overload 185

FEBRUARY 2025 £4.50

Using Senders/ Receivers

Lucian Radu Teodorescu demonstrates how senders/receivers will be used in C++26 to generate multithreaded code

Bit Fields, Byte Order and Serialization

Wu Yongwei explores issues to be aware of when network packets are represented as bit fields

Valgrind's Dynamic Heap Analysis Tool: DHAT

Paul Floyd explains what this heap analysis tool is and how to use it

Afterwood

Chris Oldwood tells us why he prefers learning in person

A magazine of ACCU

ISSN: 1354-3172

accu conference 2025

by programmers, for programmers, about programming

Tuesday 1st April
to
Friday 4th April

- 4 keynote speakers:
  Anastasia Kazakova
  Khalil Estell
  Daisy Hollman
  Matt Godbolt
- 54 presentations
- 3 lightning talk sessions
- Conference dinner

Pre- and post-conference workshops:

1 x two-day online on 29th and 30th March 2025
4 x one-day in person at the venue on 31st March 2025
2 x one-day online on 12th April 2025

To find out more and to book,
visit accuconference.org

February 2025
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

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

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Alison Peck – in the greenhouse at Dyffryn Gardens (a National Trust property), Cardiff, Wales.

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* 186 should be submitted by 1st March 2025 and those for *Overload* 187 by 1st May 2025.

All the Information is on the Task

Instructions can be useful or infuriating, Frances Buontempo wonders how to give and follow directions.

As the winter drags on, I have spent too much time watching television so haven't written an editorial. In particular, Junior Taskmaster [IMDB] has been on recently. Watching 'live' TV probably proves I'm getting old, as well as wasting my life. Nonetheless, if you're not aware of it, let me explain. The original Taskmaster [Wikipedia] is hosted by Alex Horne and Greg Davies. The contestants, all celebrities and usually comedians, are set tasks. They are awarded points and the contestant with the most points at the end wins. The tasks are very silly, and often lateral thinking wins out. Frequently, the contestants query the tasks, and are told, "All the information is on the task." Which almost never helps. Junior Taskmaster is hosted by Rose Matefeo and Mike Wozniak and has children rather than celebrities as contestants. The children's insistence on fair play gives the new series a different edge, but their imagination is amazing. One task involved moving a sand castle from a podium labelled 'A' to a podium labelled 'B'. I wondered if moving the podiums side by side might help, and a child tried this. Two children were even more sensible, just peeling the labels off and switching those. Lateral thinking often provides new and sometimes simpler ways of solving a problem.

You have probably been tasked with something which seems almost impossible or immensely tedious before. I started out as a maths teacher after university and set a pupil lines once. Rather than writing out the lines by hand they got a computer to generate a printout. Fine by me; they had done as requested, and showed some initiative. Automating can sometimes deal with the tedious, but the impossible is a different challenge. I recall a couple of interviews where I needed to stall slightly for thinking time. One involved live coding, which makes a change from using a white board to reverse a linked list. However, I wasn't 100% sure how to approach the question, which involved spotting palindromes. Not a difficult problem, but in an interview situation my brain tends to freeze up and I wasn't sure what I was allowed to use. I started, as I often do, by writing a test. Using `assert`. For an empty string, with a function that only returned `false`. The interviewer was deeply unimpressed, and pointed out my code didn't work, and all the information was in the question. Explaining I often started like that when using TDD didn't seem to help. The interviewer simply looked bemused. I managed the required function in the end. Starting with a very simple case helped me start thinking straight, though someone not getting writing a failing test first was off putting. Another interview question involved a brain teaser. I don't recall the precise details, but it involved putting pennies on a table and the person who put the last coin down either won or lost. Coins weren't allowed to overlap, and I think you had to say if you would go first or second. I had no idea how to start thinking it through, so asked probing questions about the size of coins and table. If a coin is as big as a table, you can only put one down. I suspect the interviewer wasn't impressed by me starting with edge cases, trying to flush out the specific details. But you need to start thinking somewhere.

Have you ever picked up a task from a tracking system, like Jira, and got stuck immediately? In theory, if you have backlog grooming/refinement sessions, everyone on the team should be able to understand what a task requires. And yet, it is still possible to get to some work and find things have changed, or assumptions no longer hold. Seb Rose wrote about this in his 'User Stories and BDD' series. In 'Part 2, Discovery' [Rose23]. He said:

> As professionals, we are paid to have answers. We feel deeply uncomfortable with uncertainty and will do almost anything to avoid having to admit to any level of ignorance.

Finding the uncertainty can be useful though. He goes on to talk about deliberate discovery and how to spot questions and unclear parts as well as splitting stories into manageable chunks. If a task or Jira has some example cases, or even if you have actual BDD automation tests to start coding against, you are much less likely to find yourself staring at the task wondering where to begin. In this case, all, or at least enough, information will be on the task. An example is often clearer than a Jira.

I heard a talk recently by a business person about how they wrote Jiras. Their team had a template with several sections, like acceptance criteria and so on, but they frequently forgot sections. Their solution was to use GenAI to write the tickets. The thought of this instantly horrified me. If the team subsequently talked through the Jiras I could see it working, but again having a list of what's required doesn't always mean the tasks make sense. Have you ever given someone instructions and they somehow miss the point completely? No matter how clear and precise you try to be, there is always room for misunderstanding. I recall a tale of a child making his Mum a cup of tea. Said child knew he had to boil the kettle, but thought it would be more efficient to put a teabag in the kettle while it boiled. A cup of 'tea' was made, but probably wasn't very tea flavoured. Spelling out the precise steps, in order, might avoid such creative thinking, but is very hard to do. There's usually a balance point. If a recipe says "Make a pastry case" but you don't know how to make pastry that won't be much help. Whereas, if the recipe spells out what a gram or milliliter are, that will distract from the baking instructions. An imperative set of instructions will make assumptions about a common understanding of words and instructions. "Boil a kettle" does not mean heating a kettle until it reaches boiling point. "Run the tests" should mean checking they pass, and taking appropriate action for any failures. Trying to communicate how to achieve something is hard, and often requires some back and forth.

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

The back and forth conversation necessitates people being able to communicate. Sometimes that is not possible. For example, if you write documentation, the chances are you will never meet are many people who read your instructions. You can get a friend or colleague to read through your first drafts. You might also be able to read through yourself, trying to misunderstand everything you have written, searching for potential misunderstandings or confusion. You might find you can write a script or automate some of the steps. Sometimes explaining to a computer is easier than explaining to a human.

Documentation crops up in various places. Maybe for a new machine or perhaps a game. Lots of machines no longer come with documentation, in particular mobile phones or laptops. Last time I bought a laptop, I had to search the internet to find out where the on button was. Nonetheless, you do still get written instructions, for example for games. And sometimes they are incomprehensible, so you need to attempt to play and decide amongst yourselves what to do under various circumstances. Some games don't come with full instructions. You might find a settings menu telling you key bindings like 'W', 'A', 'S', 'D' for up, left, down, right respectively. Figuring out what the rules are and how to score after that is another matter. I'm currently trying to prepare a talk for the ACCU conference [Buontempo25] about reinforcement learning (RL). RL is a type of machine learning where agents take actions in an environment, using trial and error to 'learn'. Rewards or penalties reinforce actions, and agents try to maximize rewards over time. For example, playing an arcade game and trying to get a high score. You can tell the agent the possible moves, WASD, and track the environment, letting the agent learn how to play the game. Deep Mind produced a paper showing how to train an agent using the pixels on screen to describe the environment [Mnih13]. Plug the agent and environment into an RL framework and watch your machine learn to play PacMan or similar over time [Gymnsasium (for example)]. Or wait for me to find a simple way to explain how to code the reinforcement learning up from scratch. Deep Mind's reinforcement learning, called Deep Q-Learning, did not need all the information upfront. The algorithm discovered how to play to get a good score by experimentation.

Writing code is often an iterative process, at least in terms of discovering the requirements. The code itself may be more declarative than iterative, or might even be recursive. I dip into functional languages from time to time, and can feel my brain starting to hurt/expand/change viewpoints while I get re-familiarised with recursive approaches. For example, you may see code for a sort along the lines of

```
merge_sort(A, start, end):
  if start<end
    mid = (start+end)/2
      merge_sort(A, start, mid)
      merge_sort(A, mid+1, end)
      merge(A, start, mid, end)
```

The `merge` function is left as an exercise for the reader. If you are familiar with merge sort, you will recognize this pseudocode. However, do you remember the first time you encountered code like this? How do you even start thinking this through? We've probably all seen jokes like the dictionary definition of recursion saying "see recursion". How do you start? All the information may be in the pseudocode, but you might need to rewire your brain slightly to understand. All the information is in the code, but that doesn't always help. And sometimes, some of the information is in a config file. Or more than one config file. Or replaced upfront by a setting in a database. So, we have two extremes: first a short piece of code in one place (apart from the `merge` function, sorry!) and another codebase with parts scattered in various places. Both can be hard to understand but for very different reasons. Figuring out how to understand a new codebase is a topic in itself. If you want some ProTips, watch Jonathan Boccara's ACCU 2019 conference talk, '10 Techniques to Understand Code You Don't Know' [Boccara19]. He talks about exploring, reading and understanding code. The exploring ideas start by finding where and how to experiment with input and outputs, whether a UI framework or log files or unit tests. We tend to learn by experimenting and discovering. Just staring at the merge-sort might not be enough to figure out what's going on. Finding a way to play with the code is more helpful. Or even, trying to sort some playing cards by following the instructions in the code can be useful.

