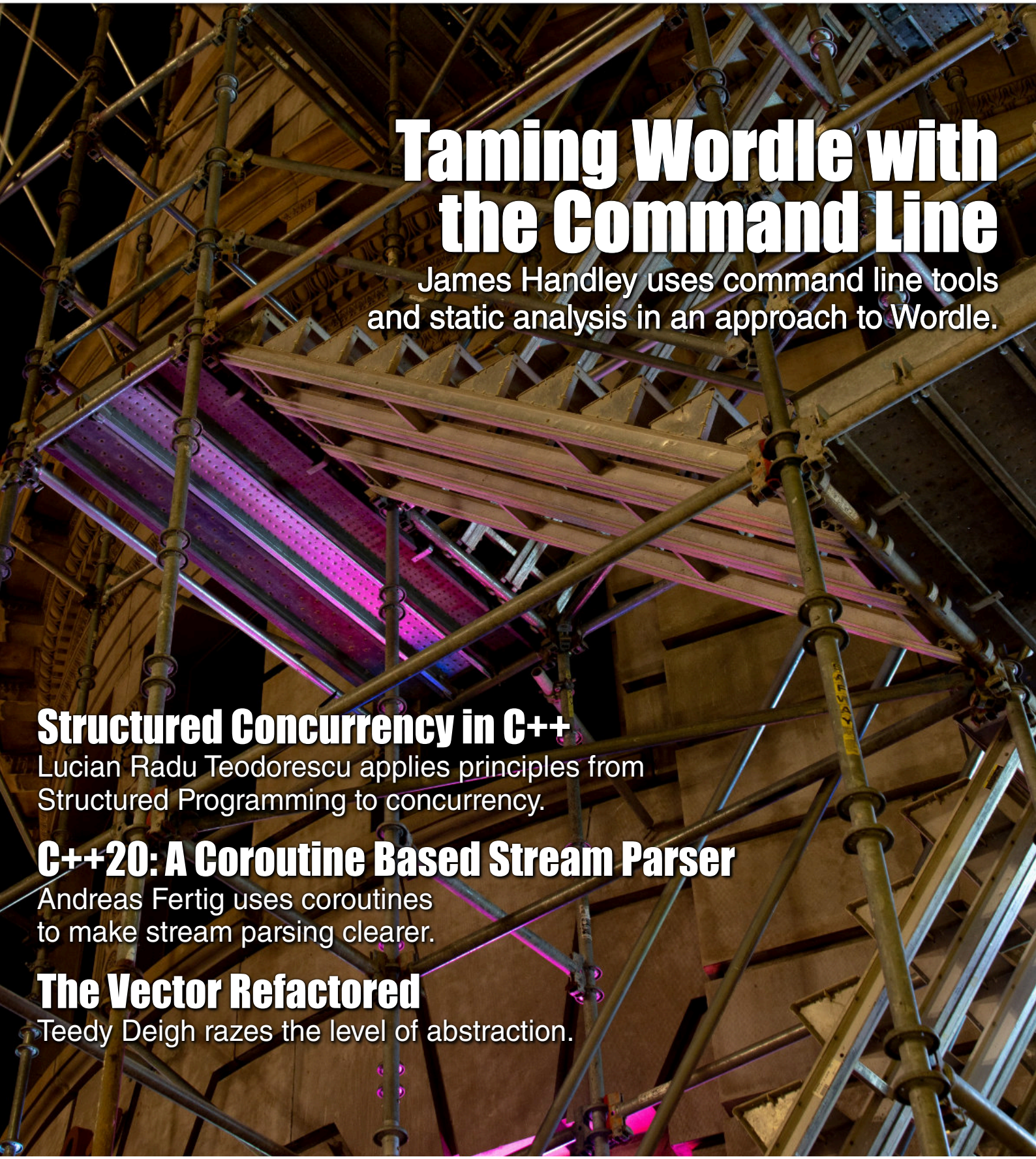
overload 168
APRIL 2022 £4.50



Taming Wordle with the Command Line

James Handley uses command line tools and static analysis in an approach to Wordle.

Structured Concurrency in C++

Lucian Radu Teodorescu applies principles from Structured Programming to concurrency.

C++20: A Coroutine Based Stream Parser

Andreas Fertig uses coroutines to make stream parsing clearer.

The Vector Refactored

Teedy Deigh razes the level of abstraction.

A magazine of ACCU
ISSN: 1354-3172

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

OVERLOAD 168

April 2022

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Dustin Humes on Unsplash.

Copy deadlines

All articles intended for publication in Overload 169 should be submitted by 1st May 2022 and those for Overload 170 by 1st July 2022.

The ACCU

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

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

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

Copyrights and Trade Marks

On Becoming Unstuck

The horsemen of the apocalypse may be on the horizon. Frances Buontempo attempts to stop doom-scrolling and solve problems instead.

I haven't written an editorial. I was stuck for ideas. Instead, I let my mind wander and once all the news of impending doom, jobs to do around the house, talks to prepare and the like bubbled up and got added to a to-do list, I was tempted to wander off myself. I did spend some time collecting all my hand scribbled to-do lists and putting them in once place. I should probably make a new list and throw out the old ones. I'll play today's Wordle first though – little rituals to make yourself sit still long enough to start concentrating can be very important. Playing some music works too. If I play a whole album rather than leave random play on, I can persuade myself to sit at my desk for the duration of the music and thereby manage to get stuff done too. Good music, goal accomplished: win, win.

It's not always that easy though, and the stuff I get done may not be what I first intended. Distractions abound when you don't have a clear goal in mind, can't form your list into a priority queue, or worse find the list is more like a tree structure with each thing you think of kicking off twice as many extra things to deal with. You may have heard of Feynman's algorithm, "facetiously suggested by Murray Gell-Mann, a colleague of Feynman, in a New York Times interview" [c2.com14]:

1. Write down the problem.
2. Think very hard.
3. Write down the solution.

On the face of it, the suggestion seems trite and unhelpful. Furthermore, some suggest that the algorithm only works if you are Feynman himself. Variations on the same idea have been suggested, for example from Hacker News [Hacker]

1. Write down the problem.
2. Try several established strategies to solve it.
3. Get stuck, resign self to failing.
4. Go to sleep.
5. Write out the answer over breakfast.

Aside from the apparent frivolity, trying to write down a problem often helps one to focus on a specific goal. Like talking to the fabled rubber duck to debug an issue, trying to be clear about what you are stuck on might help you get unstuck. I say 'unstuck'; I am aware that 'coming unstuck' can also mean everything falling apart and failing, but let's stay positive and stick to problem solving instead of doom-scrolling.

Stepping away from the keyboard and doing something else halts doom surfing, and likewise many problems can be solved by doing something completely different, though this may get mislabelled as work avoidance by the doubters. I once decided it was far more important to wash the skirting boards than revise for a looming exam. Some friends at university tried to 'mow' the lawn with nail scissors. We've all been there. Eventually, you have to face whatever it is you were supposed to be doing. I suspect changing from sitting down staring at a book or screen and physically moving shifts your perspective. Going for a walk or getting some other form of exercise can clear your head too. It might be because you are still thinking on the problems or issues while doing a menial task, but can't keep digging down deeper in the hole you started, making more notes and ending up with an exponential number of problems. If you can't just get up and go for a walk, because you are stuck in an office – remember those days? – or in an exam or similar, try a different problem. If you're stuck on one question, try another one and revisit the first one later. In a similar vein, George Polya once said "If you can't solve a problem, then there is an easier problem you can solve: find it" [Polya73]. This advice was aimed at solving mathematical problems. To solve an equation, guess a value and see if it works. In the process you might notice ways to eliminate some terms or similar. Resorting to solving a simpler problem helps too. If you are trying to integrate something messy looking, substituting a variable might work. This can be a bit like extracting a method when you refactor – leaving less cognitive load by breaking up complicated ideas. Flushing out edge cases and impossibilities leaves less to think through as well. Similar tricks work when programming, though don't forget, "We can solve any problem by introducing an extra level of indirection, except for the problem of too many levels of indirection" [Wikipedia-1]. If you can't figure out an algorithm using a tree structure, maybe try an array first. With one item in. Or, even none. Sometimes tidying up code, or writing some documentation, or deleting code helps shift your focus enough to go back and tackle the initial problem. "A change is as good as a rest", as the saying goes.

Polya's advice was aimed at getting students themselves to form a plan to tackle problems rather than the teacher giving step by step instructions. This helps students to actually learn. Solving a different problem that is similar frequently hints at ways forward and noticing connections like these shows knowledge is growing. An extreme version of solving a different problem would be coming out with Plan B if Plan A doesn't work. It is hard to know when to abandon hours of work and try something else, since the alternative may not work either. Nonetheless, the different perspective might suggest a way forward for the first attempt. You won't know until you try. If neither plan works, resort to moving the goal posts. This might mean scratching some items off a to-do list or asking a customer if some feature can wait for another day. If you get stuck solving one problem, give up and solve a different one.

