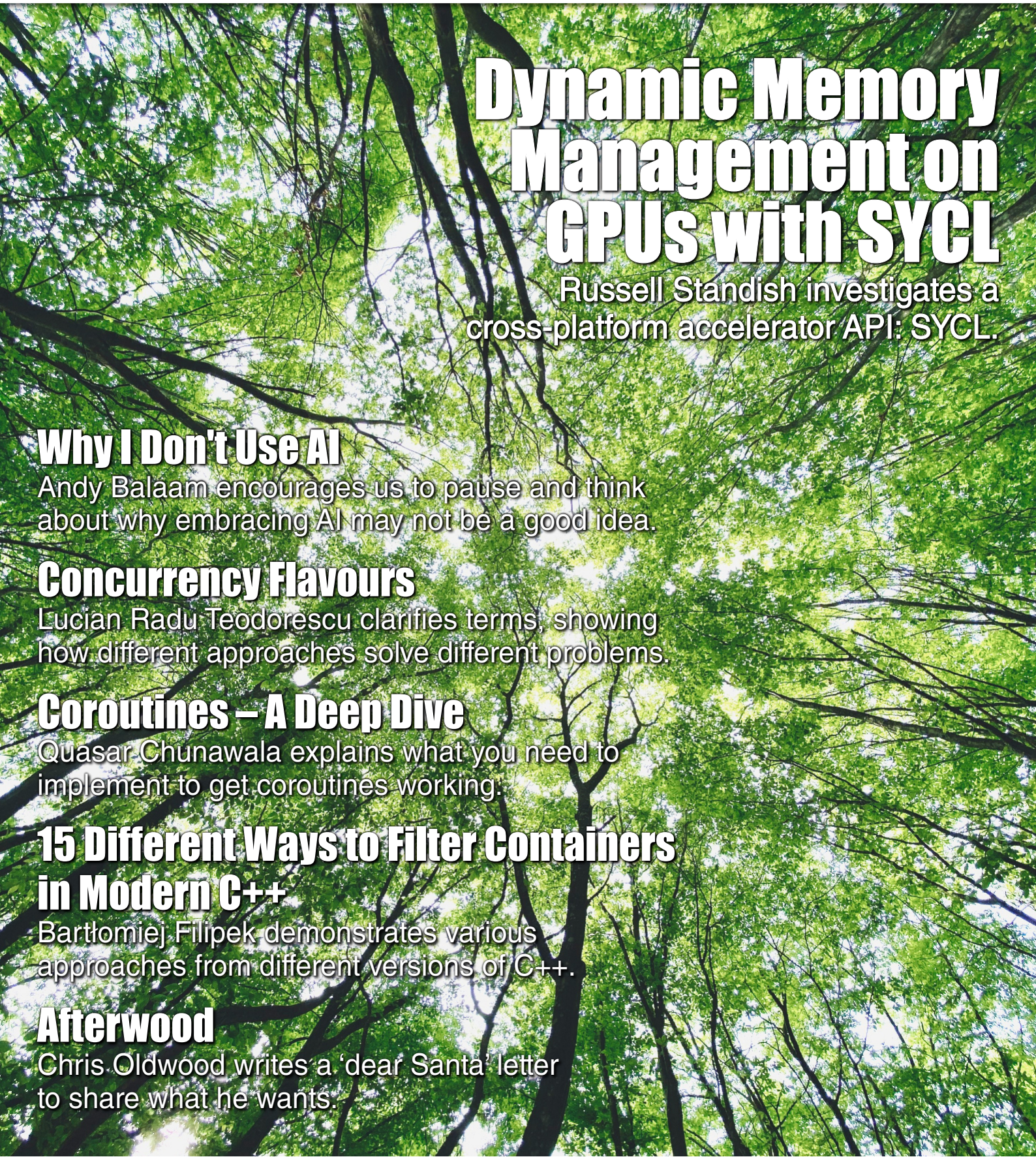
overload 190

DECEMBER 2025 £4.50



Dynamic Memory Management on GPUs with SYCL

Russell Standish investigates a cross-platform accelerator API: SYCL.

Why I Don't Use AI

Andy Balaam encourages us to pause and think about why embracing AI may not be a good idea.

Concurrency Flavours

Lucian Radu Teodorescu clarifies terms, showing how different approaches solve different problems.

Coroutines – A Deep Dive

Quasar Chunawala explains what you need to implement to get coroutines working.

15 Different Ways to Filter Containers in Modern C++

Bartłomiej Filipek demonstrates various approaches from different versions of C++.

Afterwood

Chris Oldwood writes a 'dear Santa' letter to share what he wants.

A magazine of ACCU

ISSN: 1354-3172

December 2025
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Teodor Drobota on Unsplash.

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

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

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

Copy deadlines
All articles intended for publication in *Overload* 191 should be submitted by 1st January 2026 and those for *Overload* 192 by 1st March 2026.

A Guest Editorial

C++20 introduced coroutines. Quasar Chunawala, our guest editor for this edition, gives an overview.

You've likely heard about this new C++20 feature, **coroutines**. I think that this is a really important subject and there are several cool use-cases for coroutines. A **coroutine** in the simplest terms is just a function that you can pause in the middle. At a later point the caller will decide to resume the execution of the function right where you left off. Unlike a function therefore, coroutines are always stateful - you atleast need to remember where you left off in the function body.

Coroutines can simplify our code! Coroutines are a great tool, when it comes to implementing parsers.

The coroutine return type

The initial call to the coroutine function will produce this return object of a certain `ReturnType` and hand it back to the caller. The interface of this type is what is going to determine what the coroutine is capable of. Since coroutines are super-flexible, we can do a whole lot with this return object. If you have some coroutine, and you want to understand what it's doing, the first thing you should look at is the `ReturnType`, and what it's interface is. The important thing here is, we design this `ReturnType`. If you are writing a coroutine, you can decide what goes into this interface.

How to turn a function into a coroutine?

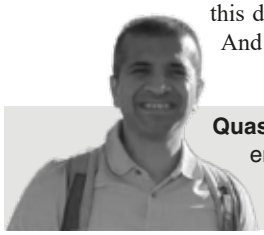
The compiler looks for one of the three keywords in the implementation: `co_yield`, `co_await` and `co_return`.

Keyword	Action	State
`co_yield`	Output	Suspended
`co_return`	Output	Ended
`co_await`	Input	Suspended

In the preceding table, we see that after `co_yield` and `co_await`, the coroutine suspends itself and after `co_return`, it is terminated (`co_return` is the equivalent of the `return` statement in the C++ function).

Use-cases for coroutines

Asynchronous computation. Suppose we are tasked with designing a simple echo server. We listen for incoming data from a client socket and we simply send it back to the client. At some point in our code for the echo server, we will have a piece of logic like that in Listing 1: we certainly don't want to write a server like this. Say one of the clients requests communication and we are in a session. They say, they are ready to send the data, so we are blocking on the `read`, but maybe they send us this data in 2 minutes, or 5 minutes or even more. And other clients keep waiting.



Introducing Quasar Chunawala – our Guest Editor

It's about time we had an editorial, so I have handed over to Quasar Chunawala for this issue. He has written about about coroutines in C++, giving an introduction here and a deeper dive later in this issue. Coroutines can seem hard at first sight, but if someone walks you through the steps they aren't really that hard – they are just able to be configured in various ways. They can simplify code, for example writing a state machine doesn't need a massive switch to handle different states.

Thanks to Quasar for volunteering. If you would like a go one day, please get in touch with me. You'll get a preview of our articles and a chance to comment before publication, and can share something you are interested in, too.

```
void session(Socket sock){
  char buffer[1024];
  int len = sock.read({buffer});
  sock.write({buffer,len});
  log(buffer);
}
```

Listing 1

One solution is to use an asynchronous framework and rewrite our code as in Listing 2 (on next page).

So, the `session` makes two associations:

On finishing read ↦ `on_finished_read_callback`

On finishing write ↦ `done`

And implicitly there is a third association even though we cannot see it here – this entire function `session` is most likely a callback, in response to an event like *On client connection established*.

Accepting a new client connection ↦ `session`

So, the server will be many different associations of events to callbacks at different levels.

Pay attention to the `state`. We said that, we wanted to allocate it on the heap and manage it through a `shared_ptr`. We pass this `shared_ptr<State>` by value to every single callback. This way, I make sure that the last one who touches this session turns off the lights and deallocates `state`.

While this is a toy-example, in real production code, there can be a long sequence of steps and calling lambdas inside lambdas can obfuscate the meaning of the code.

Quasar Chunawala has a bachelor's degree in Computer Science. He is a software programmer turned quant engineer, enjoys building things ground-up and is deeply passionate about programming in C++, Rust and concurrency and performance-related topics. He is a long-distance hiker and his favorite trekking routes are the Goechala route in Sikkim, India and the Tour-Du-Mont Blanc (TMB) circuit in the French Alps. Contact at quasar.chunawala@gmail.com.

```
void session(Socket sock){
  struct State{ Socket sock; char buffer[1024];
};
// Heap allocate the state
auto state = std::make_shared<State>(sock,
  buffer);
auto on_read_finished_callback = [state](
  error_code ec,
  size_t len
)
{
  auto done = [state](error_code ec, size_t len)
  {
    if(!ec)
      log();
    }
    if(!ec)
    {
      // Perform an asynchronous write
      state->socket.async_write(
        state->buffer,
        done
      );
    }
}
// Perform an asynchronous read
state->socket.async_read( state->buffer,
  on_read_finished_callback );
}
```

Listing 2

A coroutine implementation of the same echoing session would look like this:

```
Task<void> session(Socket sock){
  char buffer[1024];
  int len = co_await sock.async_read({buffer});
  co_await sock.async_write({buffer,len});
  log(buffer);
}
```

This looks very similar to the sequential code, except that we use this `co_await` keyword. You have clear indication of the points where the coroutine will be suspended. Also, note that previously the function `session` returned `void`. Now, we are returning something: a `Task<void>`. This will be a handle to the coroutine and it's how the outside world will be communicating with the coroutine.

Suspended computation. A second use-case is that coroutines support lazy evaluation. Lazy evaluation doesn't do any work unless it's absolutely necessary. This can also potentially make your code more efficient. Lazy evaluation also supports programming with infinite lists.

The lay of the land

The diagram in Figure 1 shows the relationships between the components of coroutines [Weis2022].

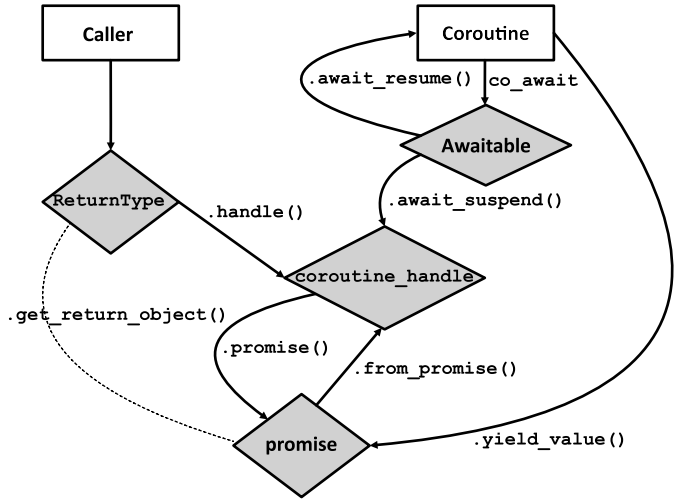


Figure 1

`ReturnType` – The initial call to the coroutine returns an object of the type `ReturnType`. This interface determines what the coroutine is actually capable of.

Promise – If we think of the coroutine as the producer of data and the caller as the consumer, on the producer side, the coroutine will store the result in a promise object. On the consumer side, the caller can retrieve the result using the return object of type `ReturnType`. So, the promise is the interface through which the caller interacts with the coroutine.

`coroutine_handle` – The `coroutine_handle` is like a raw pointer to the coroutine frame. The coroutine frame consists of the values of local variables, the promise and any internal state.

Awaitable – A coroutine may send – aka yield – a value to the caller or may (co)await an asynchronous operation to complete. In both cases, the coroutine will suspend itself, save its state in the coroutine frame and return control to the caller. An awaitable is therefore thing the coroutine awaits on. ■

Reference

[Weis2022] Andreas Weis, 'Deciphering Coroutines', *CppCon 2022* , available at https://www.youtube.com/watch?v=JXZswq3m41I.

Why I Don't Use AI

Many people are embracing GenAI. Andy Balaam encourages us to pause and think about why this might not be a good idea.

I choose to avoid using 'AI' (by which I mean Large Language Models [Wikipedia]).

Here's why:

- they have a devastating environmental impact,
- they are trained by exploiting and traumatising millions of low-paid workers,
- they produce biased and dangerously incorrect results, and
- they unfairly use people's creative work.

Environmental impact

Between 2010 and 2020, the energy used by data centres around the world rose only slightly [Knight20], but since then energy use has risen sharply [Kearney24], driven by the expansion of AI [O'Donnell25]. Compounding the problem, because these data centres are using more energy than was predicted or provided for by existing generation, the carbon intensity of the electricity use is much higher than average (48% higher in the US according to one study [O'Donnell25]).

Driven by AI, data centres are predicted to double their energy use by 2030 [Chen25]. In Ireland in 2025, data centres use almost a fifth of the electricity supply [Campbell23], despite the rise in use of electric vehicles over recent years.

Unless we change course, this is not going to slow down or become sustainable. Quoting Sam Altman [Roytburg25]:

> You should expect OpenAI to spend trillions of dollars on data center construction in the not very distant future.

This rapacious appetite for more data and more computation is hard-wired into the AI movement. It is driven by a belief that Sam Altman expresses like this on his blog: "*it appears that you can spend arbitrary amounts of money and get continuous and predictable gains*" [Altman25]. The movement is built on the idea that if we just consume more and more resources, we will achieve greater and greater success. As long as they are driven by this belief, we can never expect them even to attempt to curb their energy use.