Now, following instructions without thinking might prove that a set of instructions fulfill the requirements. That doesn't mean you have understood why the recipe works. I spent some time last year trying to solve the Rubik's cube. A friend set up a discussion group, sending videos and instructions to help. I did finally manage to solve the cube, but I would have to follow instructions to do this a second time. I know full well I don't fully understand why certain sequences of moves work, and I often have the orientation incorrect and end up moving the wrong pieces. Hopefully, I will eventually form a mental model, allowing me to think through what I need to do. Next time someone tells you "All the information is on the task" or tells you to "Read the question" in an exam, feel free to experiment and find out what happens. That's how we learn. You might discover something, or come out with a clever solution, you never know. In fact, here's a challenge. An Overload editorial requires two pages of writing for the front of the magazine. An editorial should be an opinion piece, or relevant to something topical, which as you know I never manage. If you want to try your hand, please get in touch. Task: 2,000 words or so, on a topic of your choice. Send it to me, and we'll see what the review team thinks. Over to you.

References

[Boccara19] Jonathan Boccara, '10 Techniques to Understand Code You Don't Know', *ACCU 2019*, available at https://www.youtube.com/watch?v=tOOK-VsWU-I

[Buontempo25] Frances Buontempo 'An introduction to reinforcement learning: Snake your way out of a paper bag', talk to be delivered at ACCU 2025, abstract available at: https://accuconference.org/2025/session/an-introduction-to-reinforcement-learning-snake-your-way-out-of-a-paper-bag

[IMDB] Junior Taskmaster: https://www.imdb.com/title/tt34234603/

[Gymnasium] https://gymnasium.farama.org/

[Mnih13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, 'Playing Atari with Deep Reinforcement Learning' NIPS Deep Learning Workshop 2013

[Rose23] Seb Rose, 'Part 2, Discovery' in *Overload* 31(178):4-5, December 2023 https://accu.org/journals/overload/31/178/overload178.pdf#page=6 and https://accu.org/journals/overload/31/178/rose/

[Wikipedia] Taskmaster: https://en.wikipedia.org/wiki/Taskmaster_(TV_series)

If you read something that you particularly enjoy, you disagree with or that just makes you think, why not put pen to paper (or finger to keyboard) and tell us about it? We'd love to hear from you!

Using Senders/Receivers

C++26 will introduce senders/receivers. Lucian Radu Teodorescu demonstrates how to use them to write multithreaded code.

This is a follow-up to the article in the previous issue of *Overload*, which introduced the upcoming C++26 senders/receivers framework [WG21Exec]. While the previous article focused on presenting the main concepts and outlining what will be standardized, this article demonstrates how to use the framework to build concurrent applications.

The goal is to showcase examples that are closer to real-world software rather than minimal examples. We address three problems that can benefit from multi-threaded execution: computing the Mandelbrot fractal, performing a concurrent sort, and applying a graphical transformation to a set of images.

All the code examples are available on GitHub [ExamplesCode]. We use *stdexec* [stdexec], the reference implementation for the senders/receivers proposal. Additionally, some features included in the examples are not yet accepted by the standard committee, though we hope they will be soon.

Before we get started

Before diving into more realistic examples, let's begin with a minimal example to set the stage. The code in Listing 1 prints "Hello, concurrency!" from a thread that is different from the main thread.

The code is roughly equivalent to:

```
std::thread{[]
  { printf("Hello, concurrency!\n"); }}.join();
```

Here, we acquire a thread from the system scheduler and execute the given lambda on that thread, which prints the message to the standard output.

The scheduler acts as a handle to an *execution context* – an entity that owns threads of execution, such as CPU or GPU threads. The system scheduler represents the default execution context on the current system, presumably shared among all applications running on the system. A good way to conceptualize it is as a thread pool, with an unspecified number of threads, shared across the applications currently running.

The work to be done is *described* by the sender **`snd`**. As mentioned in the previous article [Teodorescu24], senders merely describe work – they do not represent the actual execution of that work. To execute the work, the sender must be started. Senders are somewhat similar to **`std::function`** objects: they represent function-like work, but defining such an object does not immediately execute it; the function object must be invoked to start the work. In our case, the operation that starts the work is **`sync_wait`**. This function initiates the work described by the sender and blocks until the result is produced. It then returns the result of the work, although in our example, we ignore the result.

As shown in the example, the **`stdexec`** library provides two namespaces: **`stdexec`** and **`exec`**. Similarly, the include files are organized into folders

```
#include <exec/system_context.hpp>
#include <stdexec/execution.hpp>

int main() {
  stdexec::scheduler auto sched =
    exec::get_system_scheduler();
  stdexec::sender auto snd =
    stdexec::schedule(sched)
    | stdexec::then([]
    { printf("Hello, concurrency!\n"); });
  stdexec::sync_wait(std::move(snd));
}
```

Listing 1

named **`stdexec`** and **`exec`**. Everything under the **`stdexec`** namespace is part of the P2300 proposal [P2300R10], which has already been accepted into the C++26 draft. Entities within the **`exec`** namespace are not part of the original P2300 proposal but are either candidates for standardization or provide useful abstractions. In our case, **`system_context`** and **`get_system_scheduler`** are proposed for standardization [P2079R5].

Work graph

In a serial program, all instructions are executed sequentially, and the order of execution is typically straightforward. For these programs, especially when following structured programming principles, understanding the scopes of different objects and code structures is crucial.

In contrast, for concurrent programs, both the ordering of instructions and the scopes of entities become important. In concurrent execution, there is a *partial ordering* of work items, forming a graph that represents the dependencies and execution flow of these items.

When examining this graph of work items, well-structured concurrency often results in the scope of an operation aligning with the span during which the operation can be executed – specifically, from the completion of all predecessors to the initiation of any successors.

Thinking of work as a graph is a quick and effective way to understand the constraints of a problem. For this reason, we will briefly discuss this graph of execution in the context of our examples.

Computing the Mandelbrot set

The Mandelbrot set is a two-dimensional fractal of great complexity, generated by the convergence of the simple formula: $f_c(z) = z^2 + c$. Figure 1 (next page) illustrates the image of a Mandelbrot fractal, centered at $c = -1.4011$ (with no imaginary component), using a scale of 512 and an iteration limit (depth) of 1000. Each iteration count is represented by a different color.

The code to compute this fractal without using concurrency is similar to the code shown in Listing 2.

We use a matrix of dimensions **`max_x`** by **`max_y`**, where each element represents a depth value that will be mapped to a color to create a colorful image. The transform functor passed to **`serial_mandelbrot`** converts

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

creating more threads than the number of hardware threads available on the system can lead to CPU oversubscription, which will degrade the application's performance

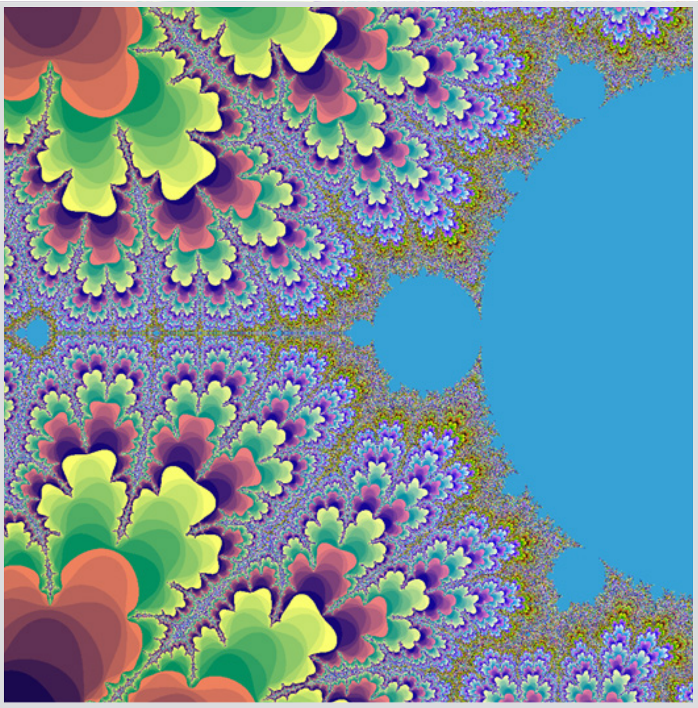


Figure 1

a position in the matrix (a pixel) into a complex value. One possible implementation for this is the `pixel_to_complex` function. The core of the algorithm resides in the `mandelbrot_core` function, which computes the depth (up to a specified limit) for a given initial complex number *c*. This function is called for each element in the matrix, iterating up to `depth` times for each.