Some problems are hard. In fact some problems, such as the halting problem are NP-hard. In case you have forgotten the definition:

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

> A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP-problem (nondeterministic polynomial time) problem. NP-hard therefore means 'at least as hard as any NP-problem', although it might, in fact, be harder. [Wolfram]

If that didn't help much, the salient point here is some problems can be reduced to other problems, which you could try to solve instead. If you can't solve the problem and someone else offers a solution, you might be able to verify it works. Perhaps the main lesson from the P=NP question, and also Feynman's algorithm, is to delegate to someone else.

If you can't move the target or abandon requirements and have a mammoth task in front of you which you can't delegate, start with the first step. One thing at a time. It's not always obvious where to start though. A large task, like preparing a conference talk or writing a book can be daunting. So, where do you start? Truth be told, usually in the middle. From the outside, a title and table of contents may appear to be produced first, however some thoughts or sketchy notes for a few important ideas that fit somewhere, exactly where to be decided, usually come first. Maybe start with a few tests, then make a to-do list? Maybe draw a mock up on a white board? Perhaps read up on the state-of-the-art – a very important part of research. We can't all spend our time reinventing the wheel. As you read around a subject you may find someone else has already solved your problem. Job done. Unfortunately, you may Google a problem and find you've asked the exact same question years ago on a forum somewhere. And still no one knows the answer. That's possibly a good thing. Sit back, relax and cherish having something difficult to think about. Take a deep breath and have another go.

Some problems do have known solutions, but you can still get stuck when you apply them. For example, some techniques for root finding [Martin16] require an initial guess which gets updated in a loop. Annoyingly, the algorithm can head to other roots, go to infinity, or get trapped in cycles. In order to find where a function, f, has the value zero, also known as root finding, you can start with an initial guess of x_0 and find a better approximation x_1 by calculating

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

where f' is the derivative of f. Then reapply the formula to the new approximation and keep going. If you can't find the derivative, you're slightly stuck, but could try to calculate that numerically. Once you have f', you might end up with a division by zero problem. Also, a bad initial guess can get you in trouble too. [Wikipedia-2]. You might have to think carefully about where to start before applying this and similar techniques. As a saying goes, "If I wanted to get there, I wouldn't start from here." At the very least, if you are tempted to have code in a `while` loop waiting for a condition to be met, consider dropping out after a pre-specified number of iterations so you don't loop forever. Instead of iterating, some algorithms rely on recursion. Instead of getting stuck in a loop, you are likely to hit a stack overflow at some point if your stopping condition is wrong. You may not be stuck, but you certainly haven't solved your problem.

There are many ways to get stuck and many things to get stuck on. Patricia Aas recently asked the world what the longest time people had spent debugging an issue was and how big the fix was [Aas22]. As you can imagine, there were many replies. My favourite was someone spending a day debugging an I/O issue until a colleague pointed out the network cable was unplugged. I confessed to having spent several days looking at a networking issue for some low-level embedded devices. I even spent hours looking at the output on a protocol analyser. Eventually the presence of a `\n` character came into focus. The nice small fix was to change it to a `\r`. Many problems in IT are hard to spot, but when you find them you might only need one small change to sort things out. That might seem as unhelpful as Feynman's algorithm, but reminding myself the issue is probably something small, simple and obvious, even though I haven't found it yet can help me avoid panicking and start being more systematic about how to troubleshoot.

Having the wrong line ending character was my fault and quickly fixed; however, sometimes things beyond your control cause problems. We're having our house rewired; annoying but necessary. This gave us the opportunity to put in a network cable because Wi-Fi doesn't always work very well through old stone walls. You'll never guess what happened next. Well, you might. The internet stopped working. The new cable was to connect the main router to another in the office, so had no reason to stop the internet into the house. We couldn't fix it so had to phone our internet provider's help line. It 'magically' started working again during the course of the call. Interesting. We were using our own router, so they decided to send us one of theirs in case it happens again. This time you really won't guess what happened next. I discovered a package, yep the new router, literally stuck in the letterbox. I guess the posty had tried to shove it through the letterbox, failed, couldn't get it back out and ran away. It needed two of us to hold the letter box open and force the parcel back out. If something is about to get stuck, maybe give up and summon help? Just saying.

References

[Aas22] Patricia Aas, on Twitter: https://twitter.com/pati_gallardo/status/1499517651999420416

[c2.com14] The Feynman Algorithm: http://wiki.c2.com/?FeynmanAlgorithm (page last updated December 2014)

[Hacker] Hacker News: https://news.ycombinator.com/item?id=14191681

[Martin16] Patrick Martin, 'Eight Rooty Pieces', *Overload*, 24(135):8-12, October 2016. https://accu.org/journals/overload/24/135/martin_2294/

[Polya73] George Polya, *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press, 1973. See also https://en.wikipedia.org/wiki/How_to_Solve_It

[Wikipedia-1] Fundamental theorem of software engineering: https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering

[Wikipedia-2] Newton's method: https://en.wikipedia.org/wiki/Newton%27s_method#Failure_of_the_method_to_converge_to_the_root

[Wolfram] NP-Hard Problem, Wolfram Mathworld, at https://mathworld.wolfram.com/NP-HardProblem.html#:~:text=A%20problem%20is%20NP%2Dhard,%2C%20in%20fact%2C%20be%20harder

Taming Wordle with the Command Line

Could static analysis provide a generic way to approach Wordle? James Handley uses simple command line tools in order to (hopefully) name that Wordle in four!

For many people (myself included) Wordle [Wordle] has become part of the daily routine. Wordle is a simple 'Mastermind'-like game where you have to guess a five-letter word in six guesses or fewer. For each guess you make (each of which has to be a valid five-letter word), you get an indication of whether each letter appears in the solution, and if it does whether you have guessed it in the correct position. One important thing to note is that the spelling is American English.

```
$ grep -P "^[[:ascii:]]*$" /usr/share/dict/american-english-huge | \
  grep "^[a-z][a-z][a-z][a-z][a-z]$" > wordle-words.txt

$ wc -l wordle-words.txt
11302 world-words.txt
```

Listing 1

After a several weeks of solving it with random guessing, I decided to try and be a bit more scientific, in part thanks to a little extra prod from a video by Hannah Dee [Dee22].

First things first. If the answer didn't have to be a valid word you are unlikely to solve it. There are $26\times26\times26\times26\times26$ (over 11 million) permutations possible. While you can rule out a lot of these permutations after every guess, basic arithmetic exposes the problem. If we guess 5 different letters each turn, then we will have tried 25 different letters in 5 guesses. For our final guess, we will know which distinct letters are in the solution, but can't be sure of their position, or if any have been repeated. Or to look at it in a more Wordle way – on the first guess, say we happen to get four letters in the correct position. That still leaves 25 possibilities for the fifth letter (as one may be repeated), but only 5 guesses to find it. SHAME, SHAPE, SHALE, SHADE, SHAKE, SHAVE anyone?

Fortunately it does have to be a valid word, which helps enormously. There are 'only' 11,302 words in Linux's 'huge' American English dictionary, from "aahed" to "zymix". We can rule out a lot of these, but we'll use this list as the worst case. Most of the analysis in this article uses two GNU commands – **grep** (a regular expression line matcher) and **wc** (a word counter). We will also pipe the output from command to the next. There are of course multiple regular expressions to solve any given problem [RegEx] – I'm using ones which (hopefully) are easy to read.

Each word in the source dictionary file is on a separate line, but there are words with diacritics, apostrophes, and some proper nouns. First of all we'll need to filter it to only ASCII characters, and then to lines with exactly five lower case letters. We'll write the result into a file called `wordle-words.txt` to use throughout the rest of the analysis (see Listing 1).

The approach I am taking is to try to extract as much information as possible from each individual guess – the theory being that at each stage you are reducing the search space by the greatest amount. I am using guesses based on the letters which are most likely to occur in the solution, which provides positional information where there is a match, or significantly reduces the candidate word list where there isn't. This is unlikely to be the optimal approach, but has the advantage of being easy to understand and implement.

We can use some **grep** magic to count how many times each letter of the alphabet appears in the full word list. The flag **-o** will output every match on a new line, for example grep -o "." on "ABA" will output "A", "B", "A" on separate lines. We can then sort this output, and group/count it with uniq. The final two commands in Listing 2 aren't strictly required – they sort the output numerically and output it in columns.

```
$ grep -o "." ./wordle-words.txt | sort | uniq -c | sort -g -r | column
   5807 s   2828 l   1637 c   1194 k   228 j
   5723 e   2722 t   1628 m    869 f    92 q
   5242 a   2414 n   1531 y    856 w
   3832 o   2013 d   1387 h    593 v
   3636 r   1898 u   1334 g    340 z
   2901 i   1661 p   1307 b    237 x
```

Listing 2

The 5 most common letters are S, E, A, O and R, and AROSE is the only word we can make from those five letters. If we count all the words which have at least one of these letters, it's an amazing 10,782 out of 11,302. This means we have a 95% chance matching at least one letter, and if we don't match any letters at all we've only 520 candidate words left to search.

```
$ grep "[arose]" wordle-words.txt | wc -l
10782
```

On the other hand if we used the least popular letters (say BUZZY), we only cover 4,665 words. Our odds of a match have gone down from 95%

Key to the examples:

A – this is a correct letter, in the correct place.

