had a lot of sails, and therefore a lot of ropes. It wasn't always obvious which rope did what to which bit of sail. This is a straightforward one: making sure people know how to do what they have to do. (https://en.wiktionary.org/wiki/show_someone_the_ropes)

Changing tack: Very much a nautical term, in use today. A sailing boat can't sail directly into the wind, but tacking (changing its direction relative to that wind) enables the wind to alternatively blow into the sails from the port (left) and starboard (right) sides. This moves the boat generally into the wind, in a zig-zag pattern. (https://www.safe-skipper.com/tacking-a-sailing-boat/)

Changing tack means changing the way you approach a task or an issue. Using different methods. Resolving an issue in a different way.

Onboarding: Sounds nautical, but appeared in the 1970s and always to do with new employees going through an induction programme. (https://www.businesstoday.in/lifestyle/wordsmith/story/meaning-of-the-word-onboarding-21672-2011-08-05)



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

"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