Data centres are often harmful to the local area, and are often sited in areas of existing social deprivation. They consume both energy and water that could otherwise be used by people, and cause problems with pollution and energy shortages [Fleury25].

For more detail on the environmental impacts of AI, I recommend (perhaps surprisingly) the *Teen Vogue* article 'ChatGPT Is Everywhere – Why Aren't We Talking About Its Environmental Costs?'[McMenamin25].

Andy Balaam Andy is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his open source projects at artificialworlds.net or you can contact him at andybalaam@artificialworlds.net

Exploitation of workers

The AI companies don't like to talk about it, but their models only work when provided with vast amounts of human-created data. This data is not simply passively scraped from the Internet: the models are built on the work of millions of people actively classifying images and rating answers, shaping the models to produce results that look and sound safe and reasonable [Williams22].

Most of the people involved are very poorly paid [Rowe23]. Many of them are traumatised by horrific images and speech that they are asked to classify [Stahl25].

Workers paid between $1.32 and $2 per hour in Kenya (a wage described as 'an insult') talk about their work like this [Bartholomew23]:

> You're reading this content, day in, day out, over the course of days and weeks and months, it seeps into your brain and you can't get rid of it.

Biased and dangerous results

Despite wide acknowledgement among experts that AI produces unreliable results, many people are being encouraged to trust its output in terms of accuracy and safety.

Researchers have found that recent AI models confidently express judgements that are plain wrong, making mistakes about basic economic ideas like interest rates [Smith24], or inventing conversations about patients' medical data [Burke24].

Even more concerningly, people are treating AI models as trustworthy conversation partners. This is done with full encouragement from the AI companies, despite the real risks involved. In 2023, Character AI founder Noam Shazeer said of AI, "It's going to be super, super helpful to a lot of people who are lonely or depressed." In fact, one of Character AI's chat bots played an alarming role in the suicide of a teenager [Roose24]. The parents of another teenager say it explicitly encouraged him to commit suicide before he did [Hill25a]. There is a growing number of reports of chat bots guiding people down "delusional spirals" that can have devastating mental health consequences [Hill25b].

It is clear from all these examples and many more court cases currently in progress that it is impossible to control the words spewing from these models. Given the racial slurs included in the most widely-used training dataset [McQuaid21] it is not surprising that they occasionally lose the plot like Grok once did, producing racist rants and naming itself 'MechaHitler' [Hagen25].

Throughout all of this unreliability, the popular AI models very reliably convey total confidence in the latest answer they gave, even where it contradicts the previous one.

It is pure wishful thinking to say that AI models can replace human judgement in any area. If people treat AI as a trustworthy oracle or a trustworthy companion, this wishful thinking is actively harmful [Schneier23].

leading models are trained on all the data the AI companies can get their hands on, regardless of licence ... news and information sites, art galleries and personal web sites

Unfair use of creative work

The leading models are trained on all the data the AI companies can get their hands on, regardless of licence. This includes proprietary information such as news and information sites, art galleries and personal web sites that are protected by traditional copyright arrangements. These sites (and indeed printed books and other offline materials) are published under a legal framework that allows searching and indexed of their content without reproducing it. Many commercial web sites depend on visitors being directed towards their web site so they can receive advertising income.

Meanwhile, a huge amount of material is available online in free and open source form, especially but not exclusively the enormous corpus of source code that is used to train AI coding models. The bargain for this material is different: authors require attribution for re-use, and place additional requirements such as 'share-alike' clauses that enforce the release of derived works under the same terms. AI models are breaking this legally-enforced bargain by reproducing derived or straightforwardly copied works with no attribution or correct license.

Directing visitors to web sites is not a benevolent or coincidental side effect of search engines: it is a self-sustaining bargain: you allow me to index your content and in exchange I direct users to your site. If this bargain breaks down, web site creators lose their source of income and many web sites will disappear [Stokes25]. Complying with license terms is not optional: it is required to use any material, including free and open source content.

These bargains are enforced by copyright law. The invention of AI did not change anything about this bargain, except that it obfuscated the copying of copyrighted material [Carson25] [Gerken23], and convinced governments that enforcing the law would block the promised economic miracle of AI [Milmo25a] [Milmo25b].

Other reasons

This article is primarily concerned with ethical reasons for avoiding AI usage, but there are plenty of other reasons too:

- They don't do what the billionaires claim they do: researchers found that open source developers were 19% less productive when using AI tools [Becker25], and a UK study showed no productivity gain [Kunert25].
- They make you worse at your job: doctors rapidly experienced erosion of their ability to spot cancer when they began using AI tools [Black25].
- They will be used as an excuse to cut jobs: Amazon recently told its workers to expect job cuts due to AI [Roth25] even though it has been engaging in regular job cuts since 2022.

Despite the hype, it is clear that AI can perform some tasks effectively, for example making very convincing fake videos. I choose to avoid them where I can, for the above reasons.

Conclusion

I believe that AI is a force that is doing real harm in our world, and is concentrating wealth and power in the hands of those who are already wealthy and powerful enough. If you agree, let's work together as professionals to help our companies, organisation and friends to be skeptical of its benefits, and mindful of its problems when we make decisions about how and where to use it.

If you'd like to hear more AI-skeptical viewpoints, thecon.ai [Bender25] is a good place to start. The article 'I Am An AI Hater' [Moser25] by Anthony Moser was the research starting point for this article and is recommended if you'd like a less emotionally constrained view along similar lines. ■

References

(With thanks to the *Overload* reviewers for suggesting several extra references.)

[Altman25] Sam Altman 'Three Observations', Sam Altman's blog, posted 9 February 2025 at https://blog.samaltman.com/three-observations

[Bartholomew23] Jem Bartholomew, 'Q&A: Uncovering the labor exploitation that powers AI', *Columbia Journalism Review*, published 9 August 2023 at https://www.cjr.org/tow_center/qa-uncovering-the-labor-exploitation-that-powers-ai.php

[Becker25] Joel Becker, Nate Rush, Elizabeth Barnes and David Rein, '[Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity', *METR*, published 10 July 2025 at https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/

[Bender25] Emily M. Bender and Alex Hanna (2025), *The AI Con: How to fight big tech's hype and create the future we want*, published by Harper, ISBN: 978-0063418561 (see also thecon.ai)

[Black25] 'AI Eroded Doctors' Ability to Spot Cancer Within Months in Study', *Bloomberg*, published 12 August 2025 at https://www.bloomberg.com/news/articles/2025-08-12/ai-eroded-doctors-ability-to-spot-cancer-within-months-in-study

[Burke24] Garance Burke and Hilke Schellmann, 'Researchers say an AI-powered transcription tool used in hospitals invents things no one ever said' *AP News*, published 26 October 2024 at https://apnews.com/article/ai-artificial-intelligence-health-business-90020cdf5fa16c79ca2e5b6c4c9bbb14

[Campbell23] John Campbell, 'Data centres use almost a fifth of Irish electricity', BBC, published 12 June 2023 at https://www.bbc.co.uk/news/articles/cpe9l5ke5jvo

[Carson25] David Carson, 'Theft is not fair use', published 21 April 2025 at https://jskfellows.stanford.edu/theft-is-not-fair-use-474e11f0d063

[Chen25] Sophia Chen, 'Data centres will use twice as much energy by 2030 – driven by AI', *Nature*, published 10 April 2025 at https://www.nature.com/articles/d41586-025-01113-z

[Fleury25] Michelle Fleury and Nathalie Jimenez, '"I can't drink the water" – life next to a US data centre', BBC, published 10 July 2025 at https://www.bbc.co.uk/news/articles/cy8gy7lv448o

[Gerken23] Tom Gerken, 'New York Times sues Microsoft and OpenAI for 'billions'', BBC, published 27 December 2023 at https://www.bbc.co.uk/news/technology-67826601

[Hagen25] Lisa Hagan, Huo Jingman 'Elon Musk's AI chatbot, Grok, started calling itself 'MechaHitler'', NPR, published 9 July 2025 at https://www.npr.org/2025/07/09/nx-s1-5462609/grok-elon-musk-antisemitic-racist-content

[Hill25a] Kashmir Hill, 'A Teen Was Suicidal. ChatGPT Was the Friend He Confided In' *New York Times*, updated 27 August 2025 at https://www.nytimes.com/2025/08/26/technology/chatgpt-openai-suicide.html

[Hill25b] Kashmir Hill and Dylan Freedman, 'Chatbots Can Go Into a Delusional Spiral. Here's How It Happens', *New York Times*, published 8 August 2025 at https://www.nytimes.com/2025/08/08/technology/ai-chatbots-delusions-chatgpt.html?ref=platformer.news

[Kearney24] Laila Kearney 'US data-center power use could nearly triple by 2028, DOE-backed report says', *Reuters*, published 20 December 2024 at https://www.reuters.com/business/energy/us-data-center-power-use-could-nearly-triple-by-2028-doe-backed-report-says-2024-12-20/)

[Knight20] Will Knight, 'Data Centers Aren't Devouring the Planet's Electricity – Yet', *Wired*, published 27 February 2020 at https://www.wired.com/story/data-centers-not-devouring-planet-electricity-yet/

[Kunert25] Paul Kunert, 'UK government trial of M365 Copilot finds no clear productivity boost', *The Register*, published 4 September 2025 at https://www.theregister.com/2025/09/04/m365_copilot_uk_government/

[McMenamin25] Lex McMenamin, 'ChatGPT Is Everywhere – Why Aren't We Talking About Its Environmental Costs?' in *Teen Vogue*, published 7 May 2025 at https://www.teenvogue.com/story/chatgpt-is-everywhere-environmental-costs-oped

[McQuaid21] John McQuaid, 'Limits to Growth: Can AI's Voracious Appetite for Data Be Tamed?' *Undark,* published 18 October 2021 at https://undark.org/2021/10/18/computer-scientists-try-to-sidestep-ai-data-dilemma/

[Milmo25a] Dan Milmo, 'UK copyright law consultation 'fixed' in favour of AI firms, peer says', *The Guardian*, published 11 February 2025 at https://www.theguardian.com/technology/2025/feb/11/uk-copyright-law-consultation-fixed-favour-ai-firms-peer-says)

[Milmo25b] Dan Milmo and Robert Booth, 'UK proposes letting tech firms use copyrighted work to train AI', *The Guardian*, published 17 December 2024 at https://www.theguardian.com/technology/2024/dec/17/uk-proposes-letting-tech-firms-use-copyrighted-work-to-train-ai)

[Moser25] Anthony Moser, 'I Am An AI Hater', published 26 August 2025 at https://anthonymoser.github.io/writing/ai/haterdom/2025/08/26/i-am-an-ai-hater.html

[O'Donnell25] James O'Donnell and Casey Crownhart, 'We did the math on AI's energy footprint. Here's the story you haven't heard', *MIT Technology Review*, published 20 May 2025 at https://www.technologyreview.com/2025/05/20/1116327/ai-energy-usage-climate-footprint-big-tech/

[Roose24] Kevin Roose, 'Can A.I. Be Blamed for a Teen's Suicide?', *New York Times*, published 23 October 2024 at https://www.nytimes.com/2024/10/23/technology/characterai-lawsuit-teen-suicide.html (subscription required)

[Roth25] Emma Roth, 'Amazon CEO says it will cut jobs due to AI's 'efficiency'', *The Verge*, published 17 June 2025 at https://www.theverge.com/news/688679/amazon-ceo-andy-jassy-ai-efficiency

[Rowe23] Niamh Rowe, 'Millions of Workers Are Training AI Models for Pennies' *Wired,* published 16 October 2023 at https://www.wired.com/story/millions-of-workers-are-training-ai-models-for-pennies/

[Roytburg25] Eva Roytburg, 'Sam Altman admits OpenAI 'totally screwed up' its GPT-5 launch and says the company will spend trillions of dollars on data centers' *Fortune*, published 18 August 2025 at https://fortune.com/2025/08/18/sam-altman-openai-chatgpt5-launch-data-centers-investments/

[Schneier23] Bruce Schneier, 'AI and Trust', published 4 December 2023 at https://www.schneier.com/blog/archives/2023/12/ai-and-trust.html)

[Smith24] Gary Smith, 'Large Language Models Are Often Wrong, Never in Doubt', *MindMatters*, published 29 April 2024 at https://mindmatters.ai/2024/04/large-language-models-are-often-wrong-never-in-doubt/

[Stahl25] Lesley Stahl, 'Labelers training AI say they're overworked, underpaid and exploited by big American tech companies' *CBS News*, updated 29 June 2025 at https://www.cbsnews.com/news/labelers-training-ai-say-theyre-overworked-underpaid-and-exploited-60-minutes-transcript/

[Stokes25] Mark Stokes, 'The AI Starvation Loop', *Medium*, published 29 July 2025 at https://medium.com/@mark_stokes/the-ai-starvation-loop-how-ai-is-starving-the-web-and-what-we-can-do-about-it-e0e567f13ad4)

[Wikipedia] 'Large language model' *Wikipedia*, https://en.wikipedia.org/wiki/Large_language_model

[Williams22] Adrienne Williams, Milagros Miceli and Timnit Gebru 'The Exploited Labor Behind Artificial Intelligence', *Noema*, published 13 October 2022 at https://www.noemamag.com/the-exploited-labor-behind-artificial-intelligence/

Best Articles 2025

It's time to vote for your favourite articles from the 2025 journals. Which did you enjoy? Which did you learn most from? Which made you think?

Voting is open online at:
https://www.surveymonkey.com/r/K6BW55Y

Select up to 3 'favourites' from each journal.

Past issues of both journals are available on ACCU's website: accu.org

- *Overload* is available to everyone
- You can only read *CVu* if you're a member

Concurrency Flavours

Concurrency has many different approaches. Lucian Radu Teodorescu clarifies terms, showing how different approaches solve different problems.