A – the letter exists in the word, but not in this place.

A – this letter does not exist in the word.

James Handley The Rev'd James Handley has an MEng in Software Engineering, a PhD in Computing and Medical Physics, and a BA in Theology. He is a Principal Software Developer at JBA Consulting as well as being an ordained priest in the Church of England. For the past 15 years or so he has specialised in GIS and mapping, and he is particularly interested in how software development can influence faith and ministry, and vice versa. He can be contacted via revjameshandley@gmail.com

there is only one five-letter word which does not have any letters from "AROSEUNTIL" in it, but that word is left as an exercise for the reader!

to less than 50%. What's worse, if we don't match any, we are still left with 6,637 words to search.

First word	Number of matches	Number of non-matches
AROSE	10,782	520
BUZZY	4,665	6,637

Ignoring the positional information for a moment, we can break down what happens when we match combinations of letters. If we only match "A" then we know we didn't match any of "ROSE", which only leaves 797 possible words:

```
$ grep "[arose]" wordle-words.txt | \
  grep "a" | \
  grep -v "[rose]" | wc -l
797
```

Do the above for all the possible combinations of letters (we've already seen that AROSE is the only word with all these letters) and you get the following:

A	797	AR	431	RS	234	ARO	153	ASE	409	ROSE	85
R	189	AO	323	RE	522	ARS	315	ROS	186	AOSE	10
O	531	AS	894	OS	659	ARE	380	ROE	211	ARSE	123
S	667	AE	561	OE	417	AOS	181	RSE	283	AROE	12
E	789	RO	249	SE	823	AOE	47	OSE	255	AROS	45

Even without positional information, we have reduced the word list from 10,782 to a maximum of 894.

By taking into account positional information, we can reduce it even further. Say we need an "A" in the correct position:

```
$ grep "[arose]" wordle-words.txt | \
  grep "a" | \
  grep -v "[rose]" | \
  grep "a...." | wc -l
123
```

That means there are 123 five-letter words which begin with A, but don't have the letters ROSE. Inverting the final grep would give us the count for "correct letter, wrong position". Completing this analysis for our five starting letters:

AROSE	123	AROSE	674
AROSE	77	AROSE	112
AROSE	140	AROSE	391
AROSE	54	AROSE	613
AROSE	252	AROSE	537

The chances are we will match more than one letter, and all the possible combinations would be too many to list here. But for the least specific combination ("AS") we get:

AROSE	17
AROSE	68
AROSE	59
AROSE	750

So it seems reasonable to estimate that our new candidate word list has a maximum of 750 words. Not bad for one guess.

As we saw with SHAME/SHAPE/etc., changing only one letter is not a good strategy. The 'best' second guess will depend on the match of the first guess, but if we are taking a generic approach we will want choose a distinct set of letters to the first guess. The next 5 on the list are I, L, T, N, and U – "UNTIL" (or "UNLIT" if you prefer). Our coverage with UNTIL against the full list is still pretty good at 8,690 matches vs. 2,612 non-matches. It also turns out there is only one five-letter word which does not have any letters from "AROSEUNTIL" in it, but that word is left as an exercise for the reader!

The 'best' third guess depends even more on the first two matches, and you will have to at least re-use a vowel. The next most frequent letters are D, P, C, Y, and M, and H so trying to make a word from these letters with whichever vowels you already know is probably your best bet. Assuming some match with "AROSEUNTIL", it turns out that the third most frequent letter set doesn't actually change very much.

So, my approach to solving Wordle:

1. Guess "AROSE"
2. Guess "UNTIL"

   If no match with either, there is only one possible word.
3. Guess a word with the letters you know, also using as many letters as possible from DPCYM.
4. You should now be able to work out the answer!

This is what happened for me with Wordle #244:

1. AROSE – 107 words left
2. UNTIL – 20 words left
3. DOPEY – 1 word left
4. DODGE

References

[Dee22] Hannah Dee – "A Linux refresher through the medium of Wordle" posted on 25th January 2022. https://www.youtube.com/watch?v=i4UipSGjaNQ

[RegEx] "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." Attributed to Jamie Zawinski – see for example http://regex.info/blog/2006-09-15/247

[Wordle] https://www.nytimes.com/games/wordle

C++20: A Coroutine Based Stream Parser

Stream parsing code can be very complicated. Andreas Fertig uses coroutines to make stream parsing code clearer.

In this article, I want to show you how C++20's coroutines can help you when writing a stream parser. This is based on a protocol Andrew Tanenbaum describes in his book *Computer Networks*. The protocol consists of a Start Of Frame (SOF) and ESC to escape characters like SOF. Hence, ESC + SOF mark the start of a frame as well as the end. In today's exercise, we will parse the string **`Hello World`** (yes, without the comma, sorry). For simplicity, I am using strings and not bytes. The steam shown in Figure 1 is what we are going to process.

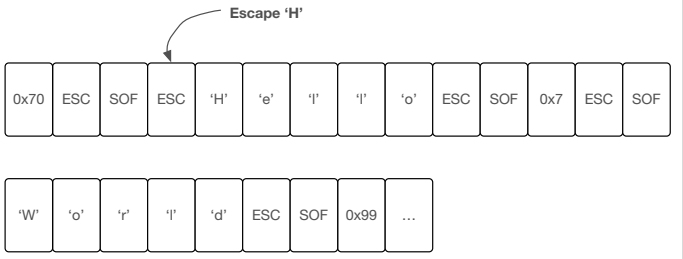


Figure 1

The classic way

Before C++20, we could have quickly written a function **`ProcessNextByte`**, which would have dealt with tokenizing the stream. It looks for the ESC + SOF for the start of a frame and calls the callback **`frameCompleted`** once a frame is complete. Error cases are not treated for simplicity here. They are silently discarded. Listing 1 is a version of **`ProcessNextByte`**.

There are other ways to implement this, such as using a class and getting rid of the **`static`** variables. However, a function is essentially what we like here. This is not really a job for a class. Even with a class, tracking the state and quickly seeing where we are remains complicated. Another way, of course, is to bring in the big guns and use some kind of state pattern using virtual methods. I once had the pleasure of working with such a beast. It turned out that it was very hard to see the control flow, and in the end, all calls ended up in one single class, which contained the actual logic.

However, one upside of using a class would be that we can have multiple objects parsing multiple streams. With the given **`ProcessNextByte`** we can only parse exactly one stream.

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.info

```
template<typename T>
void ProcessNextByte(byte b, T&& frameCompleted)
{
  static std::string frame{};
  static bool        inHeader{false};
  static bool        wasESC{false};
  static bool        lookingForSOF{false};

  if(inHeader) {
    if((ESC == b) && not wasESC) {
      wasESC = true;
      return;
    } else if(wasESC) {
      wasESC = false;

      if((SOF == b) || (ESC != b)) {
        // if b is not SOF discard the frame
        if(SOF == b) { frameCompleted(frame); }

        frame.clear();
        inHeader = false;
        return;
      }
    }
    frame += static_cast<char>(b);
  } else if((ESC == b) && !lookingForSOF) {
    lookingForSOF = true;
  } else if((SOF == b) && lookingForSOF) {
    inHeader      = true;
    lookingForSOF = false;
  } else {
    lookingForSOF = false;
  }
}
```

Listing 1

Coroutines applied

Listing 2 (overleaf) is the same logic, this time as a coroutine.

There are various great things. First, **`Parse`** is not a template, as **`ProcessNextByte`** was. By looking at the function's body, the states are much clearer to see. We have two infinite-loops (which is a bad thing on some embedded systems as infinite-loops can usually be avoided) where the outer is really infinite and the inner runs as long as the frame is incomplete. We can see that this code first looks for **`ESC`** and then for **`SOF`** ❶. If **`ESC`** is found, but the next byte is not **`SOF`**, the search for **`ESC`** starts over. The logic above was way more complicated and harder to see through.