The overall complexity of the algorithm is *O*(*max_y* * *max_x* * *depth*). It is worth noting that, for some pixels, the `mandelbrot_core` function will terminate after only a few iterations, resulting in unbalanced computation across matrix elements. Despite this, on common hardware, filling a screen with the Mandelbrot fractal at a depth of 1000 is not particularly fast. Adding concurrency to the computation could provide significant performance benefits.

Listing 3 demonstrates the changes required to modify the main function to execute the program on multiple threads. The primary change involves transforming the outer loop (which iterates over the *y* axis) into a `bulk()` call. The `bulk()` sender executes the given body `max_y` times on the current execution context. This execution context is provided by the scheduler, which, as before, is obtained using `get_system_scheduler()`. Consequently, different lines in the matrix may be computed by different threads.

If the machine running this program has 8 cores, it is reasonable to assume that the system's execution context will provide 8 OS threads to perform the work. However, creating more threads than the number of hardware threads available on the system can lead to CPU oversubscription [Wikipedia], which will degrade the application's performance.

```
int mandelbrot_core(std::complex<double> c,
    int depth) {
  int count = 0;
  std::complex<double> z = 0;
  for (int i = 0; i < depth; i++) {
    if (abs(z) >= 2.0)
      break;
    z = z * z + c;
    count++;
  }
  return count;
}

std::complex<double> pixel_to_complex(int x,
    int y) {
  double x0 = offset_x +
    (x - max_x / 2) * 4.0 / max_x / scale;
  double y0 = offset_y +
    (y - max_y / 2) * 4.0 / max_y / scale;
  return std::complex<double>(x0, y0);
}

template <typename F>
void serial_mandelbrot(int* vals, int max_x,
    int max_y, int depth, F&& transform) {
  for (int y = 0; y < max_y; y++) {
    for (int x = 0; x < max_x; x++) {
      vals[y * max_x + x] =
        mandelbrot_core(transform(x, y), depth);
    }
  }
}
```

Listing 2

The work itself is described by a sender, `snd`. To execute the work, the program invokes `sync_wait()`, which blocks until all the work is completed.

There is an important caveat in this example that is worth highlighting. By simply reading the code in Listing 3, one might assume that the definition of the `bulk()` algorithm inherently specifies the conditions under which

```
template <typename F>
void mandelbrot_concurrent(int* vals, int max_x,
    int max_y, int depth, F&& transform) {
  auto sched = exec::get_system_scheduler();
  auto snd = stdexec::schedule(sched)
    | stdexec::bulk(max_y, [=](int y) {
        for (int x = 0; x < max_x; x++) {
          vals[y * max_x + x] =
            mandelbrot_core(transform(x, y),
                            depth);
        }
      });
  stdexec::sync_wait(std::move(snd));
}
```

Listing 3

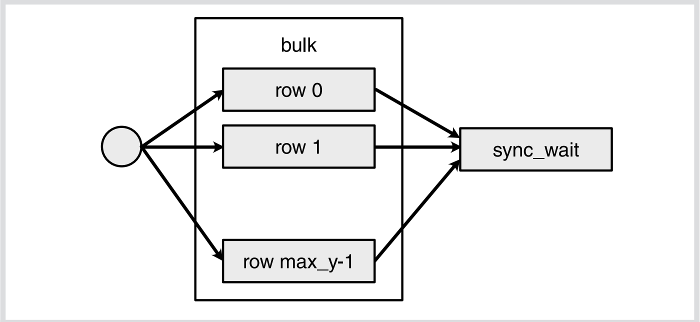


Figure 2

computations can be executed concurrently. However, this is not entirely accurate. By default, the **bulk()** algorithm functions as a glorified for loop without any built-in concurrency.

Concurrency is introduced through specialization. Algorithms like **bulk()** can be specialized based on the scheduler they execute on. In this case, the system scheduler provides a specialization for **bulk()** that leverages the execution context it manages. It is the combination of the system scheduler and the **bulk()** algorithm that enables the desired multi-threaded implementation. If the system scheduler were removed from the code, the computation would run sequentially.

The graph for this problem, shown in Figure 2, illustrates the dependencies between tasks. From a concurrency perspective, this problem is relatively straightforward, as the graph is not complex.

In conclusion, transforming single-threaded code into multi-threaded code using the senders/receivers framework does not need to be difficult.

Concurrent sort

In the previous example, achieving multi-threaded execution involved transforming a **for** loop into a **bulk()** call. Given a known number of iterations, **bulk()** effectively executes the work concurrently, adhering to the rules defined by the current scheduler. But what happens when the work to be done is not linear, and the number of iterations is unknown upfront? This section provides an example to address this scenario.

Here, we focus on adapting a classic implementation of quick sort to run concurrently. The serial version of the algorithm is shown in Listing 4[1]. For small collections, we use **std::sort** as the base case for recursion. For larger collections, the algorithm partitions the elements into three groups based on a *pivot*: elements smaller than the pivot, elements equal to the pivot, and elements larger than the pivot. The pivot is chosen to maximize the likelihood of balanced partitions. Once the data is partitioned, we recursively sort the smaller and larger partitions.

```
template <std::random_access_iterator It>
void serial_sort(It first, It last) {
  auto size = std::distance(first, last);
  if (size_t(size) < size_threshold) {
    // Use serial sort under a certain threshold.
    std::sort(first, last);
  } else {
    // Partition the data, such as elements
    // [0, mid1) < [mid1, mid2) <= [mid2, n).
    // Elements in [mid1, mid2) are equal to
    // the pivot.
    auto p = sort_partition(first, last);
    auto mid1 = p.first;
    auto mid2 = p.second;

    serial_sort(first, mid1);
    serial_sort(mid2, last);
  }
}
```

Listing 4

```
template <std::random_access_iterator It>
void concurrent_sort_impl(It first, It last,
    exec::async_scope& scope) {
  auto size = std::distance(first, last);
  if (size_t(size) < size_threshold) {
    // Use serial sort under a certain threshold.
    std::sort(first, last);
  } else {
    // Partition the data, such as elements
    // [0, mid1) < [mid1, mid2) <= [mid2, n).
    // Elements in [mid1, mid2) are equal to the
    // pivot.
    auto p = sort_partition(first, last);
    auto mid1 = p.first;
    auto mid2 = p.second;

    // Spawn work to sort the right-hand side.
    stdexec::sender auto snd
      = stdexec::schedule
          (exec::get_system_scheduler())
        | stdexec::upon_error([]
              (std::error_code ec) -> void {
            throw std::runtime_error
              ("cannot start work");
          })
        | stdexec::then([=, &scope] {
            concurrent_sort_impl(mid2, last,
            scope);
          })
        ;
    scope.spawn(std::move(snd));
    // Execute the sorting on the left side,
    // on the current thread.
    concurrent_sort_impl(first, mid1, scope);
  }
}
template <std::random_access_iterator It>
void concurrent_sort(It first, It last) {
  exec::async_scope scope;
  concurrent_sort_impl(first, last, scope);
  stdexec::sync_wait(scope.on_empty());
}
```

Listing 5

Listing 5 illustrates how this algorithm can be implemented using senders/receivers to achieve concurrent execution. This example utilizes an **async_scope**[2] object to manage dynamic concurrent work, necessitating the wrapping of the recursive function. The **async_scope** provides a dynamic scope for the concurrent tasks it spawns. The core logic of the sorting function remains largely unchanged; the primary modification is that the sorting of the right-side subrange is now offloaded to the system scheduler, allowing it to run concurrently with the sorting of the left-side subrange.

The code used to spawn work appears more complex because it includes handling errors of type **std::error_code**. The system scheduler is currently undergoing standardization, and the **stdexec** implementation is continuously evolving to align with this process. At the time of writing, scheduling work on the system context may produce an error of type **std::error_code**. However, **async_scope** does not natively handle such errors – it only manages exceptions. To bridge this gap, we need to convert the **std::error_code** into an exception, which we accomplish using the **upon_error()** algorithm.

Ideally, the result of the lambda passed to **upon_error()** is sent through the value channel (see the previous article in this series [Teodorescu24]). The value channel for the **schedule()** algorithm is **set_value(void)**. Since we do not want to introduce an additional value channel, the lambda passed to **upon_error()** must return void. Even if the lambda body is empty, it is not declared as **noexcept**. Consequently, **upon_error()** assumes that the lambda might throw, ensuring the inclusion of a **set_error(std::exception_ptr)** error channel in its response. This mechanism enables the conversion

1 This may not be the most optimal version of sorting; the serial method presented here is a simplification of the concurrent version.