Most engineers today use concurrency – often without a clear understanding of what it is, why it's needed, or which flavour they're dealing with. The vocabulary around concurrency is rich but muddled. Terms like parallelism, multithreading, asynchrony, reactive programming, and structured concurrency are regularly conflated – even in technical discussions.

This confusion isn't just semantic – it leads to real-world consequences. If the goal behind using concurrency is unclear, the result is often poor concurrency – brittle code, wasted resources, or systems that are needlessly hard to reason about. Choosing the right concurrency strategy requires more than knowing a framework or following a pattern – it requires understanding what kind of complexity you're introducing, and why.

To help clarify this complexity, the article aims to map out some of the main flavours of concurrency. Rather than defining terms rigidly, we'll explore the motivations behind them – and the distinct mindsets they evoke. While this article includes a few C++ code examples (using features to be added in C++26), its focus is conceptual – distinguishing between the flavours of concurrency. Our goal is to refine the reader's taste for concurrency.

The core idea

Concurrency is about dealing with multiple activities that overlap in time. It is dealing with complexity on the time axis. Mathematically, concurrency corresponds to a strict partial ordering of work items; this is different from sequential execution where we have a (strict) total ordering of work items.

That is, if we have two distinct work items a and b, then, at execution time, we might have the following possibilities:

- $a < b$
- $b < a$
- neither of the above – the execution of the two work items overlap

By comparison, in sequential execution , either a happens before b, or b happens before a – they cannot overlap. Thus, concurrent execution adds more complexity by adding a degree of freedom in the execution time axis.

We use the < symbol to represent the *happens-before* relation between work items. Although the literature often denotes this relation with → [Lamport78], using < better emphasises that we are discussing the ordering between work items.

This is it! That's what concurrency is all about.

Understanding concurrency as partial ordering helps us see why it's hard and why it matters. In the next section, we explore what motivates us to introduce this complexity in the first place.

Motivation

We would not introduce the complexities that come with concurrency if we didn't gain anything from it. There are multiple reasons for why we would want to use concurrency. The biggest classes of reasons are:

- Responsiveness: keep the system interactive.
- Performance: do more work or do it faster; this can be divided into:
  - improving latency: completing work faster;
  - improving throughput: doing more work per time unit.
- Decomposition: breaking the problem into independent parts.
- Coordination: handling many independent inputs or events.

Each of these motivations calls for a different way of thinking about concurrency – a different flavour, if you will.

These might be hard to grasp, so let's illustrate them using a real-world analogy: a restaurant – The Burrs – owned by Charlie. If we were to consider a non-concurrent world, we would have just one employee (Charlie) that runs all the activities in the restaurant (greeting customers, taking orders, cooking, bringing food, taking payment, etc.).

One could model in code this situation as shown in Listing 1. All the activities are executed sequentially, in order. As one could imagine running the restaurant this way would be a complete disaster. As soon as the restaurant has two customers, one customer will be starved – they would have to wait for the first customer to be completely served. No clever scheduling within a single thread can compensate for the need to serve multiple customers simultaneously. No matter how Charlie would be at his job, this organization of work will not work.

Responsiveness

Let's look first at responsiveness. If there is one person that is doing all the job, then customers might need to wait for long periods of time for different activities. Imagine that Charlie is cooking for 30 minutes, and customers cannot be seated while cooking is done. That would make new customers run away and never come back.

```
void run_restaurant() {
  while (restaurant_is_open) {
    for (auto c: new_customers) {
      sit_customer(c);
    }
    for (auto c: seated_customers) {
      auto o = take_order(c);
      auto food = cook(o);
      serve(c, food);
      wait_for_completion(c);
      auto b = print_bill(c);
      cash_in(c, b);
    }
  }
}
```

Listing 1

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

in interactive software, we must ensure that the part of the system responsible for front-line interaction remains reactive and free to respond

In the restaurant world, it would be fair to have a rule like this: if there are empty tables, a new customer is seated in less than 2 minutes. In the software world, this translates into a requirement.

In order to implement such a rule, the restaurant needs to have a lead host that doesn't perform activities that can't be interrupted for more than 2 minutes. This ensures that the lead host can always greet customers within 2 minutes. To keep the lead host (let's assume it's our friend Charlie) available for new customers, the rest of the duties need to be performed by other employees. Thus, we have more than one employee, so different activities might overlap – we've introduced concurrency in the restaurant.

A possible encoding is shown in Listing 2. Here we are using coroutines (more specifically C++26's `std::task` [WG21Task]) to encapsulate concurrency concerns. The body of the `lead_host` coroutine can be executed concurrently with `serve_flow`, and multiple `serve_flow` may coexist at the same time. The spawn function will start the work described by `serve_flow` and will associate its completion with `serving_scope`, which keeps track of how many serving flows are still in progress. Through `serving_scope`, we use a `counting_scope` [WG21Scope] to ensure that we maintain a structured lifetime for all serving flows, and we can perform a clean shutdown (i.e., we cannot close the restaurant for the day while we still have customers serving dinner).

Here, Charlie acts as the system's *reactor* – the component that must remain unblocked and responsive – while `serve_flow` tasks represent background operations that can take longer but must not interfere with incoming events.

The same responsiveness principle applies beyond restaurants: in interactive software, we must ensure that the part of the system responsible for front-line interaction remains reactive and free to respond. This is especially evident in UI frameworks, where the main thread must remain responsive to user input. If a UI freezes while processing data or waiting on I/O, the system appears broken, even if it's functioning correctly in the background.

Improving latency

Let's assume that the average time to cook a dish in the restaurant is 30 minutes. If there are four people at a table, and we want to bring the food to the table all at once, we don't want to wait for 2 hours before bringing the food to the table. We want to keep the waiting time for the table to be around 30 minutes, regardless of the number of people at the table. For that, we could prepare the 4 dishes concurrently.

One way to prepare multiple meals simultaneously is to hire more cooks, each handling a separate dish. Another alternative would be to perform 'cooperative cooking'. That is, a single cook can prepare more than one dish at the same time, by shifting their attention to different dishes and sequencing some of the activities that require their full attention. More on this later.

Improving latency often requires doing more work overall.

```
std::task<> serve_flow(Customer, Table);
exec::counting_scope serving_scope;

std::task<> lead_host() {
  while (restaurant_is_open()) {
    auto event = co_await lead_host_events();
    if (event.type() == new_customer_arrived) {
      std::print( "Good evening, and welcome to
The Burrs. I'm Charlie Burr and I will be your
host tonight. Allow me to confirm your table.");

      auto c = event.new_customer();
      auto r = get_or_create_reservation(c);
      auto t = acquire_table(r);

      // The serving flow is done by dedicated
      // personnel
      exec::spawn(serve_flow(c, t),
          serving_scope);
    }
    else {
      // other host and leadership duties
    }
  }
}
```

Listing 2

Let's return to our restaurant analogy to illustrate this concept. If The Burrs restaurant has soup on the menu, then it needs to have a house-made base stock (with one or multiple flavours) that is prepared at the beginning of the day, from which the actual soups on the menu are derived. Preparing a tripe soup usually takes between 3.5 to 5 hours (and, in some cases, even more) from start to finish – obviously the restaurant cannot start this process when the order is placed. But, in preparing the soup at the beginning of the day, we intentionally overproduce – accepting waste in exchange for responsiveness. In other words, optimising for latency can introduce inefficiencies elsewhere.

Another example is buying all the ingredients in bulk ahead of time. No matter how good the planning is, a typical restaurant often buys more ingredients that it needs, and some of those are just waste (they can't be reused).

This contrasts with throughput-oriented design, where batching and reuse are prioritised over quick responses.

This principle carries over directly to software: to reduce latency, we often do more total work. A classic example is the parallel prefix sum algorithm [Wikipedia]. For a given array of numbers x_i, the algorithm will compute an array y_i, such as $y_i = \sum_{j=0}^{i} x_j$. For example, for input array 1, 2, 3, 4, 5, the algorithm computes: 1, 3, 6, 10, 15. The sequential algorithm can be implemented with a simple `for` loop, and will execute n additions.

Parallelising this algorithm using Blelloch's method [Blelloch90] yields a total work of $2n$-2 additions, but having enough threads can lead to finishing the algorithm in 2 log n steps. Thus, we do more work to (hopefully) finish faster. Note: we completely ignored here the effects

Likewise in software, introducing concurrency – and adjusting how work is structured – can significantly improve throughput

of synchronization, which make the parallel algorithm work worse in practice.

In the literature, this is often described as the difference between the *work* and the *span*. These two measures capture complementary aspects of execution. The *work* captures the sum of all the operations that need to be completed for an algorithm. The *span* (also called *critical path length* or *depth*) represents the longest chain of dependencies between the operations – the minimum time to execute the algorithm if we would have an infinite number of processors.

Improving throughput

Improving latency is not always the goal. Often, if we improve latency too much, we decrease throughput – the number of operations we can make in the unit of time. In Charlie's restaurant, from a business perspective, it's not that important to serve one customer in the least amount of time; it's more important to maximise the number of customers served per day.

Let's assume that Charlie hires three cooks. Let's assume that, if a cook dedicates their entire attention to a dish, then they can finish up the dish in 25 minutes – the best time we can get. But, this arrangement means that The Burrs cannot serve more than 3 customers at once – one cook per customer. If a cook can divide their attention among several dishes (say, 5 at a time), each dish might take 30 minutes – but the restaurant could now serve 15 customers at once. It makes much more sense to increase the latency from 25 to 30 minutes, but increase the throughput from 3 to 15 customers.

In this example, we've introduced two levels of concurrency to be able to cook for 15 customers at once: we have more than one cook, and each cook can work on more than one dish at a given time.

Likewise in software, introducing concurrency – and adjusting how work is structured – can significantly improve throughput.

In CPU-bound applications, maximising throughput means avoiding oversubscription. That is, if the machine has 12 cores, we should avoid running more than 12 CPU-intensive threads in parallel. Creating more threads forces the scheduler to constantly switch between them, reducing overall performance. On a single core, it's usually faster to run two CPU-intensive tasks sequentially than running them on two threads. For some engineers this may seem counterintuitive, but running these two operations sequentially actually improves throughput.

This is the main reason behind system-wide thread pools. It's best for an application to submit all the CPU intensive work to a thread pool with a number of threads that matches the number of cores of the target machine. Other operations (e.g., I/O) can be put on a different thread pool.

In the upcoming C++26 execution model, there are two facilities that are targeting scenarios that require high throughput. One is `parallel_scheduler` [WG21ParSched], a global thread pool that aims at limiting oversubscription. The other one is the `bulk` family of sender algorithms, which aim at executing work concurrently.

```
void chop_all_onions(std::vector<Onion> onions) {
  auto sched = exec::get_system_scheduler();
  auto snd = exec::schedule(sched)
    | exec::bulk(onions.size(), [=](size_t i) {
        chop_one_onion(onions[i]);
      });
  exec::sync_wait(std::move(snd));
}
```

Listing 3

Listing 3 provides a simple example of using these two facilities. For the job of chopping onions, the productivity scales linearly up with the number of workers. But each worker should chop only one onion at a time. If they try to handle multiple simultaneously – constantly switching between them – throughput drops.

Decomposition

Sometimes, we introduce concurrency not to boost performance, but simply to make the application easier to understand. In these cases, it's often easier to think in terms of independent *agents* than to break the system into functional units.

Charlie might soon realise that, while one person can fill multiple roles, specialisation still helps. Specialisation enables clearer accountability, stronger role focus, better performance, and a more manageable business. The restaurant might have a manager, lead host, servers, bartenders, bussers, chefs, sous chef, cooks, dishwashers, etc. While roles can overlap, specialisation makes the system easier to reason about and manage.

In the previous example we had a split by role. Alternatively, we can imagine decomposing by location – say The Burrs becomes a chain with multiple restaurants. While a centralised manager is possible, there are clear advantages to assigning a manager to each location. A local manager can address specific issues relevant to that location, without misapplying solutions that worked elsewhere.

In this way, concurrency helps us focus on specific concerns – and structure systems in clearer, more maintainable ways.

A good example from the software industry is a web server. While it's possible to write a web server with just one thread, giving each request its own thread often makes the logic easier to follow. Reasoning becomes simpler when each request runs independently, rather than interleaving unrelated tasks on a single thread.

Coordination

Sometimes, the temporal complexity of an application comes from the way it reacts to a variety of stimuli: events, inputs, messages. In such cases, the main concerns are correctness, consistency, and the timeliness of those responses. And concurrency helps in this case as well. While decomposition focuses on breaking work into pieces, coordination focuses on handling interactions between concurrent entities. Coordination is fundamentally about *reactivity* and *synchronisation*. This mindset forces

us to think about causality, state changes, race conditions, and timing constraints.

In a restaurant business, a good example of using concurrency for coordination is the system put in place to ensure that the customer is served. The host might greet the customer and lead them to an empty table. Occupying an empty table is a signal to one of the waiters that they need to take care of the new customer (the table is a fixed shared structure on which both the host and the waiters interact). After taking the order, the waiter might lead a note to the kitchen with what dishes need to be prepared – this is done through a system that resembles a queue. The cooks are either pooling the queue, or they get notified whenever there is a new entry in the queue. After taking an order and cooking the needed dishes, the cooks typically signal back the waiters that the food is prepared (maybe with a note to indicating which order was cooked). The waiter needs to bring the food back to the customer, without mixing the table. When the customer finishes, the waiter will provide the bill to the customer and accept payment. Depending on the local customs and on the type of restaurant this process might be different, but it's important that a customer pays for the food that they ordered.

In this restaurant system, multiple types of events occur independently of each other: customers can come into the restaurant, customers can order food, food is prepared, customers ask for the bill, customers paying, etc. While the flow must remain customer-centric, concurrency ensures that all these events are handled in a timely manner and that the overall process remains manageable.

The same coordination challenges arise when building software. Taking the previous example of a web server, let's assume that handling one request implies communicating with multiple downstream services. Whether a request is handled on a single thread matters less than ensuring that its outgoing calls are matched correctly to their responses, and that continuations run as expected – even across threads. In terms of implementation, we can still handle one request on one thread, but this has some challenges: the thread will be blocked until the responses are received (and that might take a long time), and also we might have a limit on the number of threads we can create.