Once **`ESC`** followed by **`SOF`** is found, the inner loop starts ❷. It adds all received bytes to the local variable frame. But before that, **`Parse`** checks whether we are looking at an escape sequence ❸. If so, the escape character is removed and replaced by the unescaped version. In our case, this is

```
FSM Parse()
{
  while(true) {
    byte        b = co_await byte{};
    std::string frame{};

    if(ESC == b) {
      b = co_await byte{};

      if(SOF != b) { continue; }
      // ❶ not looking at a start sequence

      while(true) { ❷ capture the full frame
        b = co_await byte{};

        if(ESC == b) {
          // ❸ skip this byte and look at the
          // next one
          b = co_await byte{};

          if(SOF == b) {
            co_yield frame;
            break;
          } else if(ESC != b) { // out of sync
            break;
          }
        }
        frame += static_cast<char>(b);
      }
    }
  }
}
```

Listing 2

necessary for **ESC** as its value is **'H'**. As there are no other characters we need to escape, for now, we only check whether the byte following **ESC** is either **SOF** or **ESC**. Both are valid. Any other byte is a processing error, which we silently discard by throwing away the current frame and starting over from the beginning.

I claim that this code is fairly straightforward to read, without knowing what **co_await** and **co_yield** do. I'm sorry about the **co_**, but coroutines were late to the party, and keywords like **await** or **yield** were already used in code-bases, hence the prefix.

We can also see something new. **Parse** returns **FSM**, my name for this coroutine type.

While I like most about the version I showed you above, there is something I don't like as much. We need to write a huge amount of boilerplate code to make all the coroutine elements come alive. Sadly, C++20 does not ship with a coroutine library. Everything in the coroutine header is a very low-level construct, the building blocks required for a great coroutine library. Lewiss Baker maintains **cppcoro** in GitHub [Baker] and is the author of various proposals for a great coroutines library in C++23. If you are looking for a promising shortcut that has a good chance of ending up in the STL, **cppcoro** is the choice.

Coroutines (an explanation from Fran Buontempo)

If you're not familiar with coroutines, have a look at 'Concurrency, Parallelism and Coroutines' by Anthony Williams from *ACCU 2017* [Williams17]. Coroutines are just functions, which can be stopped and restarted or resumed.

> Coroutines are stackless: they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack. This allows for sequential code that executes asynchronously. [cpp]

They hinge on three terms: **co_await**, which suspends the function and returns control to the caller until it's resumed; **co_yield**, which yields a value to the caller (and then picks up where it left off when called again); and **co_return**, which completes execution and returns a value.

```
template<typename T, typename U>
struct [[nodiscard]] async_generator
{
  using promise_type =
    promise_type_base<T, async_generator,
    awaitable_promise_type_base<U>>;
  using PromiseTypeHandle =
    std::coroutine_handle<promise_type>;
  T operator()()
  {
    // ❶ the move also clears the mValue of the
    // promise
    auto
      tmp{std::move(mCoroHdl.promise().mValue)};
    // ❷ but we have to set it to a defined state
    mCoroHdl.promise().mValue.clear();
    return tmp;
  }
  void SendSignal(U signal)
  {
    mCoroHdl.promise().mRecentSignal = signal;
    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
  }
  async_generator(async_generator const&) =
    delete;
  async_generator(async_generator && rhs)
  : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)}
  {}
  ~async_generator()
  {
    if(mCoroHdl) { mCoroHdl.destroy(); }
  }
private:
  friend promise_type;  // ❸ As the default ctor
            // is private G needs to be a friend
  explicit async_generator(promise_type * p)
  : mCoroHdl(PromiseTypeHandle::from_promise(*p))
  {}
  PromiseTypeHandle mCoroHdl;
};
```

Listing 3

BYOC (Bring your own coroutine)

Let's look at how we can implement the missing pieces without any other library, only the STL.

The type FSM

First, what is **FSM**?

It is a **using** alias to **async_generator<std::string, byte>**:

```
using FSM = async_generator<std::string, byte>;
```

async_generator is another type we have to write, one way or the other, to satisfy the coroutine interface. This type consists of a special type, or in this case, a using alias **promise_type**. This is a name the compiler looks for to determine whether **async_generator** is a valid coroutine type. Precisely with this spelling! In Listing 3 (overleaf), we see the implementation of **async_generator**.

With the call operator, we get access to the value from inside the coroutine. This is the receiving end of something that can be seen as a pipe. The pipe is filled on the other end, inside the coroutine, by **co_yield**.

We can put data into the pipe with **SendSignal**. This is received inside the coroutine by a **co_await**.

Because **async_generator** holds a communication channel with our coroutine, we only allow this type to be moveable but not copyable. An instance of this type is created for us by the compiler when instantiating an object of type **promise_type**. This is why I chose to make the constructor of **async_generator private** and say that

```
template<typename T, typename G,
  typename... Bases>
  // ❶ Allow an optional base class
struct promise_type_base : public Bases... {
  T mValue;
  auto yield_value(T value)
  {
    mValue = value;
    return std::suspend_always{};
  }
  G get_return_object() { return G{this}; };
  std::suspend_always initial_suspend()
    { return {}; }
  std::suspend_always final_suspend() noexcept
    { return {}; }
  void return_void() {}
  void unhandled_exception() { std::terminate(); }
};
```

Listing 4

promise_type is a **friend**. This is just to prevent misuse or false assumptions.

PromiseTypeHandle is a handle to the current coroutine. With it, we can transfer data between normal and coroutine code (e.g. **co_yield** and **co_await**).

Next up is the **promise_type**. The **using** alias is directing to **promise_type_base**, which is composed of **T**, **async_generator**, **awaitable_promise_type_base<U>.** So, two more new types.

The promise_type_base

First, the reason for the suffix **_base** is that the entire example uses two generators. One for the parsing logic we are looking at and another one for simulating a data stream. Listing 4 is the implementation.

This generator satisfies the **co_yield** interface of a coroutine. A very rough view is that the call **co_yield** is replaced by the compiler calling **yield_value**. So **promise_type_base** serves as a container for the value coming from the coroutine into our normal code. All the other methods are just to satisfy the coroutine interface. As we can see, coroutines are highly customizable, but that is a different story.

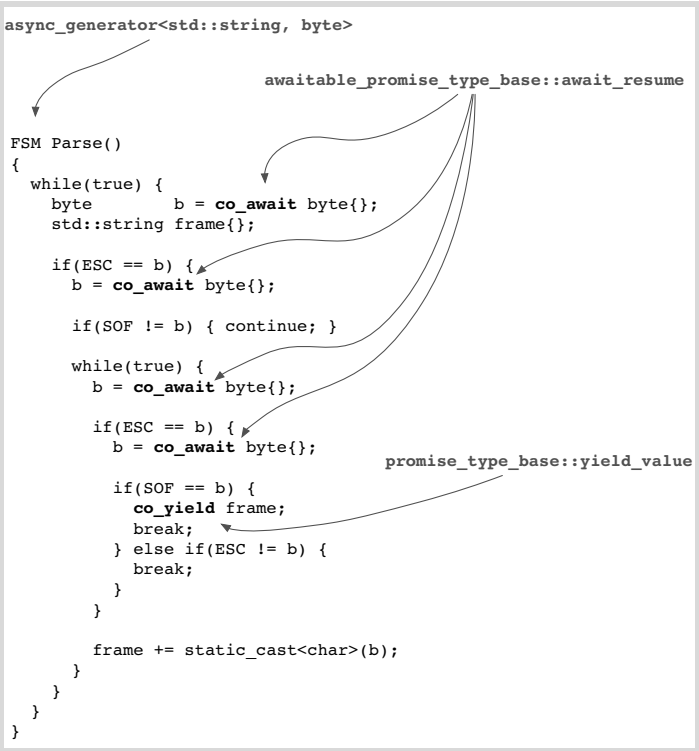


Figure 2

```
template<typename T>
struct awaitable_promise_type_base {
  std::optional<T> mRecentSignal;
  struct awaiter {
    std::optional<T>& mRecentSignal;
    bool await_ready() {
      return mRecentSignal.has_value(); }
    void await_suspend(std::coroutine_handle<>) {}
    T await_resume()
    {
      assert(mRecentSignal.has_value());
      auto tmp = *mRecentSignal;
      mRecentSignal.reset();
      return tmp;
    }
  };
  [[nodiscard]] awaiter await_transform(T) {
    return awaiter{mRecentSignal}; }
};
```

Listing 5

We can also see that **promise_type_base** derives from the third parameter passed to it. In our case, this is **awaitable_promise_type_base<U>**. This is a variadic argument, simply to allow creating a **promise_type_base** object without a base class.

The awaitable_promise_type_base

Next, we need to write the glue code for **co_await**, which waits for the next byte. This is the part of **awaitable_promise_type_base**, creating a so-called Awaitable-type. In Listing 5, you can see an implementation.

With these pieces, we have the **FSM** coroutine type and can use **Parse**. Figure 2 (overleaf) shows which element in our code interacts with which part of the **async_generator**.

Summary

I hope I could demonstrate that coroutines are a great tool when it comes to implementing parsers. The boiler-plate code aside, they are easy to write and highly readable. You can find the complete version of this example on godbolt.org [Fertig].