2 The name proposed for standardization is **counting_scope**; however, we use **async_scope** here as this is the name currently used by the **stdexec** library. See [P3149R6].

of the `set_error(std::error_code)` channel into a `set_error(std::exception_ptr)` channel. Later in this article, we will demonstrate another method for modifying the error channels of a sender.

Even if the `std::error_code` error channel is not ultimately standardized (and `stdexec` removes support for it), this exercise provides valuable insights into handling error channels effectively.

Now, let's dive into the most interesting aspect of this example: the concept of work span. In the previous examples, the span of spawned work was always contained within the span of the enclosing function, meaning the work spans were fully nested. This approach is known as *structured concurrency*. However, in the example from Listing 5, the span of the spawned work can extend beyond the end of the enclosing function. In this case, the scopes do not fully nest; we call this *weakly-structured concurrency*.

One of the key purposes of `async_scope` is to impose a weak structure on work that might otherwise lack structure. The structure imposed here ensures that all work must be completed before the call to `stdexec::sync_wait(scope.on_empty())`. This statement blocks the current thread until all work within the scope is finished (i.e., the scope is empty).

You can think of `async_scope` as a sophisticated shared counter. Each time work is spawned on the scope, the counter increments. When the work is completed, the counter decrements. The `on_empty()` method returns a sender that completes when the counter reaches zero, signifying that there is no outstanding work.

Whenever we introduce weakly-structured constructs, we must carefully double-check the safety of the approach. Specifically, we need to ensure that the spawned work does not access anything from the stack of the function that might be deallocated before the work is completed. In this case, the spawned work only accesses a section of the input sequence, and no other work item accesses the same section simultaneously.

The concurrent sort algorithm performs partitioning in a non-parallelizable manner. However, it then continues dividing the work in half, adding tasks to process the partitions concurrently. This causes the number of worker threads to gradually increase until all threads in the system scheduler are fully utilized for sorting tasks.

The concurrent structure of the problem is illustrated in Figure 3. It highlights the recursive nature of the problem and the way tasks are divided and executed concurrently.

In this example, we demonstrated how to use weakly-structured concurrency and discussed some of the challenges associated with managing error channels.

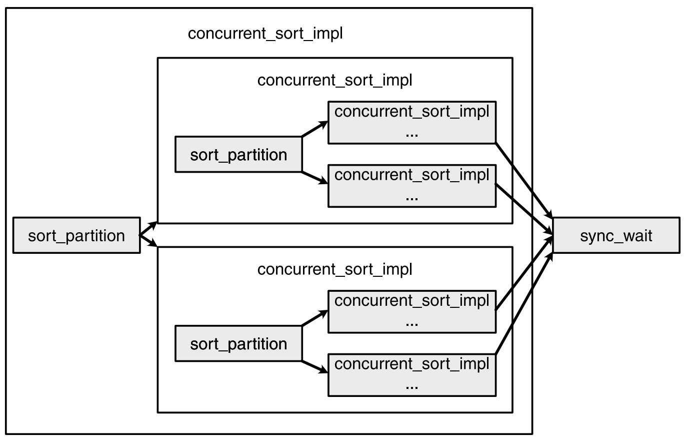


Figure 3

Processing images

Let's now tackle a more complex problem, one that introduces additional challenges and interesting discussions. We will build an application that reads all JPEG images from a folder, applies a filter to each image, and saves the processed images to a different folder. Since processing an image can be time-consuming and there may be multiple images to handle, the application could benefit significantly from leveraging multiple threads.

An outline of the program, including function declarations and the `main()` function body, is shown in Listing 6. The program uses OpenCV [OpenCV] for image processing. All functions returning `cv::Mat` are standard functions that process images and return new ones. The `read_file` and `write_file` functions perform file reading and writing, as expected. Our focus will be on three key functions: `tr_cartoonify`, `error_to_exception`, and `process_files`.

Figure 4 (next page) illustrates the execution graph for this problem, assuming there are three files to process. The graph resembles a pipeline, where the first and last stages (`read_file` and `write_file`) are I/O operations, and the intermediate stages consist of operations that can benefit from concurrent execution across multiple threads.

Adding concurrency to a small pipeline

The 'cartoonify' operation involves applying a mask to an image with reduced colors, where the mask consists of the edges of the original picture. To produce the final result, we need two intermediate images: one with reduced colors and one showing the edges. The reduced-color image is obtained by calling `tr_reduce_colors`, while the edges image is computed through a sequence of operations: `tr_blur`, `tr_to_grayscale`, and `tr_adaptthresh`. Since these operations can be computationally expensive and the two processing streams are independent, it makes sense to execute them concurrently. The code for this is shown in Listing 7 (next page).

To enable concurrency, we again rely on the system scheduler. The two concurrent chains of computation are represented by the two parameters passed to `when_all()`. Each computation begins with a call to `transfer_just()`, which transfers execution to a thread managed by the system scheduler while passing the source image as an argument. As

```
cv::Mat tr_apply_mask(const cv::Mat& img_main,
  const cv::Mat& img_mask);
cv::Mat tr_blur(const cv::Mat& src, int size);
cv::Mat tr_to_grayscale(const cv::Mat& src);
cv::Mat tr_adaptthresh(const cv::Mat& img,
  int block_size, int diff);
cv::Mat tr_reducecolors(const cv::Mat& img,
  int num_colors)
cv::Mat tr_oilpainting(const cv::Mat& img,
  int size, int dyn_ratio);
auto tr_cartoonify(const cv::Mat& src,
  int blur_size, int num_colors, int block_size,
  int diff);

auto error_to_exception();

std::vector<std::byte>
  read_file(const fs::directory_entry& file);
void write_file(const char* filename,
  const std::vector<unsigned char>& data);

exec::task<int>
  process_files(const char* in_folder_name,
    const char* out_folder_name, int blur_size,
    int num_colors, int block_size, int diff);

int main() {
  auto everything = process_files("data", "out",
    blur_size, num_colors, block_size, diff);
  auto [processed] = stdexec::sync_wait
     (std::move(everything)).value();
  printf("Processed images: %d\n", processed);
  return 0;
}
```

Listing 6

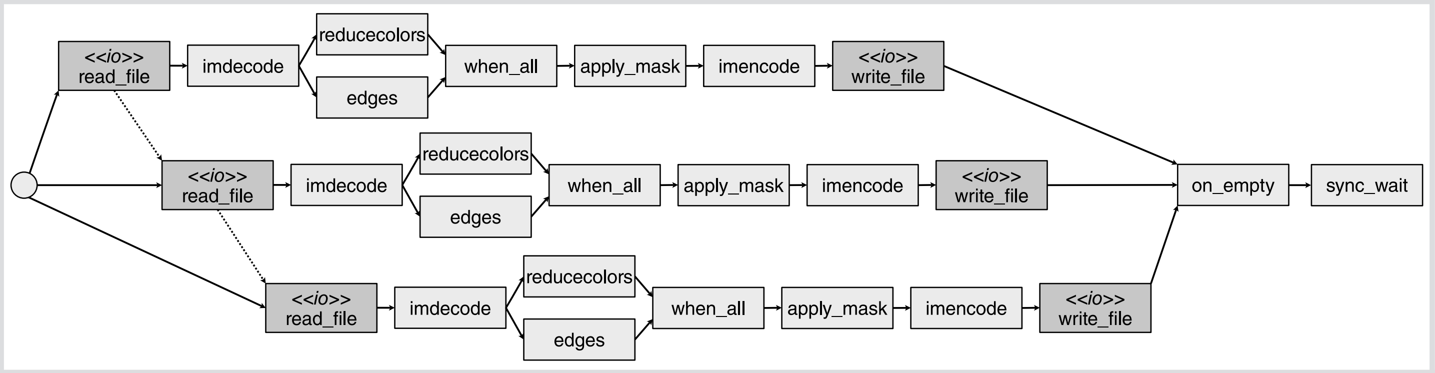


Figure 4

before, the issue of the `std::error_code` error channel arises, and this time we address it by chaining the `error_to_exception()` sender adaptor. The primary work for each computation chain is encapsulated in lambdas passed to the `then()` algorithm, clearly showing the steps needed to produce the two intermediate images.

The `when_all()` algorithm combines the two computations, creating a sender that completes only when both branches have finished. Upon completion, it triggers a value completion, passing the two resulting images. On top of `when_all()`, we use the `then()` algorithm again to combine the two images into a single output image. The result is a sender that completes with the final image as a value. Additionally, it can signal completion with an exception-encoded error or a stopped signal.

The `tr_cartoonify()` function simply returns this resulting sender. The sender's type is complex and not easily nameable, as it encapsulates type information from all the senders and lambdas involved in the function.

Although this image processing function introduces limited concurrency (less than a 2× improvement), it still provides a notable performance boost compared to the serial version.