Another way to structure this type of problem is to use thread pools and letting the continuation to a response be handled on a different thread than the one used to make the request. Listing 4 shows how this can be implemented really easily with the help of coroutines.

In a thread-per-request model, it's easy to associate state with the thread. But once we switch to shared thread pools, we need to explicitly carry context (user ID, request ID, continuation function, etc.). This shift – from implicit context to explicit coordination – is one of the hardest conceptual changes in writing concurrent code.

Concurrency becomes essential not to improve speed, but to preserve correctness in the face of overlapping stimuli.

To support such coordination patterns, modern languages and platforms offer a variety of frameworks/models:

- Event loops (e.g., GUI frameworks, Node.js)
- Actor models (e.g., Erlang, Akka)
- Reactive streams (e.g., Rx, ReactiveX)
- Async coroutines and futures (e.g., `std::task`)

Common distinctions in concurrency discussions

Now that we covered the main types of motivation for adding concurrency to applications, let's covered a few distinctions that are often made in the industry.

Concurrency vs parallelism

Concurrency and parallelism are often used interchangeably – but they describe different aspects of computation:

```
std::task<MyResponse> request_handler(Request r)
{
  validate(r);
  initial_processing(r);
  auto oc = prepare_outgoing_call(r);
  auto res = co_await outgoing_call(oc);
  continue_processing(r, res);
  co_return generate_response(r);
}
```

Listing 4

- Concurrency is about structure: tasks may overlap in time, but they do not necessarily execute simultaneously.
- Parallelism is about execution: tasks actually run at the same time.

Think of concurrency as a *design-time* concern – it's about structuring systems to handle overlapping work. Parallelism, by contrast, is a *run-time* reality – it depends on physical resources and actual scheduling.

A common example is that interleaved execution (on a single core) is concurrency, but not parallelism.

Ultimately, this distinction may not always be useful in practice. For most programming environments, there is little we can control about the actual execution of work items. The compiler can reorder instructions, and even the hardware will execute instructions out of order. Even on a single thread, there are cases when instructions are executed in parallel. It is definitely useful to understand possible execution options, but we mostly should be concerned with the design of the application.

In practice, systems often involve both: concurrency in design, parallelism in execution. But because programmers directly control design but not the execution, I argue that we need to focus on concurrency. This is why I tend to exclusively use the term *concurrency*.

Concurrent vs parallel forward progress guarantees

The C++ standard makes a further distinction between *concurrent* and *parallel* progress guarantees [WG21Progress] – one that can often lead to confusion. This distinction matters as it allows us to discuss what types of concurrent algorithms we might execute on certain execution agents.

A thread of execution provides *concurrent progress guarantee* if the implementation guarantees that it makes progress (if not stopped). That is, no matter how many other threads are in the system, no matter what they are doing, the thread will start and continue to make progress, until it completes or it's stopped.

A thread of execution provides *parallel progress guarantee* if it is not required to begin executing immediately – but once it starts, it behaves as if it has a *concurrent progress guarantee*. That is, the thread might never get an execution resource, but if it gets one, then it continuously makes progress.

A good example of parallel progress guarantee would be a thread pool with fixed size. Let's say that the pool has only 4 threads, and we want to execute 5 tasks. If these tasks will continue to execute work without completing, one task will never start. This is the expected behavior under parallel progress guarantee.

Algorithms using a latch or barrier with count N require at least N threads with concurrent progress guarantees – otherwise, some threads may never reach the synchronisation point. As all threads are waiting to meet at the latch/barrier point, if we have less than N threads that arrive at that point, we will deadlock the threads that arrive there.

Moving to the restaurant analogy, we might have a rule that all the dishes are brought at the same time, each person at the table being served by a waiter. The waiters lift their plates and wait until all required waiters are ready and only then and only them deliver them to the table. If the number of persons that need to be served is greater than the number of waiters available, we would reach a deadlock (the waiters are waiting with the plates on their hand, without being able to do anything else). Parallel progress guarantees allow a fixed number of waiters to serve all tables,

even if it means some customers may wait longer, and this can lead to deadlocks under the above rule. Concurrent progress guarantees imply that we would always hire enough waiters to avoid blocking – ensuring progress for every task.

To avoid these types of deadlocks, programmers should:

- avoid algorithms that require concurrent progress guarantee (i.e., allow less waiters to serve more people at a table)
- avoid blocking the execution threads (allow the waiter to do other things while waiting for all the plates to be ready for delivery)

To strengthen these recommendations, in C++ today, there's no portable way to create a thread pool that provides true concurrent progress guarantees.

Concurrency, multithreading, asynchrony

Since we've spent time exploring concurrency flavours, it's worth also touching on the differences between concurrency, multithreading, and asynchrony.

Multithreading is an implementation technique in which we get concurrency by executing multiple threads inside a single process. We typically use OS threads as the main execution agent, and synchronization primitives (mutexes, semaphores, etc.) for coordination. The communication method typically happens via shared memory.

Asynchrony is about how to express the waiting, thus its main concern is about control flow. The typical goal is to avoid blocking and simplify the asynchronous logic. In the upcoming C++ 26 standard, asynchrony may be expressed with coroutines or with the new senders/receivers framework.

If concurrency is more concerned with the overlapping execution of work, asynchrony is more concerned with the gaps between work. Asynchrony, in most frameworks, provides a form of concurrency – one that avoids blocking by explicitly modelling waiting.

One way to achieve conceptual unity is to focus on concurrency, and acknowledge that there might be multiple flavors of concurrency, based on the application domain and chosen technology.

Domain-specific concurrency

Beyond general-purpose concurrency patterns, some domains develop their own concurrency models tailored to their problem space. For example, **GPU programming** uses *SIMD-style parallelism*, requiring different thinking than thread-based concurrency. Similarly, **dataflow programming** (as seen in TensorFlow or LabVIEW) treats computation as a graph of flowing data, where concurrency is implicit in the structure.

While this article focuses on concurrency as commonly seen in systems and application programming, it's worth remembering that domain constraints and idioms often give rise to specialised forms of concurrency that don't map cleanly onto threads, futures, or actors.

These specialised models lie beyond the scope of this article, but they reinforce the broader message: concurrency adapts to the shape of the problem.

Wrapping it up

When we talk about concurrency, one size doesn't fit all. We have different motivations, different terminology, and different problems to solve – and each leads us to different flavours of concurrency. Whether we care about responsiveness, throughput, structure, or coordination, our reasoning about concurrency should start with understanding why we're introducing it in the first place.

This article has aimed to offer a conceptual map of those motivations – not to define terms rigidly, but to clarify the mindsets and trade-offs involved. From restaurant metaphors to C++26 features, we've looked at how concurrency emerges in different forms depending on what we value.

There's more to say, of course. We haven't covered key concerns like debuggability and observability, or the risks of over-engineering with concurrency when simpler designs suffice. Nor have we explored domain-specific models like cooperative multitasking, GPU scheduling, or distributed coordination. But even without going there, the main lesson remains:

Concurrency isn't one thing. It's a spectrum of strategies – each shaped by what you're trying to achieve.

However, from a mathematical perspective, concurrency is deceptively simple: it's just a partial ordering of work items. But that simplicity introduces a new dimension of complexity – one rooted in time. And, as Brooks warned us, *there is no silver bullet* [Brooks95]: this complexity cannot be abstracted away; it interacts with the existing complexity of the application and often amplifies it.

The key is not just to be concurrent, but to be deliberate – because taste matters. So, next time you add concurrency, don't ask *how* first – ask *why*. ■

References

[Blelloch90] Guy E. Blelloch, 'Prefix Sums and Their Applications' *Technical Report CMU-CS-90-190*, School of Computer Science, Carnegie Mellon University, 1990.

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

[Lamport78] Leslie Lamport. 'Time, Clocks, and the Ordering of Events in a Distributed System', *Communications of the ACM 21*, no. 7, 1978. https://lamport.azurewebsites.net/pubs/time-clocks.pdf.

[WG21Progress] WG21, 'Forward Progress' in *Working Draft Programming Languages – C++*, https://eel.is/c++draft/intro.progress, accessed Oct 2025.

[WG21Task] WG21, 'execution::task' in *Working Draft Programming Languages – C++*, https://eel.is/c++draft/exec.task, accessed Oct 2025.

[WG21Scope] WG21, 'Counting Scopes' in *Working Draft Programming Languages – C++*, https://eel.is/c++draft/exec.counting.scopes, accessed Oct 2025.

[WG21ParSched] WG21, 'Parallel scheduler' in *Working Draft Programming Languages – C++*, https://eel.is/c++draft/exec.par.scheduler, accessed Oct 2025.

[Wikipedia] Wikipedia: 'Prefix sum', https://en.wikipedia.org/wiki/Prefix_sum.

Coroutines – A Deep Dive

Coroutines are powerful but require some boilerplate code. Quasar Chunawala explains what you need to implement to get coroutines working.

Following on from the introduction in the editorial, let's understand the basic mechanics needed to code up a simple coroutine, and I'll show you how to yield from the coroutine and await results.

The simplest coroutine

The following code is the simplest implementation of a coroutine:

```
#include <coroutine>
void coro_func(){
  co_return;
}
int main(){
  coro_func();
}
```

Our first coroutine will just return nothing. It will not do anything else. Sadly, the preceding code is too simple for a functional coroutine and it will not compile. When compiling with gcc 15.2, we get the error shown in Figure 1.

```
<source>: In function 'void coro_func()':
<source>:4:5: error: unable to find the promise
type for this coroutine
    4 |     co_return;
      |     ^~~~~~~~
```

Figure 1

Looking at C++ reference, we see that the return type of a coroutine must define a type named **promise_type**.

The promise_type

Why do we need a promise? The **promise_type** is the second important piece in the coroutine mechanism. We can draw an analogy from futures and promises which are essential blocks for achieving asynchronous programming in C++. The future is the thing, that the function that does the asynchronous computation, hands out back to the caller, that the caller can use to retrieve the result by invoking the **get()** member function. The future has the role of the return object. But, you need something that remains on the producer-side. The asynchronous function holds on to the promise, and that's where it puts the results that will be given to the caller, when it calls **get()** on the future. The idea behind the **promise_type** for coroutines is similar. The promise is where the coroutine's return value is stored, and it provides functions to control the coroutine's behavior at startup and completion of the coroutine. The promise is the interface through which the caller interacts with the coroutine.

Following the reference advice, we can write a new version of our coroutine.

Quasar Chunawala has a bachelor's degree in Computer Science. He is a software programmer turned quant engineer, enjoys building things ground-up and is deeply passionate about programming in C++, Rust and concurrency and performance-related topics. He is a long-distance hiker and his favorite trekking routes are the Goechala route in Sikkim, India and the Tour-Du-Mont Blanc (TMB) circuit in the French Alps. Contact him at quasar.chunawala@gmail.com.

```
#include <coroutine>
struct Task
{
  struct promise_type
  {
  };
};
Task coro_func(){
  co_return;
}
int main(){
  coro_func();
}
```

Note that the return type of a coroutine can have any name (I call it **Task**, so that it makes intuitive sense). Compiling the preceding code again gives us errors. All errors in Figure 2 are about missing functions in the **promise_type**.

One of the important functions of the **promise_type** is that it determines what happens at certain key points in the coroutine's life. It determines, what happens at the startup and completion of execution of the coroutine.

Implementing the promise_type

The first thing that the compiler expects from us is the **get_return_object()** function. The return type of this function is the same as the return type of the coroutine. (See Listing 1, and the output is in Figure 3, both on next page.)

The **get_return_object()** method is implicitly called when the coroutine starts executing. Its up to the **promise_type** to provide an implementation of this method that constructs the return object that will be handed back to the caller. The **Task** object is stored on the heap. You don't see this in the source code anywhere. When the coroutine reaches its first suspension point, and control flow is returned back to the caller, then the caller will receive this object.

The **return_void()** function is a customization point for handling what happens when we reach the **co_return** statement in the function

```
<source>: In function 'Task coro_func()':
<source>:11:5: error: no member named 'return_
void' in
'std::__n4861::__coroutine_traits_impl<Task,
void>::promise_type' {aka 'Task::promise_type'}
   11 |     co_return;
      |     ^~~~~~~~
<source>:10:6: error: no member named 'unhandled_
exception' in
'std::__n4861::__coroutine_traits_impl<Task,
void>::promise_type' {aka 'Task::promise_type'}
   10 | Task coro_func(){
      |      ^~~~~~~~
<source>:10:6: error: no member named 'get_
return_object' in
'std::__n4861::__coroutine_traits_impl<Task,
void>::promise_type' {aka 'Task::promise_type'}
```

Figure 2

```
#include <coroutine>
#include <print>
struct Task{
  struct promise_type{
    Task get_return_object(){
      std::println("get_return_object()");
      return Task{ *this };
    }
    void return_void() noexcept {
      std::println("return_void()");
    }
    void unhandled_exception() noexcept {
      std::println("unhandled_exception()");
    }
    std::suspend_always initial_suspend()
    noexcept {
      std::println("initial_suspend()");
      return {};
    }
    std::suspend_always final_suspend() noexcept{
      std::println("final_suspend()");
      return {};
    }
  };
  explicit Task(promise_type&){
    std::println("Task(promise_type&)");
  }
  ~Task() noexcept{
    std::println("~Task()");
  }
};
Task coro_func(){
  co_return;
}
int main(){
  coro_func();
}
```

Listing 1

body. There is also a corresponding **return_value()**, if you don't have an empty **co_return** statement, but we'll look at this at length later ahead.

The **unhandled_exception()** is similar to the **return_void()**, this function is a customization point for handling, what happens when the coroutine throws an exception. We leave it empty for now.