Until we have C++23 and hopefully much better coroutines support in the STL, we either have to write code as above or use **cppcoro** [Baker].

As always, I hope you learned something from this article. Feel free to reach out to me on Twitter to give feedback (https://twitter.com/Andreas__Fertig). ■

References

[Baker] Lewiss Baker, cppcoro on Github: https://github.com/lewissbaker/cppcoro

[cpp] 'Coroutines (C++20)', available at: https://en.cppreference.com/w/cpp/language/coroutines

[Fertig] An electronic version of the original blog post and all examples is available on godbolt.org: https://godbolt.org/z/8hEffT

[Williams17] Anthony Williams, 'Concurrency, Parallelism and Coroutines' from *ACCU 2017* available at: https://www.youtube.com/watch?v=UhrIKqDADX8

This article was first published on Andreas Fertig's blog (https://andreasfertig.blog/2021/02/cpp20-a-coroutine-based-stream-parser/) on 2 February 2021.

It is a short version of 'Chapter 2: Coroutines', from his latest book *Programming with C++20*. The book contains a more detailed explanation and more information about this topic.

Structured Concurrency in C++

Raw threads tend to be unstructured. Lucian Radu Teodorescu applies principles from Structured Programming to concurrency.

If one made a list of the paradigm changes that positively influenced software development, Structured Programming would probably be at the top. It revolutionised the software industry in two main ways: a) it taught us how to structure programs to better fit our mental abilities, and b) constituted a fundamental step in the development of high-level programming languages.

Work on structured programming began in the late 1950s with the introduction of the ALGOL 50 and ALGOL 60 programming languages, containing compound statements and code blocks, and allowing engineers to impose a structure on programs. In 1966, Böhm and Jacopini discovered what is now called *structured program theorem* [Böhm66], proving that the principles of structured programming can be applied to solve all problems. Two years later, in 1968, Dijkstra published the highly influential article, 'Go To Statement Considered Harmful' [Dijkstra68], arguing that we need to lay out the code in a way that the human mind can easily follow it (and that the `GOTO` statement works against this principle).

It is said that *concurrency* became a concern in the software industry in 1965, when the same Dijkstra wrote the article, 'Solution of a problem in concurrent programming control' [Dijkstra65]. This article introduced the *critical section* (i.e., *mutex*), and provided a method of implementing it – the use of critical sections has remained an important technique for concurrent programming since its introduction. To reiterate the timeline, the critical section concept was introduced before the *structured programming theorem*, before the influential 'Go To Statement Considered Harmful', and before 1968, when the term *Software Engineering* was coined (at a NATO conference [Naur69]).

While concurrency is an old topic in the software industry, it appears that we still mainly handle concurrency in an unstructured way. Most of the programs that we write today rely heavily on raw threads and mutexes. And it's not just the software that's written this way; this is how we teach programmers concurrency. One can open virtually any book on concurrency, or look at most of the concurrency tutorials, and most probably the first chapters start with describing threads and mutexes (or any other form of synchronisation primitive). This is not anybody's fault; it's just the state of our industry (at least in C++).

This article aims to show that we can abandon the thread and mutex primitives for building concurrent abstractions, and start using properly structured concurrency with the senders/receivers framework [P2300R4] that may be included in C++23[1]. We look at the essential traits of structured programming, and we show that they can be found in the senders/receivers programming model.

Structured programming

The term *structured programming* is a bit overloaded. Different authors use the term to denote different things. Thus, it is important for us to define what we mean by it. We have 3 main sources for our definition of structured programming: a) the article by Böhm and Jacopini introducing the *structured program theorem* [Böhm66], Dijkstra's article [Dijkstra68], and the book *Structured Programming* by Dahl, Dijkstra, and Hoare [Dahl72]. The book contains references to the previous two articles, so, in a sense, is more complete.

Based on these sources, by *structured programming* we mean the following:

- use of abstractions as building blocks (both in code and data)
- recursive decomposition of the program as a method of creation/analysis of a program
- local reasoning helps understandability, and scope nesting is a good way of achieving local reasoning
- code blocks should have one entry and one exit point
- soundness and completeness: all programs can be safely written in a style that enables structured programming

In the following subsections, we will cover all these aspects of Structured Programming. Subsequently, we will use these as criteria for defining Structured Concurrency, and decide whether a concurrency approach is structured or not.

Use of abstractions

Abstraction is the reduction of a collection of entities to the essential parts while ignoring the non-essential ones. It's the decision to concentrate on some of the similarities between elements of that set, discarding the differences between the elements. Variables are abstractions over 'the current value'. Functions are abstractions that emphasise 'what it does' and discard the 'how it does it'.

As Brooks argues, software is essential complexity [Brooks95]. That is, we cannot fit it into our heads. The only way to reason about it is to ignore parts of it. Abstractions are good ways to ignore the uninteresting details and keep in focus just the important parts. For example, one can easily understand and reason about something that 'sorts an array', but it's much harder to understand the actual instructions that go into that sorting algorithm. And, even *instructions* are very high-level abstractions compared with the reality of electromagnetic forces and moving electrons.

Use of (good) abstractions is quintessential for structured programming. And to remind us of that, we have the following quote from Dijkstra [Dahl72]:

> At this stage I find it hard to be very explicit about the abstraction, partly because it permeates the whole subject.

Out of all the abstractions possible in a programming language, the named abstractions (e.g., functions) are most useful. Not because they have a name associated with them, but because the engineer can create many of them and differentiate them by name. I.e., having the ability to add `for`

1. This appears not to be true anymore. While writing this article, the C++ standard committee moved the P2300 proposal target release from C++23 to C++26.

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

As much as possible, all code blocks need to be fully encapsulated. Parts of a code block should not interact with the world outside the block

loops in the code is good, but having the ability to create different functions that are capable of implementing any types of problems (including abstracting out `for` statements) is much more useful.

Using functions, the engineer can abstract out large parts of the program and express them with a simple syntactical unit (a function call). This means that functions play an important role in Structured Programming. When we say that *abstractions* are at the core of Structured Programming, we mean exactly this: we're able to abstract away large portions of the program.

Furthermore, it's worth noting that, in Structure Programming, one can find abstractions that would represent the whole program. Just think of the popular `main()` function.

Recursive decomposition of programs

In the *Structured Programming* book, Dijkstra spends a fair amount of time on problem decomposition, on how we're able to build software hierarchically. That is, we're capable of building programs by:

- recursively decomposing programs into parts, using a top-down approach
- making one decision at a time, the decision being local to the currently considered context
- ensuring that later decisions do inot influence early decisions (for the majority of the decisions)

This gives us an (almost) linear process for solving software problems. Any process for solving software problems is good, but a linear process is much better. Software being essential complexity, one can imagine that we might have coupling between all parts of the system. If the system has N parts, then the amount of coupling would be in the order of $O(N^2)$. That is, a process that is focused on the interactions between these parts would have to be in the order of at least $O(N^2)$ steps. And yet, Structured Programming proposes us a linear process for solving problems.

The process advocated by Structured Programming is based on the *Divider Et Impera* strategy, and our brain is structured in such a way that allows us to easily cope with it.

One other key aspect of this method is that the sub-problems have the same general structure as the bigger problems. For example, let's assume that we have one function that calls two smaller functions. We can reason about all three functions in the same way: they all are abstractions, they all have the same shape (single entry, single exit point), they all can be further decomposed, etc. This will reduce the strain on our brains, and allows us to reuse the same type of solution in many places.

Local reasoning and nested scopes

To reduce the burden of understanding programs, Dijkstra remarks that we need to shorten the conceptual gap between the text of the program and how this is actually executed at runtime [Dahl72, Dijkstra68]. The closer the execution follows the program text, the easier it is for us to comprehend it. It is best if we can understand the consequences of running a piece of code by understanding just the preconditions and the corresponding instructions.

If we are following the pattern *preconditions* + *instructions* ⇒ *postconditions*, to properly have local reasoning (i.e., focus on the actual instructions), then we need the set of preconditions to be small. We cannot speak of local reasoning if the set of preconditions needs to contain information about everything else that happened before in the program. For example, if we want to analyse a block of code that sorts an array of numbers, we should not be concerned with how that array was generated; any parts of the sorting algorithm should not be coupled with other parts of the program.

Another way to look at local reasoning is by looking at encapsulation. As much as possible, all code blocks need to be fully encapsulated. Parts of a code block should not interact with the world outside the block. Dijkstra imagines that, in the ideal world, every part of the program would run on its dedicated (virtual) machine. This way, different parts will be completely independent.

In this context, the notion of scope is important. A lexical scope isolates the local concerns of a code block (lexically specified) from the rest of the blocks. If I declare a local variable in my block, no other block can interfere with my local variable. And again, the more stuff we have locally defined, the more local reasoning we're able to do.