Consolidating error completion signals

Let's now focus on the `error_to_exception()` function, shown in Listing 8. This function achieves essentially the same goal as the `upon_error()` approach from the previous section, but in a slightly more general manner. The limitations of `upon_error()` make it less practical for some scenarios. Specifically, `upon_error()` cannot handle multiple error completion signals from the previous sender, and it must return the correct value type to integrate seamlessly into the pipeline.

Our approach in this case converts any error type into an exception. Each time an error is sent by the previous sender, the lambda passed to `let_error()` is invoked. If the previous sender supports both the `set_error(std::exception_ptr)` and the `set_error(std::error_code)` completion signatures, the lambda must handle both an `std::exception_ptr` and an `std::error_code` as arguments. To accommodate this, we use a generic auto parameter for the lambda.

In the body of the lambda, we differentiate between two cases: if the argument is an exception pointer, we simply forward it; otherwise, we create a new exception and forward that.

In both cases, the lambda returns a sender that produces an error. It is crucial that the return types of the two cases are the same; otherwise, the code would result in a compilation error.

While this process may seem cumbersome to users unfamiliar with such completion signal manipulations, it is likely that users will adapt quickly to these patterns with practice.

The main transformation

Listing 9 (next page) shows the main body of the `process_files()` function, which represents the core process of the program. Setting aside the fact that this is a coroutine, as well as the initialization of the two schedulers and the `async_scope` object at the start of the function, the body itself is relatively straightforward. It iterates over all the JPEG images in the source folder and processes each one. The processing is divided into two parts: reading the file's content and processing the image.

The file-reading step simply involves a call to the `read_file()` function, executed within the context of the `io_sched` scheduler object. The reason for using this scheduler will be explained in the next section. This step also involves a `co_await` operation, which will be discussed later.

The main transformation is shown in Listing 10 (also next page). Here, the content of the input file is transferred to the `cpu_sched` scheduler (which is the system scheduler), where most of the processing takes place. As in previous examples, we consolidate the error channel by including `error_to_exception()` in the pipeline. Once this is done, the image is decoded on a CPU thread using `cv::imdecode()`.

```
auto tr_cartoonify(const cv::Mat& src,
    int blur_size, int num_colors,
    int block_size, int diff) {
  auto sched = exec::get_system_scheduler();
  stdexec::sender auto snd =
    stdexec::when_all(
      stdexec::transfer_just(sched, src)
        | error_to_exception()
        | stdexec::then([=](const cv::Mat& src) {
            auto blurred = tr_blur(src,
              blur_size);
            auto gray = tr_to_grayscale(blurred);
            return tr_adaptthresh(gray,
              block_size, diff);
          }),
      stdexec::transfer_just(sched, src)
        | error_to_exception()
        | stdexec::then([=](const cv::Mat& src) {
            return tr_reducecolors(src,
              num_colors);
          })
    )
    | stdexec::then([](const cv::Mat& edges,
          const cv::Mat& reduced_colors) {
        return tr_apply_mask(reduced_colors,
          edges);
      });
  return snd;
}
```

Listing 7

```
auto error_to_exception() {
  return stdexec::let_error([](auto e) {
    if constexpr (std::same_as<decltype((e)),
                  std::exception_ptr>)
      return stdexec::just_error(e);
    else
      return stdexec::just_error
        (std::make_exception_ptr
        (std::runtime_error("other error")));
  });
}
```

Listing 8

```
exec::task<int> process_files(const char*
    in_folder_name, const char* out_folder_name,
    int blur_size, int num_colors,
    int block_size, int diff) {
  exec::async_scope scope;
  exec::static_thread_pool io_pool(1);
  auto io_sched = io_pool.get_scheduler();
  auto cpu_sched = exec::get_system_scheduler();

  int processed = 0;
  for (const auto& entry
      : fs::directory_iterator(in_folder_name)) {
    auto extension = entry.path().extension();
    if (!entry.is_regular_file() || (extension
        != ".jpg") && (extension != ".jpeg"))
      continue;
    auto in_filename = entry.path().string();
    auto out_filename =
      (fs::path(out_folder_name) /
       entry.path().filename()).string();
    printf("Processing %s\n",
      in_filename.c_str());
    auto file_content =
       co_await (stdexec::schedule(io_sched)
       | stdexec::then([=]
       { return read_file(entry); }));
    stdexec::sender auto work = ...
    scope.spawn(std::move(work));
  }
  co_await scope.on_empty();
  co_return processed;
}
```

Listing 9

Once we retrieve the image, we apply the **`tr_cartoonify()`** transformation. However, instead of using the typical **`then()`** algorithm, we use **`let_value()`**. The **`then()`** algorithm is appropriate when the given functor returns a value, whereas **`let_value()`** is used when the functor returns a sender. Since **`tr_cartoonify()`** returns a sender, **`let_value()`** is required. The **`let_value()`** algorithm is highly versatile and serves as the monadic bind operation for senders.

After completing the transformation, we encode the image back into a stream of JPEG bytes using the **`cv::imencode()`** function. This operation is performed on a CPU thread, as it is typically CPU-intensive. Next, we write the resulting byte stream to disk. Since this is an I/O operation, we transition to the scheduler dedicated to I/O tasks. Once the file writing is complete, we print a message to standard output (still on the I/O thread) and increment the counter for successfully processed images.

Undersubscription and oversubscription

On some modern computers, I/O operations may be fast and predominantly consume CPU resources. However, let's assume that this is not the case. Specifically, let's assume that both reading and writing image files are slow operations that do not heavily utilize CPU cycles. For the sake of discussion, we will assume that I/O accounts for 25% of the program's total runtime[3].

If we were to add concurrency to the program without considering this, the CPU cores would spend significant time processing images only to go idle for approximately 25% of the time, waiting on I/O operations. This inefficiency could worsen if I/O operations on one thread interfere with I/O on another thread, leading to greater performance degradation as the level of concurrency increases.

A common solution to this problem is to create a pipeline where all I/O operations are handled on a single thread, while CPU-intensive operations are distributed across a thread pool sized to match the number of physical cores on the machine. To implement this, we use a scheduler obtained from a **`static_thread_pool`** (note that this is not proposed for standardization) dedicated to I/O tasks. This scheduler is distinct from the system scheduler, which is designed to match the available hardware resources.

3 These assumptions are made to illustrate the thread-switching technique described. In practice, this approach may not always be worthwhile. Readers should measure performance before making similar assumptions.

```
stdexec::sender auto work =
  stdexec::transfer_just(cpu_sched,
    cv::_InputArray::rawIn(file_content))
  | error_to_exception()
  | stdexec::then([=](cv::InputArray
      file_content) -> cv::Mat {
        return cv::imdecode(file_content,
        cv::IMREAD_COLOR);
    })
  | stdexec::let_value([=](const cv::Mat& img) {
      return tr_cartoonify(img,
        blur_size, num_colors, block_size, diff);
    })
  | stdexec::then([=](const cv::Mat& img) {
      std::vector<unsigned char>
        out_image_content;
      if (!cv::imencode(extension, img,
          out_image_content)) {
        throw std::runtime_error
          ("cannot encode image");
      }
      return out_image_content;
    })
  | stdexec::continues_on(io_sched)
  | stdexec::then([=]
      (const std::vector<unsigned char>& bytes) {
      write_file(out_filename.c_str(), bytes);
    })
  | stdexec::then([=] { printf("Written %s\n",
      out_filename.c_str()); })
  | stdexec::then([&] { processed++; });
```

Listing 10

If the target hardware has *N* physical cores, one might wonder why not use a thread pool with *N* + 1 threads. The reason lies in the risk of *oversubscription*: running more CPU-intensive tasks simultaneously on a system with less physical cores can lead to decreased performance due to excessive task switching.

A common misconception is that running two tasks, each requiring one second to complete, simultaneously on one core will somehow finish in one second. In reality, running them concurrently on the same core often takes longer than two seconds due to the overhead of context switching. Running such tasks sequentially is typically more efficient. I explored this concept in my ACCU 2023 talk [Teodorescu23]. To illustrate, imagine trying to read two books at the same time or a physician performing complex surgery while attending a hospital board meeting over the phone. Running two tasks on the same physical core involves frequent context switches, which are inherently expensive.

For optimal performance, the goal is to achieve near 100% CPU utilization across all cores for the entire program duration. If CPU utilization falls below 100%, we encounter undersubscription, where some cores remain idle despite work being available. Conversely, if workload exceeds 100% CPU utilization, excessive task switching occurs, and the processor spends valuable time managing context switches instead of executing critical tasks.

To address this, it is common practice to offload all I/O operations from CPU-intensive work and execute them on a dedicated execution engine.

Coroutines and senders

This example highlights another intriguing aspect of the senders/receivers framework: its interaction with coroutines. With minimal annotations to a coroutine type, coroutines can effectively behave as senders. This allows us to **`co_await`** a sender or use a coroutine object in place of a sender.