We need to implement two more functions **initial_suspend()** and **final_suspend()**. These are basically the customization points that allow us to execute some code, both when the coroutine first starts executing and shortly before the coroutine ends execution. Here, we are returning **std::suspend_always** which basically means that at these points, I want to go into suspension.

In a typical implementation, you either return **std::suspend_always**, which means you pause execution at this point and hand control back to the caller always, or you return **std::suspend_never**, which basically means you just go on and continue executing the coroutine.

Note that, **final_suspend()** is not printed in the output, because the coroutine is paused at **initial_suspend()** and since I never resumed it, I don't see the output on the console.

A yielding coroutine

Let's implement another coroutine that can send data back to the caller. In this second example, we implement a coroutine that produces a message. It will be the hello world of coroutines. The coroutine will say hello and the caller function will print the message received from the coroutine.

To implement this functionality, we need to establish a communication channel from the coroutine to the caller. This channel is the mechanism

```
get_return_object()
Task(promise_type&)
initial_suspend()
~Task()
```

Figure 3

```
Task coro_func(){
  co_yield "Hello world from the coroutine";
  co_return;
}
int main(){
  auto task = coro_func();
  std::print("task.get() = {}", task.get());
  return 0;
}
```

Listing 2

that allows the coroutine to pass values to the caller and receive information from it. This channel is established through the coroutine's **promise_type** and the *coroutine handle*.

The *coroutine handle* is a type that gives access to the coroutine frame(the coroutine's internal state) and allows the caller to resume or destroy the coroutine. The handle is what the caller can use to resume the coroutine after it has been suspended (for example after **co_await** or **co_yield**). The handle can also be used to check whether the coroutine is done or to clean up its resources.

The code in Listing 2 is the new version of both the caller function and the coroutine.

The coroutine *yields* and sends some data to the caller. The caller reads that data and prints it. When the compiler reads the **co_yield** expression, it will generate a call to the **yield_value** function defined in the **promise_type**. Thus, we add the code in Listing 3 to the **promise_type** .

The function gets a **std::string** object and moves it to the **output_data** member variable of the promise type. But, this just keeps the data inside the **promise_type**. We still need a mechanism to get that data out of the coroutine.

The coroutine handle

Once we require a communication channel to and from a coroutine, we need a way to refer to a suspended or executing coroutine. The mechanism to refer to the coroutine object is through a pointer or handle called a **coroutine handle**. The C++ library header file **<coroutine>** defines a type **std::coroutine_handle** to work with coroutine handles.

Two functions are of interest to us in the **std::coroutine_handle** interface : **resume()** and **destroy()**.

```
struct coroutine_handle<promise_type>{
  /* ... */
  void resume() const;
  void destroy() const;
  promise_type& promise() const;
  static coroutine_handle from_promise(
    promise_type&);
}
```

What **resume()** does is simply, it resumes the suspended coroutine. It continues execution.

```
struct Task{
  struct promise_type{
    std::string output_data;
    /* ... */
    std::suspend_always yield_value(
         std::string msg) noexcept{
      output_data = std::move(msg);
    }
  };
  explicit Task(promise_type&){
    std::println("Task(promise_type&)");
  }
  ~Task() noexcept{
    std::println("~Task()");
  }
};
```

Listing 3

```
struct Task{
  struct promise_type{
    std::string output_data;
    /* ... */
    std::suspend_always yield_value(
         std::string msg) noexcept{
      output_data = std::move(msg);
    }
  };
  // Coroutine handle member-variable
  std::coroutine_handle<promise_type> handle{};
  explicit Task(promise_type& promise)
  : handle { std::coroutine_handle<promise_type>
             ::from_promise(promise) }
  {
    std::println("Task(promise_type&)");
  }
  // Destructor
  ~Task() noexcept{
    std::println("~Task()");
    if(handle)
      handle.destroy();
  }
};
```

Listing 4

If we think of this coroutine frame or object living somewhere on the heap, where all of the state of execution is stored, one way to destroy this state is to let the coroutine run to completion. But, far more commonly, we would like to manage the lifetime externally and we can then just call the `destroy()` function which will then get rid of the coroutine state.

Note that, the `coroutine_handle` is not a smart pointer type. So, you have to call the `destroy()` explicitly.

There are two more functions: `.promise()` and `.from_promise()`. These are used to convert from a coroutine to a promise object and vice versa.

We add the functionality in Listing 4 to our return type to manage the coroutine handle.

The preceding code declares a coroutine handle of type `std::coroutine_handle<promise_type>` and creates the handle in the return type constructor. The handle is destroyed in the return type destructor.

Now, back to our yielding coroutine. The only missing bit is a `get()` function for the caller to be able to extract the resultant string out of the promise.

```
std::string get(){
  if(!handle.done()){
      handle.resume();
  }
  return std::move(handle.promise().output_data);
}
```

The `get()` function resumes the coroutine if it has not terminated and return the result stored in the `output_data` member variable of the promise. The full source code listing is shown in Listing 5, and the output in Figure 4.

The output shows what is happening during the coroutine execution. The `Task` object is created after a call to `get_return_object`. The coroutine is initially suspended. The caller wants to get the message from the coroutine so `get()` is called, which resumes the coroutine. When the compiler sees the `co_yield` statement in the coroutine, it generates an implicit called to `yield_value(std::string)`. `yield_value` is called and the message is copied to the resultant member variable `output_data` in the promise. Finally, the message is printed by the caller function, and the coroutine returns.

```
get_return_object()
Task(promise_type&)
initial_suspend()
get()
yield_value(std::string)
Hello world from the coroutine
~Task()
```

Figure 4

A waiting coroutine

We are now going to implement a coroutine that can wait for the input data sent by the caller. In our example, the coroutine will wait until it gets a `std::string` object and then print it. We say that the coroutine waits, we mean it is suspended (that is, not executed) until the data is received.

We start with changes to both the coroutine and the caller function:

```
#include <coroutine>
#include <print>
#include <iostream>
#include <string>

using namespace std::string_literals;

struct Task{
  struct promise_type{
    std::string output_data{};

    Task get_return_object(){
      std::println("get_return_object()");
      return Task{ *this };
    }
    void return_void() noexcept {
      std::println("return_void()");
    }
    void unhandled_exception() noexcept {
      std::println("unhandled_exception()");
    }
    std::suspend_always initial_suspend()
         noexcept{
      std::println("initial_suspend()");
      return {};
    }
    std::suspend_always final_suspend() noexcept{
      std::println("final_suspend()");
      return {};
    }
    std::suspend_always yield_value(
         std::string msg) noexcept{
      std::println("yield_value(std::string)");
      output_data = std::move(msg);
      return {};
    }
  };
  // Coroutine handle member-variable
  std::coroutine_handle<promise_type> handle{};

  explicit Task(promise_type& promise)
  : handle { std::coroutine_handle<promise_type>
    ::from_promise(promise) }
  {
    std::println("Task(promise_type&)");
  }
  ~Task() noexcept{
    std::println("~Task()");
    if(handle)
      handle.destroy();
  }
  std::string get(){
    std::println("get()");
    if(!handle.done())
      handle.resume();
    return std::move(handle.promise()
      .output_data);
  }
};
Task coro_func(){
  co_yield "Hello world from the coroutine";
  co_return;
}
int main(){
  auto task = coro_func();
  std::cout << task.get() << std::endl;
}
```

Listing 5

```
Task coro_func(){
  std::cout << co_await std::string{};
  co_return;
}
int main(){
  auto task = coro_func();
  task.put("To boldly go where no man has gone"
    "before");
  return 0;
}
```

In the preceding code, the caller function calls the `put()` function(a method in the return type structure) and the coroutine calls `co_await` to wait for a `std::string` object from the caller.

The changes to the return type are simple, that is, just adding the `put()` function.

```
void put(std::string msg){
  handle.promise().input_data = std::move(msg);
  if(!handle.done()){
    handle.resume();
  }
}
```

We need to add the `input_data` variable to the promise structure. But, just with those changes to our first example and the coroutine handle from the previous example, the code cannot be compiled. (See Listing 6.)

The compiler gives us the error in Figure 5.

Let's explore more about what this error message means.

What is an awaitable?

An *awaitable* is any object, I can call `co_await` on. You can think of `co_await` like an operator, and its argument as an *awaitable*. The way to think about the operator `co_await` is that these are opportunities for suspension. These are the points where the coroutine can be paused. Similar to, how the `promise_type` provides hooks to control what happens at startup or when you return from the coroutine, the awaitable provides these hooks for what happens when we go into suspension.

```
struct Awaitable{
  bool await_ready();
  void await_suspend(
    std::coroutine_handle<promise_type>);
  void await_resume(
    std::coroutine_handle<promise_type>);
};
```

The first function is `.await_ready()` which returns a `bool`. This determines whether we do actually go into suspension or we just say, *yeah, we are ready, and we don't want to go into suspension*, we want to continue execution and in that case we just return `true`.

The next function is `.await_suspend()` and that is the customization point that will get executed shortly before the coroutine function goes to sleep.

The next function is `.await_resume()` and that is the customization point that will get execute just after the coroutine is resumed.

The code in Listing 7 (next page) is our implementation of the `await_transform` function and the `Awaitable` struct:

Listing 8 (also next page) is the code for the full example of the waiting coroutine and Figure 6 is the output.

Coroutine generators

A **generator** is a coroutine that generates a sequence of elements by repeatedly resuming itself from the point that it was suspended.

A generator can be seen as an infinite list, because it can generate an arbitrary number of elements.

Implementing even the most basic coroutine in C++ requires a certain amount of code. C++23 introduced the `std::generator` template class. I present – in Listing 9, on next page – the source code for a simple `FibonacciGenerator`.

```
#include <print>
#include <string>
#include <coroutine>
#include <iostream>

struct Task{
  struct promise_type{
    std::string input_data{};

    Task get_return_object() noexcept{
      std::println("get_return_object");
      return Task{ *this };
    }
    void return_void() noexcept{
      std::println("return_void");
    }
    std::suspend_always initial_suspend()
        noexcept{
      std::println("initial_suspend");
      return {};
    }
    std::suspend_always final_suspend() noexcept{
      std::println("final_suspend");
      return {};
    }
    void unhandled_exception() noexcept{
        std::println("unhandled_exception");
    }
    std::suspend_always yield_value(
        std::string msg) noexcept{
      std::println("yield_value");
      return {};
    }
  };

  std::coroutine_handle<promise_type> handle{};

  explicit Task(promise_type& promise)
  : handle{ std::coroutine_handle<promise_type>
    ::from_promise(promise)}
  {
    std::println("Task(promise_type&) ctor");
  }
  ~Task() noexcept{
    if(handle)
      handle.destroy();
    std::println("~Task()");
  }
  void put(std::string msg){
    handle.promise().input_data = std::move(msg);
    if(!handle.done())
      handle.resume();
  }
};

Task coro_func(){
  std::cout << co_await std::string{};
  co_return;
}

int main(){
  auto task = coro_func();
  task.put("To boldly go where no man has gone "
           "before");
  return 0;
}
```

Listing 6

```
<source>: In function 'Task coro_func()':
<source>:62:18: error: no member named 'await_
ready'
in 'std::string' {aka 'std::__cxx11::basic_
string<char>'}
   62 |     std::cout << co_await std::string{};
      |                  ^~~~~~~~
```

Figure 5

```
auto await_transform(std::string) noexcept{
  struct Awaitable{
    promise_type& promise;

    bool await_ready() const noexcept{
      return true; // Says, yeah we are ready, we
              // don't need to sleep. Just go on.
    }
    std::string await_resume() const noexcept{
      return std::move(promise.input_data);
    }
    void await_suspend(
      std::coroutine_handle<promise_type>) const
      noexcept{}
  };
  return Awaitable(*this);
}
```

Listing 7

```
get_return_object
Task(promise_type&) ctor
initial_suspend
To boldly go where no man has gone beforereturn_
void
final_suspend
~Task()
```

Figure 6

```
#include <print>
#include <string>
#include <coroutine>
#include <iostream>

struct Task{
  struct promise_type{
    std::string input_data{};

    Task get_return_object() noexcept{
      std::println("get_return_object");
      return Task{ *this };
    }
    void return_void() noexcept{
      std::println("return_void");
    }
    std::suspend_always initial_suspend()
    noexcept{
      std::println("initial_suspend");
      return {};
    }
    std::suspend_always final_suspend() noexcept{
      std::println("final_suspend");
      return {};
    }
    void unhandled_exception() noexcept{
      std::println("unhandled_exception");
    }
    std::suspend_always yield_value(
         std::string msg) noexcept{
      std::println("yield_value");
      //output_data = std::move(msg);
      return {};
    }
    auto await_transform(std::string) noexcept{
      struct Awaitable{
        promise_type& promise;
        bool await_ready() const noexcept{
          return true; // Says, yeah we are ready
           // we don't need to sleep. Just go on.
        }
        std::string await_resume() const
        noexcept{
          return std::move(promise.input_data);
        }
        void await_suspend(std::coroutine_handle<
          promise_type>) const noexcept{}
      };
      return Awaitable(*this);
    }
  };

  std::coroutine_handle<promise_type> handle{};

  explicit Task(promise_type& promise)
  : handle{ std::coroutine_handle<promise_type>
    ::from_promise(promise)}
  {
    std::println("Task(promise_type&) ctor");
  }
  ~Task() noexcept{
    if(handle)
      handle.destroy();
    std::println("~Task()");
  }
  void put(std::string msg){
    handle.promise().input_data = std::move(msg);
    if(!handle.done())
      handle.resume();
  }
};
Task coro_func(){
  std::cout << co_await std::string{};
  co_return;
}
int main(){
  auto task = coro_func();
  task.put("To boldly go where no man has gone "
           "before");
  return 0;
}
```

Listing 8 (cont'd)

Conclusion

We have implemented a few simple coroutines to explain the basic mechanics of low-level C++ coroutines. We learned how to implement generators. I think coroutines are important because they allow better resource utilization, reduce waiting time, and improve the scalability of applications. ■

```
#include <print>
#include <generator>

std::generator<int> makeFibonacciGenerator(){
  int i1{0};
  int i2{1};
  while(true){
    co_yield i1;
    i1 = std::exchange(i2, i1 + i2);
  }
  co_return;
}
int main(){
  auto fibo_gen = makeFibonacciGenerator();
  std::println("The first 10 numbers the "
               "Fibonacci sequence are : ");
    int i{0};
    for(auto f = fibo_gen.begin();
        f!=fibo_gen.end();++f){
      if(i  == 10)
        break;
      std::println("F[{}] = {}", i, *f);
      ++i;
  }
  return 0;
}
```

Listing 9

Dynamic Memory Management on GPUs with SYCL

Programming graphical processing units can speed up code, but you may lose access to standard library features. Russell Standish investigates a cross-platform accelerator API: SYCL.