Single entry, single exit point for code blocks

Looking at a sequence of regular instructions (i.e., without loops or alternatives) is easy. The preconditions of an instruction directly depend on the postconditions of the previous instruction. This is what Dijkstra calls *enumerative reasoning*. The conceptual gap between a sequence of instructions and the execution of those instructions in time is minimal.

If we want to treat code blocks or function calls as instructions, we should ensure that they share as many properties as possible with the simple instructions. One of these properties is *single entry, single exit point*. Every instruction, every block of code and every function should have one single entry point so that we can easily check whether the preconditions are met. Similarly, they should have one single exit point so that we analyse a single set of postconditions.

There is another advantage of using a single entry, single exit point strategy. The blocks and the function calls have the same shape as simple instructions. That allows us to apply the same type of reasoning to code blocks and to function calls, and permits us to have a simpler recursive decomposition.

Soundness and completeness

Having a single entry and a single exit point is a big restriction on the number of programs we can write. We need to make sure that this restriction does not impose a limit on the number of problems that can be solved with Structured Programming. We must have a sound method to ensure that we are able to build all programs with the restrictions imposed by our method.

We may create abstractions that correspond to tasks, but it's hard to create abstractions of different sizes, spanning multiple threads

The structured program theorem [Böhm66] proves that we can write all our programs using 3 simple control structures: sequence, selection and repetition. The Böhm and Jacopini paper has 3 major takeaways:

- ensuring that Structured Programming can be applied to all types of problems
- providing a set of primitives that can be used as building blocks for our programs
- providing alternative ways to visualise the programs – flowcharts

Concurrency with threads and synchronisation primitives

In the classic model of concurrency, one would use raw threads to express parallelism and then use synchronisation primitives to ensure that the threads do not interact in ways that break the correctness of the program. This model of organising concurrency is completely unstructured.

We don't have a general way of creating higher level abstractions that can represent parts of our concurrent program (and fully handle concurrency concerns inside it). Thus, we cannot use any such abstractions to decompose a program into subparts while keeping the concurrency constraints straight.

In this model, it's generally hard to do local reasoning. With synchronisation primitives, we almost always need to consider the rest of the program. We cannot simply look at one single function and draw any conclusion about what synchronisation we need.

As we don't have good general-purpose abstractions to encapsulate concurrency constraints, it is hard for us to discuss single-entry and single-exit points for such abstractions. Furthermore, as there is no general composable method for expressing concurrency, it's hard for us to discuss the soundness of the approach.

Synchronisation primitives act like `GOTO` commands, breaking local reasoning and encapsulation.

Concurrency with raw tasks

Let's now turn our attention to another model of dealing with concurrency. The one that is based on raw tasks (see [Teodorescu20]). We call a *task* an independent unit of work that is executed on one thread of execution. We call an *executor* something that dictates how and when tasks are executed. Those two concepts are enough for building concurrent systems.

We have proved before that a task-based system can be used to implement all concurrent problems [Teodorescu20] and that we can compose task-based system without losing correctness and efficiency [Teodorescu21]. That is a step towards Structured Concurrency. However, task-based systems still don't fully have the characteristics we've taken from Structured Programming.

Let's first look at the ability to create concurrent abstractions. We may create abstractions that correspond to tasks, but it's hard to create abstractions of different sizes, spanning multiple threads. Taking tasks as they are, without any workarounds, doesn't fully encapsulate concurrency constraints. In this case, they also cannot be used for recursively decomposing concurrent programs.

Simple tasks are equivalent to functions, so they allow local reasoning just like Structured Programming does (that is, only if tasks are independent, as we said previously).

In [Teodorescu21], we introduced a technique to allow the decomposition of tasks. First, the concept of a task is extended to also contain a *continuation*. A continuation is a type-erased function that is executed whenever the task is completed. Most of the machinery that can be built on top of tasks can be made to work by using continuation. Secondly, the article introduced a trick that allows tasks to change their continuations while running. More precisely, to exchange the continuation of the current task with the continuation of another tasks (that can be executed in the future, or on a different thread). With this trick, we can make tasks represent concurrent work that span across multiple threads, with inner details.

If we apply this trick, then we lose the ability for local reasoning. The bigger abstraction (original task + continuation) is not specified in a one place; it is distributed between the start task, and all the tasks that are used to exchange the continuations (directly or indirectly) with this task. We lose lexical scoping and nesting properties.

With raw tasks, it's often the case that we *spawn* tasks and continue. This is the *fire-and-forget* style of creating concurrent work. Each time we do this, the *spawn* operation has one entry point, but two exit points: one that remains with the current thread, and one that follows the newly spawned task.

In conclusion, even if tasks systems allow us to achieve most of the goals we set up for a structured approach, we cannot achieve all the goals at the same time.

Concurrency with senders/receivers

There is a new model in C++ that can solve concurrency in a structured approach. This is the model proposed by [P2300R4], informally called senders/receivers. As we shall see, this model works well with all the important points considered to be 'structured'.

A sender is an entity that describes concurrent work. The concurrent work described by the sender has one entry point (starting the work) and one exit point, with three different alternatives: successful completion (possible with a value), completed with error (i.e., exception), or cancelled.

It is worth mentioning that senders just describe work; they don't encapsulate it. That would be the role of what P2300 calls o*peration states*. However, users will rarely interact directly with operation states. These are hidden behind simple-to-use abstractions.

Listing 1 shows an example of using a sender to describe concurrent work. We specify that the work needs to be executed on `pool_scheduler` (e.g., an object identifying a pool of threads), we specify what work must happen, and we specify that if somehow the scheduler is cancelled to transform this into a specific error. While creating the sender object, no

having a multiple exit strategy doesn't necessarily imply multiple exit points ... function calls are considered to have a single exit point, even if they can exit either with a value or by throwing an exception

```
using ex = std::execution;
int compute_something() {...}

ex::sender auto snd =
    ex::schedule(pool_scheduler)
  | ex::then(compute_something)
  | ex::stopped_as_error(my_stopped_error)
  ;
auto [r] =
std::this_thread::sync_wait(std::move(snd))
  .value();
```

Listing 1

actual work happens; we just describe what the work looks like. The call to `sync_wait` starts the work and waits for it to complete, capturing the result of our concurrent work (or maybe forwarding the exception if any exception is thrown).

For the purpose of this article, we define a *computation* as a chunk of work that can be executed on one or multiple threads, with one entry point and one exit point. The exit point can represent the three alternatives mentioned above: success, error and cancellation. Please note that having a multiple exit strategy doesn't necessarily imply multiple exit points; this is similar to how function calls are considered to have a single exit point, even if they can exit either with a value or by throwing an exception.

A computation is a generalisation of a task. Everything that can be a task can also be a computation, but not the other way around.

The senders/receivers model allows us to describe any *computation* (i.e., any concurrent chunk of work) as one sender. A proof for this statement would be too lengthy to show here, and the reader can find it in [P2504R0][2]. This paper also shows the following:

- all programs can be described in terms of senders, without the need of synchronisation primitives
- any part of a program, that has one entry point and one exit point, can be described as a sender
- the entire program can be described as one sender
- any sufficiently large concurrent chunks of work can be decomposed into smaller chunks of work, which can be described with senders
- programs can be implemented using senders using maximum efficiency (under certain assumptions)

In summary, all concurrent single-entry single-exit chunks of works, i.e., *computations*, can be modelled with senders.

It is important to note that *computations* fully encapsulate concurrency concerns. *Computations* are to concurrent programming what functions are to Structured Programming. *Computations* are the concurrent version of functions.

The above paragraph represents the quintessence of this whole article.

2. The paper uses the term *computation* in a different way to this article. There, *computation* is defined to be equal to a sender, and then it's proven that it can represent arbitrarily chunks of work. It's the same thing, but coming from a different perspective.

```
using ex = std::execution;
ex::sender auto whole_program_sender() {...}
int main() {
  auto [r] = std::this_thread::sync_wait
    (whole_program_sender()).value();
  return r;
}
```

Listing 2

Use of abstractions

Let's start our analysis of whether the senders/receivers model can be considered structured by looking at abstractions. In this model, the obvious abstraction is the *computation*, which can be represented by a sender. One can use senders to abstract out simple computations (smaller than typical tasks), to abstract out work that corresponds to a task, or to abstract out a chunk of work that may span multiple threads. The upper limit to how much work can be represented by a sender is the size of the program itself. Actually, the entire program can be represented by a single sender. See Listing 2 for how this may be done.

Recursive decomposition of programs

As mentioned above, [P2504R0] proves that any program, or part of the program, can be recursively decomposed with the use of senders. Moreover, within a decomposition, the smaller parts can be made to look like the original part; they can be all senders.

Listing 3 shows an example on how a problem can be decomposed into two smaller parts, using senders to describe both the problem and the sub-problems.