The **`stdexec`** library provides such a coroutine type, **`exec::task`**, which we use in our example for the **`process_files()`** coroutine. Within the coroutine, we **`co_await`** the result of reading the input file on the I/O execution context and also **`co_await`** the completion of all activities using **`scope.on_empty()`**. On the other end, in the **`main()`**

function, we pass the coroutine object to the **`sync_wait()`** algorithm, demonstrating that coroutines can seamlessly integrate where senders are used.

In this case, **`process_files()`** begins execution on the main thread. After the first **`co_await`**, execution continues on the I/O thread. At the end of the coroutine, execution remains on the I/O thread. The final **`sync_wait()`** then switches the main execution path back to the main thread.

While writing this, I realized there is a bug in the code. I decided to leave the bug as is and explain it, as this may be more helpful for the reader. The issue is that we are destroying the **`io_pool`** object when exiting the scope of the coroutine, but execution may still be ongoing on one of its threads. Ideally, we should switch back to the main thread before destroying this pool. Alternatively, we could transfer control to one of the CPU threads, as the system scheduler guarantees the validity of its threads throughout the application's lifetime, including before and after **`main()`**.

Returning to the topic of coroutines, there is nothing that coroutines can achieve that cannot also be done with senders, and the reverse is true as well. However, using senders is generally more efficient. Despite this, I find coroutines useful in two specific scenarios:

- **Non-linear control flow:** When logic involves loops or branches, expressing these flows using senders can be challenging due to the lack of standardized algorithms for such patterns. Even if such algorithms were standardized, expressing everything through expression composition would likely be more cumbersome than using traditional control structures.
- **Type erasure:** Currently, there is no type-erased sender proposed for standardization. This means that every sender's internal structure must be fully visible at the point of use. In contrast, coroutines naturally hide implementation details, making them a good choice for situations requiring type erasure.

At the time of writing, the **`task`** type used in this example has not been proposed for standardization. However, there is broad consensus that it is worth standardizing.

Takeaways

Following the article in the last *Overload* [Teodorescu24], which introduced the senders/receivers framework accepted into the working draft of the C++26 standard, this article explores several examples. The goal is to familiarize readers with writing programs using senders/receivers. Each of the three examples presented here aims to improve performance by employing multi-threading.

The examples demonstrate that adding multi-threading to applications does not have to be a daunting task. By thinking in terms of execution graphs, concurrent solutions can be expressed clearly and intuitively, avoiding the need for manual synchronization primitives, which are notoriously error-prone[4].

While there are some challenges users may encounter when working with senders/receivers, they are relatively minor compared to the complexities of multi-threading with raw threads and locks. One important consideration is managing the lifetime of objects in relation to the threads accessing them. This article highlights a bug encountered by the author during implementation to emphasize this point. In contrast, manual multi-threading is typically far more difficult, as it requires reasoning about a larger number of objects, with much of the reasoning being non-local.

Another challenge users might face is handling the completion signals of senders. Certain transformations may create unexpected completion signals, forcing the user to address them. Improperly connected senders can result in long, cryptic compilation errors. In our case, we had to consolidate two types of error completions into a single type to resolve these issues.

The examples presented here highlight several key strengths of the senders/receivers framework:

- **Structuredness.** Senders/receivers impose a clear structure on an application's concurrency. In well-structured code, concurrency is nested in such a way that concurrency concerns can be abstracted away by the enclosing construct (e.g., a function or coroutine). The framework also supports weakly-structured concurrency, where scopes do not fully nest but can be organized using dynamic scopes to encompass all computations. Both approaches are far superior to the unstructured methods of managing concurrency with raw threads and locks.
- **Local reasoning.** Most concurrency reasoning can be confined to a local scope. For fully structured code, all reasoning remains local. In weakly-structured code, while concurrency concerns may extend eyond the current function, they are still constrained to a defined dynamic scope.
- **Safety.** The reader might have noticed that the discussion about safety was minimal. This is because, when object lifetimes are properly managed, the framework inherently avoids safety issues. It eliminates concerns like data races and deadlocks, which are common in unstructured multithreading.
- **Performance.** The senders/receivers framework can achieve zero abstraction cost. There are no unnecessary memory allocations, and no extra synchronization overhead is introduced. This makes it possible to build highly performant multi-threaded applications.

Together, these strengths make senders/receivers an excellent framework for writing multi-threaded code. While the syntax might feel less intuitive and diagnostics may sometimes be trickier, the framework offers a powerful and reliable way to build robust and efficient multi-threaded software.

The real question is how well this framework works for you, the reader. Is it as straightforward as the article suggests, or do you encounter challenges when applying it to your problems? I would love to hear your feedback and learn about your experiences using this approach. ■

References

[ExamplesCode] Lucian Radu Teodorescu, overload185_sr_examples, https://github.com/lucteo/overload185_sr_examples.

[OpenCV] OpenCV, OpenCV – Open Computer Vision Library, https://opencv.org/.

[P2079R5] Lucian Radu Teodorescu, Ruslan Arutyunyan, Lee Howes, Michael Voss, P2079R5: System execution context, 2024, https://wg21.link/P2079R5.

[P2300R10] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, P2300R10: std::execution, 2024, https://wg21.link/P2300R10.

[P3149R6] Ian Petersen, Jessica Wong, Ján Ondrušek, Kirk Shoop, Lee Howes, Lucian Radu Teodorescu, async_scope – Creating scopes for non-sequential concurrency, https://wg21.link/P3149R6.

[stdexec] NVIDIA, Senders – A Standard Model for Asynchronous Execution in C++, https://github.com/NVIDIA/stdexec.

[Teodorescu24] Lucian Radu Teodorescu, Senders/receivers: An Introduction, *Overload* 184, December 2022

[Teodorescu23] Lucian Radu Teodorescu, 'Concurrency Approaches: Past, Present, and Future', *ACCU Conference*, 2023, https://www.youtube.com/watch?v=uSG240pJGPM.

[WG21Exec] WG21, 'Execution control library' in Working Draft Programming Languages – C++ https://eel.is/c++draft/#exec.

[Wikipedia] Wikipedia, Resource contention, https://en.wikipedia.org/wiki/Resource_contention.

4 The lack of need for manual synchronization is discussed in [Teodorescu24]. The main idea is that we prefer structuring concurrency and explicitly encoding the dependencies between work items.

Bit Fields, Byte Order and Serialization

Network packets can be represented as bit fields. Wu Yongwei explores some issues to be aware of and offers solutions.

In order to store data most efficiently, the C language has supported bit fields since its early days. While saving a few bytes of memory isn't as critical today, bit fields remain widely used in scenarios like network packets. Endianness adds complexity to bit field handling – especially since network packets are typically big-endian, while most modern architectures are little-endian. This article explores these problems and their solutions, including my reflection-based serialization project.

Memory layout of bit fields

The memory layout of bit fields is implementation-defined. In a typical little-endian environment, bit fields start from the lower bits of the lower byte and extend toward higher bits and bytes. In a typical big-endian environment, bit fields start from the higher bits of the lower byte and extend toward lower bits and higher bytes.

Let's consider a practical scenario. Suppose we want to use a 32-bit integer to store a date. How should we achieve this? A simple approach is to store the number of days from a fixed point of time (e.g. 1 January 1900). We can calculate the number of years that can be expressed as follows:

$$\text{years} = \frac{2^{32}}{365.2425} \approx 11,759,221 \quad (1)$$

However, with this approach, extracting specific year, month, and day information becomes very cumbersome. A simpler way is to store the year, month, and day as bit fields. We can define the following struct, using only 32 bits:

```
struct Date {
  int      year  : 23;
  unsigned month : 4;
  unsigned day   : 5;
};
```

Our intention is to use a 23-bit signed integer for the year (ranging from -4,194,304 to 4,194,303), a 4-bit unsigned integer for the month (0–15, covering legal values 1–12), and a 5-bit unsigned integer for the day (0–31, covering legal values 1–31). This representation is similarly compact, with a slightly narrower range, but it's quite sufficient and much more convenient for many common usages (excepting interval calculation).

If you want to store data in a file or send it over a network, directly sending in-memory data is potentially problematic. Big-endian and little-endian environments have different memory layouts for such bit fields.

Little-endian environments store them as follows (the memory layout on mainstream processors):

bit:	7	6	5	4	3	2	1	0
byte 0	y7	y6	y5	y4	y3	y2	y1	y0
byte 1	y15	y14	y13	y12	y11	y10	y9	y8
byte 2	m0	y22	y21	y20	y19	y18	y17	y16
byte 3	d4	d3	d2	d1	d0	m3	m2	m1

Big-endian environments store them differently (the memory layout expected by network packets):

bit:	7	6	5	4	3	2	1	0
byte 0	y22	y21	y20	y19	y18	y17	y16	y15
byte 1	y14	y13	y12	y11	y10	y9	y8	y7
byte 2	y6	y5	y4	y3	y2	y1	y0	m3
byte 3	m2	m1	m0	d4	d3	d2	d1	d0

As we can see, these two approaches differ significantly. If we want to serialize in the big-endian order (which is the standard in the networking world), we have two possible solutions:

1. When bit fields don't cross byte boundaries, we can design separate structs for big-endian and little-endian systems, using conditional compilation to select the appropriate definition.
2. When bit fields cross byte boundaries, the above approach alone isn't sufficient. We need to define different structs (with reversed bit-field ordering for big-endian versus little-endian) and perform byte-order conversion during serialization and deserialization (using functions like **`htonl`**).