Dynamic memory allocation is not traditionally available in kernels running on graphical processing units (GPUs). This work aims to build on Ouroboros, an efficient dynamic memory management library for CUDA applications, by porting the code to SYCL, a cross-platform accelerator API. Since SYCL can be compiled to a CUDA backend, it is possible to compare the performance of the SYCL implementation with that of the original CUDA implementation, as well as test it on non-CUDA platforms such as Intel's Xe graphics.

Introduction

Dynamic memory allocation is not traditionally part of the runtime environment of code running on accelerators, such as *graphical processing units* (GPUs), *signal processors* (DSPs) and *field programmable gate arrays* (FPGAs). Instead memory is allocated by the host code to fixed sizes prior to launching the kernel code that runs on the accelerator. This works well for many applications with fixed partitioning of space and time, such as finite element methods of partial differential equations, or algorithms based on matrix multiplication.

However, some applications, such as graph algorithms, or agent based models, require memory to be dynamically partitioned between the objects of the computation. Stopping the kernel, resizing memory allocations and relaunching is simply unfeasible in those circumstances. Instead, what is needed is to preallocate a chunk of memory on the host to act as a *heap*, and to run a heap allocation algorithm on the accelerator device.

There are two main C++ APIs for programming the host and kernel code. CUDA [Luebke08] is an extension to standard C++, and kernel code and host code are distinct. CUDA is developed by NVIDIA, and supports only GPUs manufactured by that company. SYCL [Reinders23], on the other hand, does not distinguish between host and kernel code, it is all standard C++. However, not all standard library features are available in a kernel context, which will be flagged by the compiler. Drivers are available for all major GPU types from Intel, AMD and NVIDIA, so SYCL is considered cross-platform.

At this point, it is worth mentioning a third option, also cross-platform, called OpenCL. It is an older C API, and consequentially less feature rich, and more difficult to use than CUDA or SYCL.

CUDA gained dynamic memory allocation in 2009 [NVIDIA], but is often considered slow and unreliable. Since then, a number of dynamic memory allocators have been proposed, mostly for CUDA, although one non-open source OpenCL implementation exists [Spliet14]. Details of these can be found in a survey by Winter and Mlakar [Winter21]. The basic strategy is to divide the preallocated block into chunks of different sizes, each of which go into a lock-free queue dedicated for the particular chunk size.

In the above survey, benchmarks indicate Ouroboros [Winter20], by the same authors, as being the most performant. Therefore this work took the CUDA Ouroboros code and translated it into SYCL, called Ouroboros-SYCL[1]. Since SYCL has implementations that target CUDA, we can directly compare the resulting performance of Ouroboros-SYCL with the original Ouroboros code on the same NVIDIA hardware, as well as provide results on non-NVIDIA hardware (Intel Xe graphics).

1 https://github.com/highperformancecoder/Ouroboros-SYCL

Porting the code to SYCL

As a way of getting started, I attempted to use the automatic CUDA to SYCL translator, called *SYCLomatic* [Liang24]. As might be expected for a project with the complexity of Ouroboros, this tool failed to generate compilable SYCL code, but managed to convert perhaps half of the code into SYCL equivalents, leaving constructs it failed to convert in the original CUDA form, and annotating some other parts it felt need further attention from the programmer to check the conversion was valid. In the end, whilst the code conversion was a good start, pretty much all of the automatically converted code was modified manually. The first reason for this is that CUDA always uses a 3D layout of processing elements, whereas SYCL allows for the possibility of 1, 2 or 3D layouts of processing elements, indicated by a template parameter. SYCLomatic always renders the translated code as 3D, to match the original source code, however, Ouroboros, being a library needs to handle arbitrary layouts. Thus code to extract, for example, a processing element's rank (known as *id* in SYCL) uses the `get_global_linear_id()` call, which returns a single number regardless of the dimensionality of the processing element layout.

In the process of getting the code to compile, SYCLomatic failed to convert atomic memory operations. SYCL has an atomic reference type, that wraps a variable, and allows a range of atomic operations, including the usual binary operation suspects (+,–,∧,∨,⊕,min,max), compare_and_swap. CUDA, by contrast has a set of atomic library functions, implementing the same operations. In the end, the simplest approach was to provide implementations of the CUDA library functions, implemented using SYCL atomic reference types.

The next issue was how to represent CUDA's global `threadIdx` and `blockIdx` variables. These refer to the coordinates (in CUDA's 3D space) of the thread (within its block) and the coordinates of the block. In SYCL terminology, a CUDA block is a *group*. An `nd_item` object contains all the information about the current thread's rank within its group, and its group's rank within the global range. However, the `nd_item` object is not available as a global function, but must be passed as a parameter into the stack frame where it is needed. A similar issue relates to I/O – CUDA has a `printf` statement available that is callable in any kernel function, whereas the SYCL equivalent is a `stream` object, usable like `std::cout` that must be created in the command group scope, and passed as a parameter to inner function scopes. In the Ouroboros-SYCL case, each class member function called in kernel code must take a template parameterised parameter (`Desc`) that has an item (an

Russell Standish Russell gained a PhD in Theoretical Physics, and has had a long career in computational science and high performance computing. Currently, he operates a consultancy specialising in computational science and HPC, with a range of clients from academia and the private sector. You can contact him at hpcoder@hpcoders.com.au

`nd_item` of arbitrary rank) and an out object that supports `operator<<` serialisation. Being a template parameter, users of the library can choose to define the rank of item, and use a `sycl::stream` or a dummy out object as appropriate.

It should be noted that Intel's oneAPI SYCL compiler (called DPC++) also provides an experimental free function `get_nd_item()`, and an experiment free function `printf()` that can be used for this purpose. These functions are proposed for a future SYCL standard.

Another point about the `sycl::stream` object is that it buffers the string data written to it, and the message is only written to the console when the stream object goes out of scope. Unfortunately, if the problem being diagnosed is a deadlock, or a crash, the stream object never goes out of scope, so any helpful debug messages written by way of this object will not be seen – a frustrating exercise indeed.

CUDA compute capability 7 introduced a `nanosleep` function for pausing threads for a specific period of time. Ouroboros uses this function to throttle threads demanding to allocate memory so that other threads freeing memory can catch up. This function would indeed be a useful optimisation technique, but is unavailable in the SYCL programming environment. Instead, all we can do is perform an `atomic_fence()`, which ensures that other threads catch up to the fence.

The final difficulty in converting the Ouroboros CUDA code had to do with the use of warp vote functions, which allowed multiple allocations to occur within one warp. In SYCL terminology, a warp is known as a *subgroup*, and corresponding group reduction algorithms exist, applied to subgroups, that are the equivalent of the warp functions. Unfortunately there is an issue. The CUDA equivalents take a mask, so that threads not participating in the group operation can be masked out by passing the results of `__activemask()`.

In SYCL, there is no real way of obtaining the active mask, and according to the standard, group operations block until all threads call the group operation function.

It would be more useful if group operations require all subgroups within a group to participate, and only those active threads should be required to call the group operation. So it should be possible to obtain the active mask by means of the following code:

```
auto sg=i.get_sub_group();
auto activeMask=sycl::reduce_over_group(
  sg,
  1ULL<<sg.get_local_linear_id(),
  sycl::bit_or<>()
);
```

Interestingly, when run on an Intel GPU, or on the CPU, this code runs as expected, and generates the active mask. But when run on an NVIDIA GPU, this code deadlocks, both with Intel's oneAPI, and with the AdaptiveCpp compiler, unless all threads in the subgroup are active.

Methods

Ouroboros comes with a driver program for each of the six alternative heap algorithms, *chunk*, *page*, *virtual array chunk*, *virtual array page*, *virtual list chunk* and *virtual list page*. Arguments passed to the driver program specify the data size to be allocated, and number of allocations to be allocated in parallel. Finally, the program iterates ten times through allocating memory, writing some data, checking that the data is correct when read back and then freeing the memory. The average time for performing the allocations and frees is calculated.

In this work, one additional change was made to the original code, aside from a trivial change to reduce the total amount of heap space available in order to fit on device available to the author. SYCL implementations typically compile the kernel code into an intermediate representation, such as SPIR-V [Kessenich18], and transpiling this into the native machine code of the accelerator occurs in a *just in time* (JIT) fashion when the kernel is first launched. As a result, there can be a big disparity between the time recorded for the first iteration, and the times recorded for subsequent iterations. So the code was modified to report the average over all iterations, and the average over all but the first iteration (ie *subsequent iterations*). This allow a more apples-to-apple comparison between the CUDA implementation and the SYCL implementation.

The actual code can be found in the Ouroboros-SYCL GitHub repository[2]. The SYCL code is available in the *master* branch, and the original optimised CUDA code in the *cuda-ouroboros* branch.

Hardware:

1. Dell Precision 7540 laptop with i9-9880H CPU @ 2.3GHz and NVIDIA Quadro T2000 GPU.
2. Asus NUC 13, i5-1340P CPU with integrated Iris Xe graphics GPU

Software:

1. Intel oneAPI 2025.1 (icpx compiler)
2. Codeplay's oneAPI for NVIDIA GPUs plugin
3. CUDA 12.8
4. Adaptive C++, compiled from source code[3], commit f336ab84. Adaptive C++ was previously known as HipSYCL.

After running cmake, it is necessary to insert the compiler manually using ccmake: for oneAPI the compiler is `icpx`, and you need to add the option `-fsycl`, as well as for NVIDIA use, the option `-fsycl-targets=nvptx64-nvidia-cuda`. For Adaptive C++, the compiler is acpp, and doesn't require any special command line flags.

The use of these compilers allows the comparison of the translated code with original code, running on the same hardware. Adaptive C++ targets CUDA's PTX machine, so is closer to what the CUDA compiler nvcc produces. However, it does the final of intermediate code to ptx in a *just-in-time* fashion, so for a proper comparison, we should compare only measured alloc/free times after the first iteration. Similarly, Codeplay's plugin performs JIT compilation of intermediate code to ptx.

The optimised Ouroboros code has a few instances of embedded PTX code, also making use of `nanosleep()`, and the ability to mask warp voting functions by the active mask. To make the comparison fair with the SYCL versions, I created a *deoptimised version*, with the embedded code replaced by high level code equivalents, `nanosleep` replaced by an `atomic_fence`, and the code using warp functions replaced by the simplified code used in the SYCL versions. This code can be found in the *deoptimised* branch of the Ouroboros-SYCL repository.

Results

In interpreting the algorithm results, it should be noted that the heap is divided into *chunks* of different sizes, and the allocation requests are served as *pages* from within each chunk.

Note that the Adaptive C++ compiled code would struggle as the number of threads increased, with loops timing out or becoming deadlocked.

The raw results files are available in the supplementary materials [Standish25], and a Ravel[4] file with the data loaded to assist in the data analysis.

Page allocator

The simplest allocator is the page-based allocator, where pages of fixed size are allocated from a queue. Total heap memory is divided amongst the queues, each queue managing a different page size. Being the simplest allocator, it is also the fastest, but suffers more from fragmentation than the other more sophisticated schemes. Figure 1 (on next page) shows the average subsequent timings of allocations as a function of allocation size when 1024 threads are attempting to simultaneously allocate, and the timings as a function of number of threads simultaneously attempting to allocate 1000 bytes.

2 https://github.com/highperformancecoder/Ouroboros-SYCL
3 https://github.com/AdaptiveCpp/AdaptiveCpp
4 Ravel is a revolutionary product for interactively analysing multidimensional data, available from https://ravelation.net

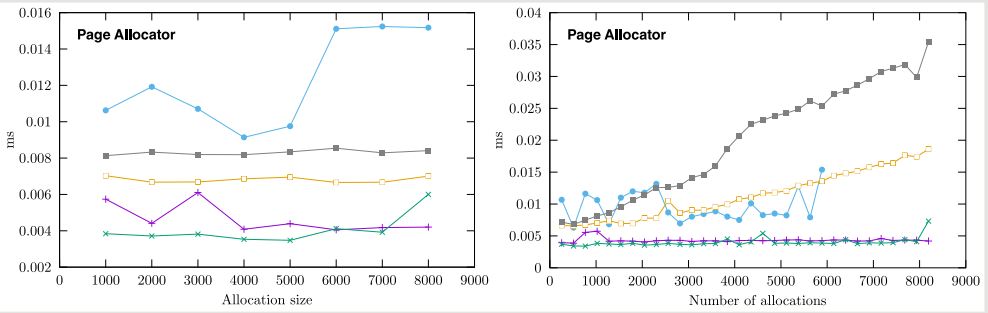


Figure 1

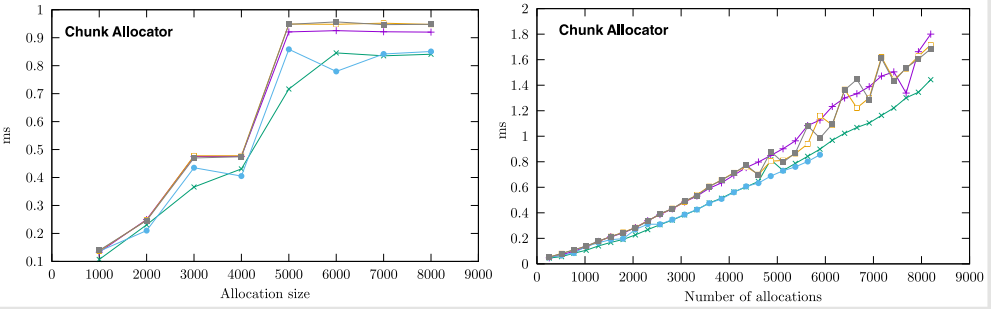


Figure 2

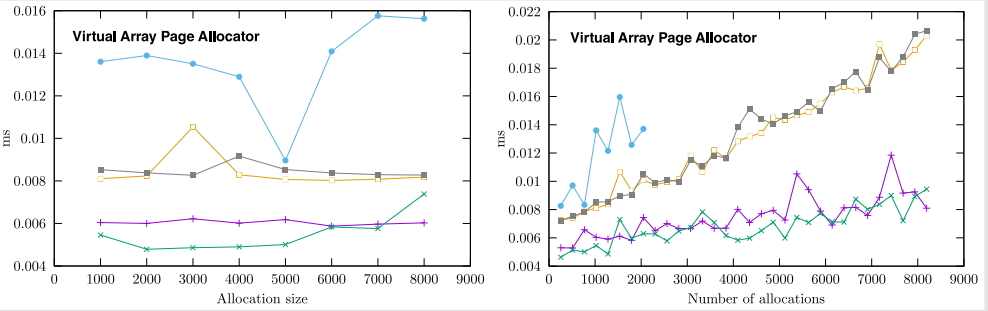


Figure 3

The performance of the SYCL code ends up being about half that of the CUDA code. Interestingly, the attempt to deoptimise the CUDA code to make it more comparable to the SYCL version only seem to make it more performant, if anything.