Local reasoning and nested scopes

The reasoning can be local with the senders/receivers model. The definition of a sender completely describes the computation from

```
ex::sender auto do_process() {
  ex::sender auto start
    = ex::schedule(pool_scheduler);
  ex::sender auto split_start = ex::split(start);
  return ex::when_all(
    split_start | ex::let_value(comp_first_half),
    split_start | ex::let_value(comp_second_half)
  );
}
```

Listing 3

beginning to end. The reasoning of all the aspects of that sender can be done locally, even if the sender contains multiple parts.

Let us look again at the code in Listing 3. The top-level computation contains the elements needed to reason about how the sub-computations start and finish. We're able to inspect this code and draw the following conclusions:

- one entry and one exit for the entire process
- both halves start processing on the pool scheduler
- if one half ends with error/cancellation, then the other half is cancelled and the whole process ends with the original error/cancellation
- the process ends when both parts end

The idea of nested scopes was a major point in the design of senders/receivers. One can properly nest senders, and also nest *operation states*, and also *receivers nest*. Please see [Niebler21a] for more details.

Single entry, single exit point

By definition, senders have one entry point and one exit point. There are multiple alternatives if the work described by a sender may terminate, but essentially, we may say that there is only one exit point. This is similar to how a function call might exit either with a return value or an exception.

While one can use the *fire-and-forget* style with senders, this is not the recommended way of using senders.

See also [Niebler21b] for a better illustration about how senders are fulfilling the single entry, single exit point requirement.

Soundness and completeness

In Structured Programming, the *structured program theorem* [Böhm66] provides the soundness and completeness guarantees for the method. For the senders/receivers model, as mentioned above, [P2504R0] shows that one can soundly use it to solve all possible concurrency problems.

If one is able to use senders to describe any chunk of concurrent work (with one entry and one exit point), and if we can decompose any concurrent programs in such work chunks, then we are covered.

The reader should note that, out of the box, P2300 doesn't provide primitives for representing repetitive computations or for type-erasing the senders; these are needed to be able to describe many concurrent chunks of work. However, the model is general enough to allow the user to create abstractions for these. In fact, there are libraries out there that already support these abstractions; one of the most known ones is [libunifex].

Bonus: coroutines

I felt it would be better not to include coroutines as one of the requirements needed for structured concurrency. We can write good concurrent code even without the use of coroutines. However, coroutines can help in writing good concurrent (and parallel) code.

Interestingly enough, in his part of the Structured Programming book [Dahl72], Dahl discusses coroutines, a SIMULA feature, as a way of organising programs. If a program uses only subroutines, then, at the function-call level, the program is always organised hierarchically. Using coroutines might help when strict hierarchical organisation is not necessarily needed, and cooperative behaviour is more important.

The senders/receivers proposal guarantees good interoperability between senders and coroutines. According to the proposal, all awaitables are also senders (i.e., can be used in places where senders are used). Moreover, most of the senders (i.e., senders that yield a single value) can be easily made awaitables (by a simple annotation in the coroutine type).

Let us look at the example from Listing 4[3]. Here, **`task<T>`** is a coroutine type. The reference implementation associated with the P2300 paper provides an implementation for this abstraction [P2300RefImpl].

3. This is a bad implementation of a Fibonacci function, for multiple reasons. We use it here just to exemplify the interaction between components, not to showcase how one would implement a parallel version of Fibonacci

The senders/receivers model

The senders/receivers model proposed to C++ share many traits with the async/await model [Wikipedia]. The latter model was introduced in F# 2.0 in 2007 [Syme11], and then spread to multiple programming languages (C#, Haskell, Python, etc.), eventually reaching C++ with the addition of coroutines in C++20. In F#, a type **`Async<T>`** represents an asynchronous computation (wording taken from [Syme11]). This would be analogous to the **`Task<T>`** coroutine type we discussed above. And, as senders are equivalent to these types, it means that the **`Async<T>`** from F# is an abstraction similar to a sender.

However, the senders/receivers model can be implemented more efficiently than coroutines or other similar type-erased abstractions.

Structured Concurrency is not something that can be done with senders/receivers only in C++. Other languages can support Structured Concurrency too; the P2300 senders/receivers model just ensures that we get the most performance out of it, as it avoids type-erasure and other performance penalties.

`naive_par_fib` is a coroutine; it uses **`co_return`** twice and **`co_await`** once. The res variable inside the coroutine is a sender. We can co_await this sender, as shown in the body of our coroutine; thus, senders can be awaited. Towards the end of the example, we show how one can simply use the coroutine invocation to chain it to a sender, thus using it just as if it was a sender. Furthermore, the function passed to **`let_value`** needs to return a sender; we are returning a **`task<uint64_t>`** awaitable.

Using coroutines might make the user code easier to read. In this particular example, we also showed how coroutines can be used to obtain type-erasure of senders; **`task<T>`** acts like a type-erased sender that generates a value of type **`T`**.

Discussion

The current senders/receivers proposal, [P2300R4] doesn't aim at being a comprehensive framework for concurrent programming. For example, it lacks facilities for the following:

- type-erased senders
- a coroutine **`task`** type (the name is misleading, as it would encapsulate a *computation*)
- facilities for repeating computations
- facilities for enabled stream processing (reactive programming)
- serialiser-like abstractions
- etc.

```
template <ex::scheduler S>
task<uint64_t> naive_par_fib(S& sched, int n) {
  if (n <= 2)
    co_return 1;
  else {
    ex::sender auto start = ex::schedule(sched);
    ex::sender auto res = ex::when_all(
      start | ex::let_value([&] {
                return naive_par_fib(sched, n - 1);
              }),
      start | ex::let_value([&] {
                return naive_par_fib(sched, n - 2);
              })
    ) | ex::then(std::plus<uint64_t>());
    co_return co_await std::move(res);
  }
}
ex::sender auto snd =
    naive_par_fib(sched, n)
    | ex::then([](uint64_t res) {
        std::cout << res << "\n";
      });
std::this_thread::sync_wait(std::move(snd));
```

Listing 4

However, the most important part of the proposal is the definition of the general framework needed to express concurrency. Then, all the facilities needed to represent all kinds of concurrent programs can be built on top of this framework. The proposal contains just a few such facilities (sender algorithms), but the users can create their own abstractions on top of this framework.

This article focuses on the framework, showing that it has the needed requirements for Structured Concurrency; we are not arguing that the facilities described in P2300 are enough.

Another important topic to discuss, especially in conjunction with concurrency, is performance. After all, the move towards more parallelism is driven by the performance needs of the applications. Our single-threaded processors are not fast enough, so we add more parallel processing to speed things up, and concurrency is needed to ensure the safety of the entire system.

In task-based systems, we put all the work into tasks. Usually, we type-erase these tasks to be able to pass them around. That means that we have an abstraction cost at the task level. This will make it impractical to use tasks for very small chunks of work. But, it is typically also impractical to use tasks with very big chunks of works, as this can lead to situations in which we do not properly fill up all the available cores. Thus, task-based systems have good performance in just a relatively narrow range, and users cannot fully control all performance aspects. Task-based systems can have good performance, but not always the best performance.

In contrast, the senders/receivers model doesn't imply any abstraction that might incur performance costs. The user is free to introduce type-erasure at any level that is best for the problem at hand. One can write large concurrent chunks of work without needing to pay the costs of type-erasure or memory allocation. Actually, most facilities provided in the current P2300 proposal do not incur any additional type-erasure or memory allocation costs.

The senders/receivers model allows one to model concurrency problems without any performance penalties.

Conclusions

Structured Programming produced a revolution for the good in the software industry. Sadly, that revolution did not also happen in concurrent programming. To improve concurrent programming, we want to apply the core ideas of Structured Programming to concurrency.

After discussing the main ideas that shape Structured Programming, we briefly discussed that classic concurrent programming is not structured at all. We've shown that task-based programming is better, but still doesn't meet all the goals we set to enable Structured Concurrency.

The rest of this article showed how the senders/receivers model fully supports all the important ideas from Structured Programming, thus making concurrent programming with this model worthy to be called structured.

In concurrent programming, we have *computations* to play the role of functions in structured programming, and any *computation* can be described by a sender. With the help of [P2504R0], we argued that one can use senders to describe any concurrent program. We discussed here how senders can be used as abstractions to describe concurrent work at any level (from the entire program to small chunks of work). We've shown that this model lends itself well to recursive decomposition of concurrent programs and to local reasoning of abstractions. Likewise, we've also discussed how senders have the single entry, single exit shape and how one can build complex structures from simple primitives.

All these make the senders/receivers model match the criteria we set to achieve Structured Concurrency. When this proposal gets into the C++ standard, we will finally have a model to write safe, efficient and structured concurrent programs.