Examples of the simple approach

Here are some actual definitions from Linux.

A simple case (single byte, only requiring order reversal):

```
struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;
    unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
#else
# error     "Please fix <bits/endian.h>"
#endif
    // …
  };
```

Listing 1 (next page) is a more complex case (multiple bytes, but not crossing byte boundaries).

As we can see, the field ordering here is quite distinctive. This arrangement ensures these fields have a consistent memory layout on both little-endian and big-endian systems.

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

for targets with known lengths, we should be able to avoid heap memory allocation entirely

```
struct tcphdr
  {
    __extension__ union
    {
      // …
      struct
      {
        uint16_t source;
        uint16_t dest;
        uint32_t seq;
        uint32_t ack_seq;
# if __BYTE_ORDER == __LITTLE_ENDIAN
        uint16_t res1:4;
        uint16_t doff:4;
        uint16_t fin:1;
        uint16_t syn:1;
        uint16_t rst:1;
        uint16_t psh:1;
        uint16_t ack:1;
        uint16_t urg:1;
        uint16_t res2:2;
# elif __BYTE_ORDER == __BIG_ENDIAN
        uint16_t doff:4;
        uint16_t res1:4;
        uint16_t res2:2;
        uint16_t urg:1;
        uint16_t ack:1;
        uint16_t psh:1;
        uint16_t rst:1;
        uint16_t syn:1;
        uint16_t fin:1;
# else
#  error "Adjust your <bits/endian.h> defines"
# endif
        // …
      };
    };
};
```

Listing 1

Example of the 'standard' approach

Since its bit fields cross byte boundaries, merely modifying the field order will not do for our **`Date`** struct as shown above. The conventional approach is to use the code in Listing 2 for serialization.

```
#ifdef _WIN32
#include <winsock2.h>  // htonl/...
#else
#include <arpa/inet.h> // htonl/...
#endif

struct Date {
  union {
    struct {
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
      unsigned   day : 5;
      unsigned month : 4;
      int       year : 23;
#else
      int       year : 23;
      unsigned month : 4;
      unsigned   day : 5;
#endif
    };
    unsigned year_month_day;
  };
};

// …
Date date;
// …
date.year_month_day = htonl(date.year_month_day);
// date is ready for transmission
```

Listing 2

Under this 'standard' approach, bit fields that can be assembled into a single integer must be placed in a struct, which is then wrapped in a union. This allows us to directly access the integer for byte-order conversion later. Of course, we need to determine whether the platform is little-endian or big-endian to choose the appropriate struct definition.

Since macros for endianness detection aren't standardized, such code isn't truly cross-platform. However, the code above works correctly on mainstream compilers like GCC, Clang, and MSVC. While GCC and Clang recognize the special macros **`__BYTE_ORDER__`** and **`__ORDER_LITTLE_ENDIAN__`**, MSVC recognizes neither. Therefore, MSVC defaults to the first case (**`#if 0 == 0`**), which conveniently matches Windows' use of little-endian order.

This approach remains cumbersome, requiring manual maintenance of two code branches and attention to **`htonl`**-like function calls – exactly once for serialization or deserialization, no more and no less! Experience from real projects shows this method is error-prone, and issues like missed or duplicate byte-order conversions are common.

It is even worse than that. While this approach is common in C, and all mainstream C++ compilers continue to allow such code to work, it is technically undefined behaviour in C++. The orthodox way is to use **`bit_cast`** or **`memcpy`**, which would make the code even more complicated.

Serialization features in Mozi

At the 2023 C++ Summit (China), I presented static reflection and demonstrated the Mozi open source project [mozi] that utilized manual reflection techniques. Using macros and templates, this project provides basic reflection functionality in C++17, even though C++ does not yet support static reflection natively.

This year I found more time to implement serialization and deserialization for network messaging in Mozi. Now, we only need to define a *reflected* struct to enable fully automated serialization and deserialization – users don't need to manually write field-specific handling code or perform byte-order conversions. For example, for a **`Date`** struct (without using bit fields for now), we can serialize and deserialize it as shown in Listing 3 (next page).

```
#include <assert.h>
#include <stdint.h>
#include <mozi/bit_fields.hpp>
#include <mozi/net_pack.hpp>
#include <mozi/print.hpp>
#include <mozi/serialization.hpp>
#include <mozi/struct_reflection.hpp>

DEFINE_STRUCT(
  Date,
  (int16_t)year,
  (uint8_t)month,
  (uint8_t)day
);

DECLARE_EQUAL_COMPARISON(Date);


int main()
{
  Date date{2024, 8, 19};
  mozi::println(date);

  mozi::serialize_t result;
  mozi::net_pack::serialize(date, result);
  mozi::println(result);

  mozi::deserialize_t input{result};
  Date date2{};
  auto ec =
    mozi::net_pack::deserialize(date2, input);
  assert(ec ==
         mozi::deserialize_result::success);
  assert(input.empty());
  assert(date == date2);
}
```

Listing 3

This program will output the following result:

```
{
  year: 2024,
  month: 8,
  day: 19
}
{ 7, 232, 8, 19 }
```

Here are some important details:

- Reflected structs don't provide comparison operations by default to avoid unnecessary 'unused function' warnings. However, you can easily enable comparison operations using macros like **`DECLARE_COMPARISON`** or **`DECLARE_EQUAL_COMPARISON`**. These operations perform member-wise comparisons.
- As a reflected object, **`date`** can be output directly to **`cout`** using **`mozi::print`**/**`println`**. Due to special handling in the code, **`uint8_t`** (i.e. **`unsigned char`**) is output as an integer rather than as a character, as is the usual case when using **`<<`**.
- The code uses the serialization result as input for deserialization, where **`input`** is a **`span`** of **`byte`**s. When deserialization completes successfully, the following conditions should be met:
  - **`ec`** indicates success
  - **`input`**` is empty (indicating all input has been consumed)
  - **`date2`** equals **`date`**

I would like to mention that the implementation doesn't use non-standard functions like **`htons`**. Instead, it uses handwritten platform-independent function templates. These templates are friendly for compile-time programming, and they can be translated into optimal assembly instructions during serialization (under GCC and Clang compilers at least), or even eliminated entirely on big-endian systems. You can check the results in the following link: https://godbolt.org/z/f1Gn8Mcx1

(The deserialization logic is similar, but the compiler wasn't able to generate similarly highly-optimized code, possibly due to alignment.)

The serialization target type

For flexibility and safety, the serialization target is a **`vector`**. However, for targets with known lengths, we should be able to avoid heap memory allocation entirely. Therefore, in environments that support polymorphic allocators, the default serialization target type is set to **`std::pmr::vector<std::byte>`** (which can be overridden by setting the macro **`MOZI_SERIALIZATION_USES_PMR`** to **`0`** or **`1`**). Using allocators provided by C++17, we can avoid the heap allocations easily in such circumstances. Here's an example:

```
std::byte buffer[50];
std::pmr::monotonic_buffer_resource
  res(buffer, sizeof buffer);
std::pmr::polymorphic_allocator<std::byte>
  a(&res);
mozi::serialize_t result(a);
result.reserve(50);
// Heap memory will now be allocated only
// if the size exceeds 50 bytes
mozi::net_pack::serialize(date, result);
// Use result as you like in current scope
```

The bit_field type

Reflected structs don't directly support bit fields; but they don't have to. Instead, we can define a special class template **`bit_field`** that represents bit fields and automatically converts objects to the appropriate memory layout during serialization.

Objects of this type use the most compact integer type (**`uint8_t`**, **`uint16_t`**, or **`uint32_t`**) to store their data. The type supports construction and assignment from integers, as well as on-demand conversion to appropriate integer types. Using objects of this type feels similar to using regular integers, but like bit fields, the data is limited to a specified number of bits, with values being truncated if they exceed this limit.

Here's an example demonstrating its basic usage:

```
bit_field<4> f{13};  // Construct from integer
cout << f << '\n';   // 13 (automatically
                     // converts to unsigned)
f = 17;              // Can be assigned to
 << f << '\n';       // 1 (due to truncation)
```

The previous example showed the most common case – an unsigned **`bit_field`**. Since bit fields can also be signed (like our earlier year bit field), **`bit_field`** uses a second template parameter to specify whether it is signed (unsigned by default). Using SFINAE, I've constrained unsigned **`bit_field`**s to be convertible with **`unsigned`**, while signed **`bit_field`**s are convertible with (**`signed`**) **`int`**.