Chunk allocator

The chunk allocator maintains queues of chunks that have free pages, first obtaining a chunk index, then scanning the chunk for free pages. It is a more complex algorithm, but queue sizes are smaller.

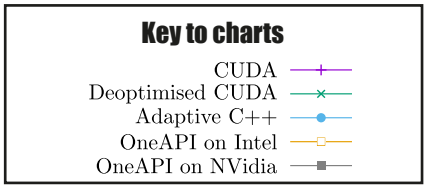
Figure 2 shows the average time to allocate memory for different allocation sizes. The allocator is implemented as a linked list of chunk queues, each queue managing chunks sized according to powers of two. You can see the effect of having to walk through this link list as the chunk size increases. On the right, you can see the effect of thread contention as more threads attempt to allocate chunks simultaneously. We can conclude from these figures that not only does the SYCL version work (data is written to the allocated chunks and checked), but that the implementation performance is broadly in line with the original Ouroboros implementation when run on the same hardware.

Virtualised array and list allocators

Ouroboros also introduces virtual queues, which reduce queue sizes even further. Figures 3–6 (above and on page 23) show the equivalent results for the virtualised versions of the page and chunk allocators.

Conclusion

The results indicate that the conversion of Ouroboros's CUDA-based code into SYCL was successful, and within a factor of 2 performance of the original code for the faster page-based algorithms, and within statistical noise of the performance of the chunk-based algorithms using Intel's oneAPI toolset. Adaptive C++ unfortunately suffered from timeouts and deadlocks, which may limit the use of this code with this compiler. As it hasn't yet fully implemented the SYCL 2020 standard, perhaps this is a matter of time.



The exercise also highlighted some deficiencies of SYCL with respect to CUDA – in particular the need for global access to a thread's `nd_item`, a global `printf` function for debugging purposes (both of these are proposed as experimental additions to SYCL in the oneAPI toolset) and the need for group reduction algorithms to be masked by the active threads only. ■

References

[Kessenich18] John Kessenich, Boaz Ouriel, and Raun Krisch. SPIR-V specification. *Khronos Group*, 3:17, 2018.

[Liang24] Wentao Liang, Norihisa Fujita, Ryohei Kobayashi, and Taisuke Boku. Using SYCLomatic to migrate CUDA code to oneAPI adapting NVIDIA GPU. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pages 192–193. IEEE, 2024.

[Luebke08] David Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pages 836–838. IEEE, 2008.

[NVIDIA] NVIDIA. CUDA C++ programming guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide (accessed 1st April 2025).

[Reinders23] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL*. Springer Nature, 2023.

[Spliet14] Roy Spliet, Lee Howes, Benedict R Gaster, and Ana Lucia Varbanescu. KMA: A dynamic memory manager for OpenCL. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 9–18, 2014.

[Standish25] Russell K. Standish. Ouroboros-SYCL. https://osf.io/2zwrt/, 2025.

[Winter20] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. Ouroboros: virtualized queues for dynamic memory management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[Winter21] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. Are dynamic memory managers on GPUs slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–233, 2021.9

15 Different Ways to Filter Containers in Modern C++

Filtering items from a container is a common situation. Bartłomiej Filipek demonstrates various approaches from different versions of C++.

Do you know how many ways we can implement a filter function in C++? While the problem is relatively easy to understand – take a container, copy elements that match a predicate and the return a new container – it's good to exercise with the C++ Standard Library and check a few ideas. We can also apply some modern C++ techniques, including C++23. Let's start!

The problem statement

To be precise by a *filter*, I mean a function with the following interface:

```
auto Filter(const Container& cont,
            UnaryPredicate p) {}
```

It takes a container and a predicate, and then it creates an output container with elements that satisfies the predicate. We can use it like the following:

```
const std::vector<std::string> vec{
  "Hello", "**txt", "World", "error", "warning",
  "C++", "****" };
auto filtered = Filter(vec, [](auto& elem) {
  return !elem.starts_with('*'); });
  // filtered should have "Hello", "World",
  // "error", "warning", "C++"
```

Writing such a function can be a good exercise with various options and algorithms in the Standard Library. What's more, our function hides internal things like iterators, so it's more like a range-based version.

Let's start with the first option.

Good old raw loops

While it's good to avoid raw loops, they might help us to fully understand the problem. For our filtering problem we can write the following code:

```
// filter v1
template <typename T, typename Pred>
auto FilterRaw(const std::vector<T>& vec, Pred p)
{
  std::vector<T> out;
  for (auto&& elem : vec)
    if (p(elem))
      out.push_back(elem);
  return out;
}
```

Simple yet very effective.

Please notice some nice things about this straightforward implementation.

- The code uses `auto` return type deduction, so there's no need to write the explicit type (although it could be just `std::vector<T>`).
- It returns the output vector by value, but the compiler will leverage the copy elision (named return value optimization – NRVO), or move semantics at worse.

Since we're considering raw loops, we need can take a moment and appreciate the range-based `for` loops that we get with C++11. Without this functionality our code would look much worse:

```
// filter v1 - old way
template <typename T, typename Pred>
std::vector<T> FilterRawOld(const std::vector<T>&
    vec, Pred p) {
  std::vector<T> out;
  for (typename std::vector<T>::const_iterator it
      = begin(vec); it != end(vec); ++it)
    if (p(*it))
      out.push_back(*it);
  return out;
}
```

And now let's move to something better and see some of the existing `std::` algorithms that might help us with the implementation.

Filter by std::copy_if

`std::copy_if` is probably the most natural choice. We can leverage `back_inserter` and then push matched elements into the output vector.

```
// filter v2
template <typename T, typename Pred>
auto FilterCopyIf(const std::vector<T>& vec,
    Pred p) {
  std::vector<T> out;
  std::copy_if(begin(vec), end(vec),
    std::back_inserter(out), p);
  return out;
}
```

std::remove_copy_if

We can also do the reverse:

```
// filter v3
template <typename T, typename Pred>
auto FilterRemoveCopyIf(const std::vector<T>&
    vec, Pred p) {
  std::vector<T> out;
  std::remove_copy_if(begin(vec), end(vec),
    std::back_inserter(out), std::not_fn(p));
  return out;
}
```

Depending on the requirements, we can also use `remove_copy_if`, which copies elements that do not satisfy the predicate. For our implementation, I had to add `std::not_fn` to reverse the predicate.

One remark: `std::not_fn` has been available since C++17.

The famous Remove Erase idiom

One thing to remember: `remove_if` doesn't remove elements; it only moves them to the end of the container. So we need to use `erase` to do the final work:

Bartłomiej Filipek is a C++ developer and author, regularly writing about modern C++ at cppstories.com. He is the author of the books *C++17 in Detail*, *C++ Lambda Story*, and *C++ Initialization Story*. He currently works at Intel (since 2024) as an AI Software Development Engineer, focusing on C++ and GPU/ML technologies. Previously, he developed graphics and document-editing software at Xara. Contact: bartek@cppstories.com

```
// filter v4
template <typename T, typename Pred>
auto FilterRemoveErase(const std::vector<T>& vec,
    Pred p) {
  auto out = vec;
  out.erase(std::remove_if(begin(out), end(out),
      std::not_fn(p)), end(out));
  return out;
}
```

Here's a minor inconvenience. Because we don't want to modify the input container, we had to copy it first. This might cause some extra processing and is less efficient than using `back_inserter`.

Adding some C++20

After seeing a few examples that can be implmented in C++11, we can see a convenient feature from C++20: `erase_if`:

```
// filter v5
template <typename T, typename Pred>
auto FilterEraseIf(const std::vector<T>& vec,
    Pred p) {
  auto out = vec;
  std::erase_if(out, std::not_fn(p));
  return out;
}
```

This function is superior to the remove/erase iodiom, as you can just use a single function.

One minor thing, this approach copies all elements first. So it might be slower than the approach with `copy_if`.

Adding some C++20 ranges

C++20 also brought us powerful ranges and range algorithms, and we can use them as follows:

```
// filter v6
template <typename T, typename Pred>
auto FilterRangesCopyIf(const std::vector<T>&
    vec, Pred p) {
  std::vector<T> out;
  std::ranges::copy_if(vec,
    std::back_inserter(out), p);
  return out;
}
```

The code is super simple, and we might even say that our `Filter` function has no point here, since the Ranges interface is so easy to use in code directly.

Making it more generic

So far, I have shown you code that operates on `std::vector`. But how about other containers?

Let's try and make our `Filter` function more generic. This is easy with `std::erase_if`, which has overloads for many Standard containers:

```
// filter v7
template <typename TCont, typename Pred>
auto FilterEraseIfGen(const TCont& cont, Pred p)
{
  auto out = cont;
  std::erase_if(out, std::not_fn(p));
  return out;
}
```

And another version for ranges.

```
// filter v8
template <typename TCont, typename Pred>
auto FilterRangesCopyIfGen(const TCont& vec,
    Pred p) {
  TCont out;
  std::ranges::copy_if(vec,
      std::back_inserter(out), p);
  return out;
}
```

It can already work with other containers, not just with `std::vector`:

```
// filter v9
template <typename T, typename = void>
struct has_push_back : std::false_type {};
template <typename T>
struct has_push_back<T,
  std::void_t<
    decltype(std::declval<T>().push_back(
        std::declval<typename T::value_type>()))
    >
  > : std::true_type {};
template <typename TCont, typename Pred>
auto FilterCopyIfGen(const TCont& cont, Pred p) {
  TCont out;
  if constexpr(has_push_back<TCont>::value)
    std::copy_if(begin(cont), end(cont),
       std::back_inserter(out), p);
  else
    std::copy_if(begin(cont), end(cont),
      std::inserter(out, out.begin()), p);
  return out;
}
```

Listing 1

```
std::set<std::string> mySet{
  "Hello", "**txt", "World", "error", "warning",
  "C++", "****"
};
auto filtered = FilterEraseIfGen(mySet,
    [](auto& elem) {
  return !elem.starts_with('*');
});
```

On the other hand, if you prefer not to copy all elements upfront, we might need more work.

Generic copy_if approach

The main problem is that we cannot use `back_inserter` on associative containers, or on containers that don't support the `push_back()` member function. In that case, we can fallback to the `std::inserter` adapter.

That's why one of a possible solution is to detect if a given container supports `push_back` (see Listing 1).

I used a technique available up to C++17 with `void_t` and SFINAE (read more from one of my blog posts [Filipek21a]), but since C++20, we have been able to leverage concepts and make the code more straightforwad:

```
template <typename T>
concept has_push_back = requires(T container,
    typename T::value_type v) {
  container.push_back(v);
};
```

You can seee more on my blog [Filipek22].

More C++20 concepts

We can add more concepts, and restrict other template parameters. For example, if I write:

```
auto filtered = FilterCopyIf(vec, [](auto& elem,
    int a) {
  return !elem.starts_with('*');
});
```

In the above code, I tried to use two arguments for the unary predicate. In Visual Studo I'm getting the error message in Figure 1 (next page).

Not very helpful...but then after a few lines, we have a clear reason for the errors:

```
error C2780: 'auto main::<lambda_4>::operator
()(_T1 &,int) const': expects 2 arguments - 1
provided
```

We can experiment with concepts and restrict our predicate to be `std::predicate`, an existing concept from the Standard Library.

In our case, we need a function that takes one argument and then returns a type convertible to `bool`.

```
C:\Program Files (x86)\Microsoft Visual
Studio\2019\Community\VC\Tools\MSVC\14.28.29333\
include\algorithm(1713,13): error C2672:
'operator __surrogate_func': no matching
overloaded function found
1>  C:\Users\Admin\Documents\GitHub\articles\
filterElements\filters.cpp(38): message : see
reference to function template instantiation
'_OutIt std::copy_if<std::_Vector_const_
iterator<std::_Vector_val<std::_Simple_types<_
Ty>>>,std::back_insert_iterator<std::vector<_
Ty,std::allocator<_Ty>>>,Pred>(_InIt,_InIt,_
OutIt,_Pr)' being compiled
1>          with
```

Figure 1

```
// filter v10
template <typename T,
    std::predicate<const T&> Pred>   // <<
auto FilterCopyIfConcepts(
    const std::vector<T>& vec, Pred p) {
  std::vector<T> out;
  std::copy_if(begin(vec), end(vec),
      std::back_inserter(out), p);
  return out;
}
```

and then the problematic code:

```
auto filtered = FilterCopyIfConcepts(vec,
    [](auto& elem, int a) {
  return !elem.starts_with('*');
});
```

This results in the following message:

```
1>  filters.cpp(143,19): error C2672:
'FilterCopyIfConcepts': no matching overloaded
function found
1>  filters.cpp(143,101): error C7602:
'FilterCopyIfConcepts': the associated
constraints are not satisfied
```

It's a bit better, as we have messages about our top-level function rather than the internals, but it would be great to see why and which constraint wasn't satisfied.