References

[Böhm66] Corrado Böhm, Giuseppe Jacopini, 'Flow Diagrams, Turing Machines and Languages With only Two Formation Rules', http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.9119&rep=rep1&type=pdf, *Communication of the ACM*, May, 1966

[Brooks95] Frederick P. Brooks Jr., *The Mythical Man-Month* (anniversary ed.)., Addison-Wesley Longman Publishing, 1995

[Dahl72] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*, Academic Press Ltd., 1972

[Dijkstra65] Edgar Dijkstra, 'Solution of a problem in concurrent programming control', *Communications of the ACM*, September, 1965.

[Dijkstra68] Edgar Dijkstra, 1Go To Considered Harmful', https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf, Communication of the ACM, March, 1968

[libunifex] Eric Niebler, et al., libunifex, https://github.com/facebookexperimental/libunifex, 2022

[Naur69] Peter Naur, Brian Randell, *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October, 1968*, 1969

[Niebler21a] Eric Niebler, 'Working with Asynchrony Generically: A Tour of C++ Executors (part 1/2)', *CppCon*, 2021, https://www.youtube.com/watch?v=xLboNIf7BTg

[Niebler21b] Eric Niebler, 'Working with Asynchrony Generically: A Tour of C++ Executors (part 2/2)', *CppCon*, 2021, https://www.youtube.com/watch?v=6a0zzUBUNW4

[P2300R4] Michał Dominiak et al., '`std::execution`', 2022, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2300r4.html

[P2300RefImpl] Bryce Lelbach, et al., 'P2300 Reference implementation', 2022, https://github.com/brycelelbach/wg21_p2300_std_execution

[P2504R0] Lucian Radu Teodorescu, 'Computations as a global solution to concurrency', 2021, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2504r0.html

[Syme11] Don Syme, Tomas Petricek, Dmitry Lomov. 'The F# asynchronous programming model', *International Symposium on Practical Aspects of Declarative Languages*, Springer, 2011

[Teodorescu20] Lucian Radu Teodorescu, 'The Global Lockdown of Locks', *Overload* 158, August 2020, available online at https://accu.org/journals/overload/28/158/teodorescu/

[Teodorescu21] Lucian Radu Teodorescu, 'Composition and Decomposition of Task Systems', *Overload* 162, April 2021, available online at https://accu.org/journals/overload/29/162/teodorescu/

[Wikipedia] 'Async/await' on Wikipedia, https://en.wikipedia.org/wiki/Async/await, 2022

The Vector Refactored

Finding the right level of abstraction can be challenging. Teedy Deigh razes the level of abstraction.

@Polymorpheus: What if I told you that **`vector`** is not the only data structure available from the standard library?

@Teedy: But what if I told you, Polly, that the C++ standard says that "**`vector`** is the type of sequence that should be used by default"? As any engineer knows, don't mess with the defaults, amiright?

[Teedy looks for support from the others on the call. She has Zoom in Muppet Show mode, but all she gets is a wall of muted silence against a back drop of either camera-off darkness or Star Wars backdrops and devs who are obviously 'multitasking'.]

@Polymorpheus: I think it's reasonable to use **`string`** for strings rather than **`vector<char>`**, don't you?

@smith: And also **`u16string`** and **`u32string`** – but please, please, don't use **`wstring`**.

@Polymorpheus: Yes, thank you, Winston. For a relatively new codebase, this system does seem a little stuck in the last century. Either way, **`basic_string`** is the template you're looking for.

@Teedy: I can roll with that – still an array under the hood, which we know is the one true data structure.

@Polymorpheus: I appreciate the fundamental nature of arrays, but it feels a little low level for some of the code, such as – wait a moment, let me just share my screen *[code appears]* – this code here, which queues requests.

@Teedy: Oh yes, love that code.

@smith: Love is blind.

@Polymorpheus: I think a **`deque`** would have been just fine.

@Teedy: Pfft, **`deque`**'s just a **`vector`** wannabe that lacks cache locality. It's all over the place. This variant of a circular buffer does everything needed, and with the kind of code that keeps me in work.

@smith: Wouldn't it be better to encapsulate all this *[waves hand at camera]* in a class rather than leaving the logic strewn all over the code, big footprints of duplicated control flow everywhere the **`vector`** – sorry, circular buffer – passes? There's so much copy and pasting *[cat walks across keyboard in front of camera, a stream of garbled characters fill the chat]* I get a haunting sense of déjà vu every time I look through this code.

@Teedy: Repetition legitimises.

@smith: ...

@Teedy: What?

@smith: That, Ms Deigh, is the sound of incredulity.

@Polymorpheus: Winston has a point, though, Teedy. I think for a system that is I/O bound – spending most of its time in the OS, the network, setting cookies and chatting to the Oracle database – this micro-optimisation doesn't justify the code complexity it brings.

I presume low-level caching is also why you steer clear of **`map`** in – wait a moment – this code?

@Teedy: Strictly speaking, that is a **`map`**... it's just that it's implemented as a **`vector`**. I mean, really, that's all a **`map`** is, isn't it? It's a binary tree that wants to be a sorted array. I just chopped it down to size.

@smith: I think you should be trying to raise the level of abstraction rather than raze it to the ground.

@Teedy: Look, all those other data structures are just simulating certain algorithms on a **`vector`**. Rather than live in a simulation, I prefer to keep it real.

@Polymorpheus: How do you define 'real'? At one level, these are all simply electrical signals. We need to work at a higher level than that.

@Teedy: This talk of higher-level stuff is all very well, but performance has always been an issue with this system – it's like watching everything in slow motion. I'm doing my bit to make my code as cache friendly as possible, given how cash unfriendly some of our customer response has been.

And what about thread safety, atomic operations and low-lock code? A lot easier with **`vector`** than any data structure based on linking together fragmented memory. I could go on.

@smith: I don't doubt it.

Well, rather than let Teedy rabbit on, I think we should address the elephant in the Zoom: the architecture. Teedy's micro-optimisations are infuriating from a coding perspective, but her desire to optimise is understandable given what she's working with and around.

@Teedy: Thanks... I think.

@smith: I mean, this code shouldn't even be multi-threaded. A thread per request? That's so 1990s. I'm not even convinced we should be using Oracle.

Teedy Deigh noodles with code and codes with noodles. She enjoys unlimited coffee refills and avoiding long walks... and short walks... and, in fact, any kind of walk whatsoever. She has no love of the outdoors and believes that doors are there for a reason. For her star sign, she identifies more with -> than *. She has no pets, and the only **`cat`** she likes is on the command line.

Do not try and bend the architect.
That's impossible.
Instead, only try to realise the truth.

@Teedy: Agreed. With hindsight, NoSQL might be the better option.

@Polymorphism: You're SWOMming.

@Teedy: What?

@Polymorphism: Not you, the architect. I think he's been trying to say something, but has been speaking while on mute.

@TheArchitect: Ah, yes, umm... as I have been saying for the last few minutes, I planned the architecture up front and in meticulous detail. I sent the spec round a few months ago.

@smith: I think that ended up in my spam folder.

@Teedy: I got it. It inspired me to write that limerick.

@smith: Oh yes, that was actually quite good.

@Teedy: Thanks... I think.

@smith: How did it go?

@Teedy: UML, UML, UML,
Bloody Hell, Bloody Hell, Bloody Hell,
UML, Bloody Hell,
Bloody Hell, UML,
UML, UML, Bloody Hell.

[Thumbs-up and applause emojis appear next to some of the callers.]

@TheArchitect: Anyway, this is the sixth time we have rewritten this system, and I am convinced it is going to work out well this time. There is no need to change anything.

@smith: You mean, there's no need to change anything from all the times it didn't work out?

@Polymorpheus: We're a bit more agile these days. We run retrospectives to avoid repeating the mistakes of the past.

@TheArchitect: Repetition legitimises.

@smith: Dammit, not everyone believes what you believe.

@TheArchitect: My beliefs do not require them to.

@Polymorpheus: I think it would help us and the system if we understood the rationale behind some of your more, errm, interesting and dated–

@TheArchitect: –timeless–

@Polymorpheus: –decisions. I'm not sure we're ready to just swallow the take-it-as-read pill.

@TheArchitect: Some of my answers you will understand, and some of them you will not.

@Teedy: Always riddles with you, huh?

@smith: Do not try and bend the architect. That's impossible. Instead, only try to realise the truth.

@Teedy: What truth?

[@TheArchitect disappears.]

@Polymorpheus: There is no architect. One of the advantages of being the meeting host.

[Thumbs-up and applause emojis appear next to some of the callers.]

@smith: Yeah, he'll probably think it was a bad connection or blame it on Zoom.

@Teedy: OK, so now we can redesign the system, right?

@smith: We should aim for something more stateless, functional and habitable. For storing options and other data, I'd like to move away from XML... all those bloody, pointy angle brackets – whoever said XML was human readable clearly had a very specific human in mind. Perhaps we could bridge the code–data divide and opt for something like LISP? It's like wiping your config with silk.

@Teedy: I know Scheme.

@Polymorpheus: Show me.