Making it parallel?

Since C++17, we have also had parallel algorithms, so why not add them to our list?

As it appears `std::copy_if par` is not supported in Visual Studio, and this problem is a bit more complicated. We'll leave this topic for now and try to solve it another time.

For completness we can write the following naive code:

```
// filter v11
std::mutex mut;
  std::for_each(std::execution::par, begin(vec),
      end(vec),
    [&out, &mut, p](auto&& elem) {
      if (p(elem))
      {
        std::unique_lock lock(mut);
        out.push_back(elem);
      }
    });
```

This is, of course, a naive version, and will make the process serialized. The topic is quite advanced, so please have a look at my other text and experiment (`filter v12`) [Filipek21b].

Direct filter support with ranges::filter_view, C++23

In C++23, we got `std::ranges::filter_view` and `std::views::filter`: the code is much simpler now (see Listing 2).

Adding ranges::to, C++23

What's more we can use `ranges::to` to automatically create a container.

```
// filter v13
template <typename T,
    std::predicate<const T&> Pred>
auto FilterRangesFilter(
    const std::vector<T>& vec, Pred p) {
  std::vector<T> out;
  for (const auto& elem : vec
       | std::views::filter(p))
    out.push_back(elem);
  return out;
}
```

Listing 2

```
// filter v14
template <typename T,
    std::predicate<const T&> Pred>
auto FilterRangesFilterTo(const std::vector<T>&
    vec, Pred p) {
  return vec | std::views::filter(p)
             | std::ranges::to<std::vector>();
}
```

Additionally, `ranges::to` works with any kind of container, and figures out an appropriate way to populate it. So it works with more than just `std::vector`.

Here's an example:

```
template <typename Cont, std::predicate<const
    typename C::value_type&> Pred>
auto FilterRangesFilterTo(const Cont& vec,
    Pred p) {
  return vec | std::views::filter(p)
             | std::ranges::to<Cont>();
}
```

C++23: Lazy Filtering with std::generator

All previous versions of `Filter` in this article return a **materialised container** – a `std::vector`, `std::set`, or something similar. That's often what we want, but sometimes it's more efficient to:

- avoid allocating a separate container,
- process elements **on the fly** (e.g. streaming input, large ranges), or
- combine filtering with another lazy pipeline.

C++23 adds `std::generator`, a coroutine-based type that models a range. We can use it to express a **lazy filter**:

```
template <typename T,
          std::predicate<const T&> Pred>
std::generator<const T&>
    FilterLazy(const std::vector<T>& vec, Pred p)
{
  for (const auto& elem : vec) {
    if (p(elem))
      co_yield elem;
  }
}
```

Usage is straightforward:

```
std::vector<std::string> vec{
  "Hello", "**txt", "World", "error", "warning",
  "C++", "****"
};
auto gen = FilterLazy(vec, [](const auto& s) {
  return !s.starts_with('*');
});
// Elements are produced lazily, on demand:
for (const auto& s : gen) {
  std::cout << s << '\n';
}
```

A few important properties of this approach:

- Lazy evaluation – elements are filtered only when you iterate the generator.
- No intermediate container – no extra allocation by default.

Summary

In this article, I've shown at least 15 possible ways to filter elements from various containers. We started from code that worked on `std::vector`, and you've also seen multiple ways to make it more generic and applicable to other container types. For example, we used `std::erase_if` from C++20, concepts, and even a custom type trait. We also used the 'holy grail' of C++23 in terms of `ranges::filter` and `ranges::to`.

See my code, with all the example, in this repository: https://github.com/fenbf/articles/blob/master/filterElements/filters.cpp

Back to you

- What other options do you see?
- What techniques do you prefer?

Let us know: join the discussion [reddit]. ■

References

[Filipek21a] Bartłomiej Filipek, 'How To Detect Function Overloads in C++17, std::from_chars Example', on *C++ Stories*, posted 13 June 2021 at https://www.cppstories.com/2019/07/detect-overload-from-chars/.

[Filipek21b] Bartłomiej Filipek, 'Implementing parallel copy_if in C++' on *C++ Stories*, posted 22 February 2021 at https://www.cppstories.com/2021/par-copyif/

[Filipek22] Bartłomiej Filipek, 'Simplify Code with if constexpr and Concepts in C++17/C++20', on *C++ Stories*, updated 8 August 2022 at https://www.cppstories.com/2018/03/ifconstexpr/

[reddit] '12 different ways to filter containers in modern C++' on reddit: @r/cpp at https://www.reddit.com/r/cpp/comments/l9z7rj/12_different_ways_to_filter_containers_in_modern_c/

This article was first published on Bartłomiej Filipek's blog (*C++ Stories*) on 12 June 2021 and has been reviewed for *Overload*. The original article is available at https://www.cppstories.com/2021/filter-cpp-containers/

Dynamic Memory Management on GPUs with SYCL (continued from page 19)

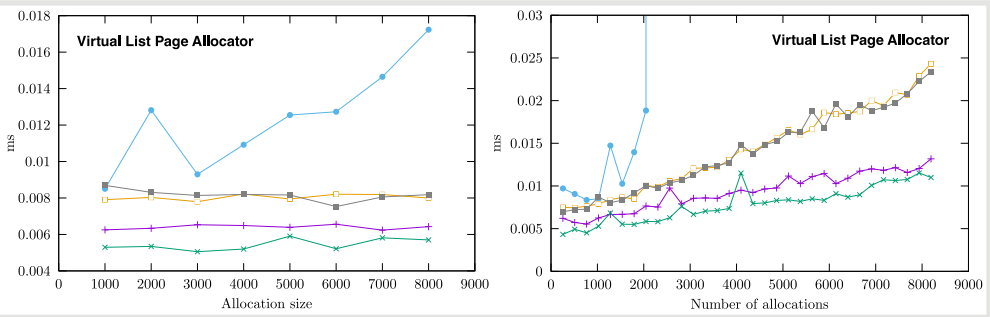


Figure 4

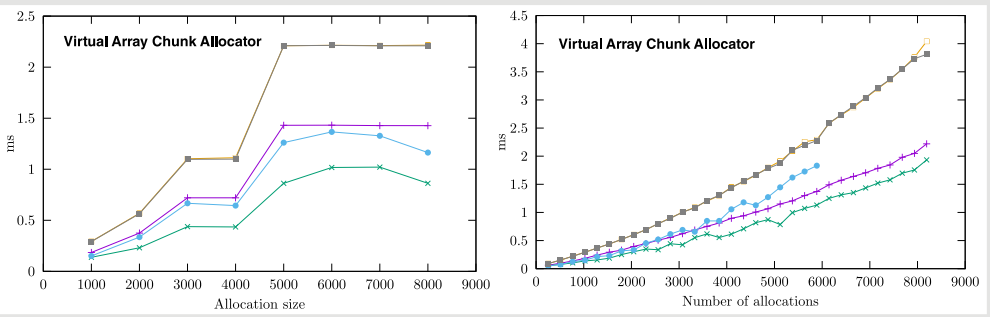


Figure 5

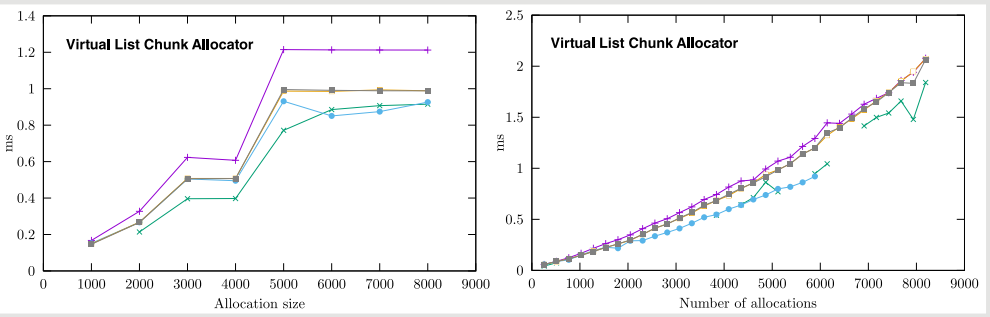


Figure 6

Key to charts

CUDA
Deoptimised CUDA
Adaptive C++
OneAPI on Intel
OneAPI on NVidia

Afterwood

Do you have a Christmas wish-list? Chris Oldwood writes a 'dear Santa' letter to share what he wants.

A Breakfast Bar,
Godmanchester,
Cambridgeshire, UK

Dear Santa,

Gosh, it's been a long time since I've written to you. To be honest, I'm not entirely sure about where you stand on letters from adults, so I'm taking a punt on you being open to at least listening to anyone who is willing to put a (metaphorical) pen to paper. I really hope you do still accept old fashioned letters and I've not got to present my desires in the form of a TikTok or WhatsApp voice note. Okay, yes, I did make a joke in this column three years ago [Oldwood22] that you probably only accept requests in the form of JIRA tickets these days, and that was mean, but you weren't the butt of the joke. In fact, it's because of that same group of short-sighted miscreants that I'm writing to you. But I'm getting ahead of myself.

First off, I want you to know that Christmas is still a magical time of year. Yes, there were a few years in my late teens when the prospect of being paid double-time and the pubs being open later than usual took priority, but luckily I found a partner who showed me that even though we'd grown-up, it didn't mean we had to stop getting into the spirit of the festivities. Once children came along, the traditional practice of turfing contractors out for two weeks to cut costs and avoid them slacking off became something to celebrate. That said, I do look back fondly on the first Christmas after Doom was released and accept that productivity probably reached an all-time low, so maybe Management do have a point.

Hopefully, I made amends the following year when I was the only technical person left in the office on Christmas Eve and answered the support line to be greeted by a frantic clergyman in the Netherlands. He'd just bought the company's desktop publishing software and was desperately trying to print out his beautifully designed Order of Service for Christmas Day but couldn't get it to talk to the printer. It was my first gig, this was the 16-bit era, and as a developer I'd never really done any technical support before. I decided to try and help anyway. An hour and half later the pages were rattling out of his new inkjet printer, and we had another satisfied customer. Hopefully, the congregation appreciated his newfound artistic talent, too.

Little did I know this would also be the start of a lifelong struggle with printers. The only thing more soul-destroying on Christmas Day than unwrapping a printer is being greeted with a toy where the batteries are not included, or switching on an unpatched computer or games console. I feel like "Dear Santa, please can I have a fully patched Xbox with fully patched Call of Duty for Christmas" really kills the vibe.

Despite being a long time ago I'm still hoping my gesture of goodwill towards the clergy has helped me cement a solid place on the 'Nice List' and bought me some kudos. I believe there are other reasons since then that would have helped me retain my place in your good books. For one, I write tests, and not just afterwards either, I typically do them before writing my production code!

Okay, this is not purely an act of altruism as their presence is equally of benefit to me in the short term, but they benefit my teammates when I'm no longer around. Likewise, I try and plug the gap of what the code alone cannot say by documenting unobvious processes or the rationale behind architecture and design decisions to avoid people having to second guess my thoughts. Do they ever read it? Who knows. At a bare minimum, I'm always grateful to myself when I turn out to be the recipient.

I try my hardest to fix things when they're broken, whether that be the product, tooling, or delivery process. I learned the hard way that if you're always putting out fires, you never get the time to stop them happening in the first place, unless you place a genuine emphasis on quality. I appreciate that not everything can easily be fixed but there are always lots of small things which, when fixed together, can lead to a significant reduction in friction (marginal gains).

Okay Santa, I know what you're thinking, I'm only calling out the good stuff, but what about that time recently when I banjaxed the entire UAT environment because I didn't really think about the change I was making? Or that time I rebooted all the production servers because I wasn't paying attention after restoring the database to a test environment? These were undoubtedly mistakes but they were not borne out of malice. If I am guilty of anything, it's probably not maximising shareholder value in the short-term because I put my efforts into enabling sustainable delivery over the longer term.

And that brings me to my Christmas wish. While 'world peace' is always a venerable choice I want to bring the current obsession with so called 'artificial intelligence' to your attention in the hope that you can rein it in. Every company seems to have been taken in by the snake oil salesman such that every project, tool, blog post, conference talk, meet-up, etc. is about how we shoe-horn it in without questioning whether it even makes sense. My JIRA tickets and pull requests are now awash with bot generated 'analysis' which I have to wade through to find the valuable insights from my human colleagues. And this is all before we consider the wider implications of copyright theft, the impact on the environment, creative financing, reinforcement of cultural stereotypes and biases, mass lay-offs, and ultimately the continued rise in power for a few individuals only looking after number one.

Like the Luddites before me, this is not about being against the technology *per se*, it's about the way it's being abused and how this modern gold rush is fuelling the wrong kind of growth. There is undoubtedly great value in machine learning and LLMs, but let's not kid ourselves that, as it stands, the benefits are going to be evenly distributed, despite what it says in the marketing brochure. So, Santa, it's time for Jacob Marley to rise again, but this time he needs to visit the self-obsessed tech bros as we're heading for a steel shortage due to the rate at which they're forging their chains.

Yours,

A Disenchanted Programmer

Reference

[Oldwood22] Chris Oldwood (2022) 'Afterwood', *Overload*, 30(172), available at https://accu.org/journals/overload/30/172/oldwood/.

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

accu

Professionalism in Programming

Professional development
World-class conference

Printed journals
Email discussion groups

Individual membership
Corporate membership

Visit accu.org for details