Here's some code demonstrating the subtle differences between signed and unsigned **`bit_field`**s:

```
bit_field<4> f1{13};
cout << f1 << '\n';  // 13
bit_field<4, bit_field_signed> f2{13};
 << f2 << '\n';  // -3 (due to truncation)
f1 = -1;             // Triggers warning with
                     // -Wsign-conversion
cout << f1 << '\n';  // 15
f2 = -1;             // OK
cout << f2 << '\n';  // -1
```

Bit-field containers

Just as we needed to encapsulate bit fields in a struct for byte-order conversion earlier, we need to explicitly place multiple bit fields into a bit-field container to enable proper byte-order conversion. The serialization process explicitly checks that the total number of bits is 8, 16, or 32 – otherwise, we get a compilation error.

In practice, we can simply change the **`Date`** definition to:

```
DEFINE_BIT_FIELDS_CONTAINER(
  Date,
 (bit_field<23, bit_field_signed>)year,
  (bit_field<4>)month,
  (bit_field<5>)day
);
```

```
#include <assert.h>
#include <mozi/bit_fields.hpp>
#include <mozi/net_pack.hpp>
#include <mozi/print.hpp>
#include <mozi/serialization.hpp>
#include <mozi/struct_reflection.hpp>

using mozi::bit_field;
using mozi::bit_field_signed;

DEFINE_BIT_FIELDS_CONTAINER(
  Date,
  (bit_field<23, bit_field_signed>)year,
  (bit_field<4>)month,
  (bit_field<5>)day
);

DECLARE_EQUAL_COMPARISON(Date);

int main()
{
  Date date{2024, 8, 19};
  mozi::println(date);

  mozi::serialize_t result;
  mozi::net_pack::serialize(date, result);
  mozi::println(result);

  mozi::deserialize_t input{result};
  Date date2{};
  auto ec =
    mozi::net_pack::deserialize(date2, input);
  assert(ec ==
         mozi::deserialize_result::success);
  assert(input.empty());
  assert(date == date2);
}
```

Listing 4

We do not need to change anything else in the code, and it will produce new output:

```
{
  year: 2024,
  month: 8,
  day: 19
}
{ 0, 15, 209, 19 }
```

Listing 4 is the complete working code for experimentation and reference.

If we change ‘23’ to ‘22’, we get a compilation error immediately (see Figure 1).

In other words, if padding is needed, my current approach requires explicitly writing out the padding rather than letting the compiler handle it automatically. I believe this approach better ensures serialization consistency.

It might be worth noting that, unlike the built-in bit fields (especially those on big-endian architectures), the in-memory layout of `Date` is now different from the serialization result. The serialization result is like the true (big-endian) bit fields, but in memory **`year`**, **`month`**, and **`day`** are represented as integral values, which can be accessed faster than bit fields. So we get the benefits of simplicity and performance, at the cost of a few more bytes of memory use.

Nested struct handling

The processing of reflected structs, whether for output or serialization, is recursive. For objects without variable-length data (which lacks an intuitive/direct handling method and isn’t supported in the current **`net_pack`** scheme), we can now simply nest and use them. For example:

```
DEFINE_STRUCT(
  Data,
  (std::array<char, 8>)name,
  (uint16_t)age,
  (Date)last_update
);
// …
Data data{{"John"}, 17, {2024, 8, 19}};
mozi::println(data);
mozi::serialize_t result;
mozi::net_pack::serialize(data, result);
mozi::println(result);
```

Here’s the output we get (using **`-DMOZI_PRINT_USE_FMTLIB`** flag for prettier formatting with the {fmt} library [fmt]):

```
{
  name: { 'J', 'o', 'h', 'n', '\x00', '\x00',
          '\x00', '\x00' },
  age: 17,
  last_update: {
      year: 2024,
      month: 8,
      day: 19
  }
}
{ 74, 111, 104, 110, 0, 0, 0, 0, 0, 17, 0, 15,
  209, 19 }
```

Quite convenient, isn’t it?

Extensible serialization schemes

The **`net_pack`** serialization demonstrated above is stateless and simple, suitable for basic network messaging scenarios. Mozi supports more sophisticated serialization schemes, including:

- Extending existing serialization schemes via explicit specialization to support your custom data types
- Creating new serialization schemes that can work alongside existing ones (next scheme in list is used only when previous ones don’t support a type)
- Using state data during serialization/deserialization to track counts, nesting levels, and suchlike

For more details on these advanced features, please refer to the test code in the Mozi project.

I hope you find my work and approach useful, and can apply them in your software projects. If you find any issues in the Mozi project, please don’t hesitate to report them. And patches are even more welcome! ■

References

[fmt] https://github.com/fmtlib/fmt

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

```
…
…/mozi/net_pack_bit_fields.hpp:41:5: fatal error: static_assert failed due to requirement 'size_bits ==
8 || size_bits == 16 || size_bits == 32' "A bit-fields container must have 8, 16, or 32 bits"
    static_assert(size_bits == 8 || size_bits == 16 || size_bits == 32,
    ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
…
```

Figure 1

Valgrind's Dynamic Heap Analysis Tool: DHAT

Valgrind experimental tool DHAT is now official. Paul Floyd explains what this heap analysis tool is and how to use it.

Background

Is it really over 10 years since I last wrote an article on Valgrind? It is indeed [Floyd13]. Back then I wrote about the tools that make up the standard Valgrind toolkit. Since then, one of the experimental tools has been removed (exp-sgcheck, 'experimental statics and globals check', removed mainly because of excessive false positives). Another of the tools, exp-dhat has been promoted from the experimental category to being a first-class component. DHAT is the subject of this article. One other thing that's happened in that period is that I've joined the rather informal team of Valgrind developers [Valgrind]. This means that I've progressed from believing that I know roughly how Valgrind works to being able to work on some bits and knowing that I don't understand most of it.

About DHAT

DHAT is a tool that can give you insights into heap memory use that will allow you to make changes that will make your memory use more efficient.

Since DHAT is part of Valgrind it will only work on Linux, FreeBSD, Solaris (probably) and macOS (old versions only). I don't know of any equivalent tool for Windows.

DHAT underwent a major reworking in Valgrind 3.15 (April 2019). In this change

- The 'experimental' status was removed, and the tool name changed from *exp-dhat* to just *dhat*.
- The command line options were simplified.
- The tool output changed from the console to a file.
- A web interface was added to view the results file and to allow sorting on different criteria.

If you are using Valgrind 3.14 or earlier, you should be able to follow this article, but you should expect your output to be different. You will probably want to set the **`--show-top-n`** to a value higher than the default (for instance, **`--show-top-n=500`**).

What is DHAT, exactly? It is a data profiler (the acronym stands for Dynamic Heap Analysis Tool) [DHAT]. I expect most readers are familiar with code profiling tools [Wikipedia] (like Callgrind, VTune, Quantify, Linux perf and others). As the Heap Analysis part of the name implies, DHAT performs profiling of memory accesses to blocks of heap memory.

DHAT doesn't perform profiling of the amount of heap allocation (like Massif [Massif, Floyd12], another Valgrind tool, Flame Graphs [Gregg] generated with bcc or heaptrack [Github1]). For every heap allocated block, DHAT will count every read and write within that block. For larger blocks of memory of over 1024 bytes, it will just aggregate accesses to the blocks. For smaller blocks of 1024 bytes and less, it will also generate a map of access counts within the block. I don't know of any tool that produces a whole-memory heat map, probably because that would have an excessive memory and run time overhead.

DHAT is somewhat difficult to use and works best for structures that get allocated individually on the heap. Having said that, I find it very useful, and I'm not aware of any other tools that perform the same task. There is one non-tool alternative: manual code instrumentation. The problems with manual instrumentation are:

1. You don't necessarily know in advance which structures to instrument.
2. If you want to instrument every member of your structures, that will entail a lot of code.

Using DHAT

DHAT is quite simple to use.

1. Build your executable, preferably with debug information (adding **`-g`** to the build when using GCC or LLVM toolchains).
2. Run your executable with DHAT:

   ```
   valgrind –tool=dhat {your exe name}
   ```

   At the end of the run DHAT will print a summary of the run and instructions as to how to view the results. It will also have generated a results file `dhat.out.PID` where PID will be the number of the process ID when DHAT was running. The results file isn't meant to be human readable.
3. Load the results following the instructions from step 2.

Be aware that DHAT, like all of the Valgrind tools, is very slow. I recommend that you only use it with scenarios that run in no more than a few minutes outside of Valgrind.

Example

Let's look at a small example, starting with a data structure (Listing 1, next page). I'm assuming 64bit desktop-style applications throughout the examples. The source code and an example of the results along with the DHAT viewer files can be found on GitHub [Floyd]. You can view the results on any platform with a web browser.

I have deliberately not initialized **`f2`** in the constructor. I have also deliberately initialized **`f3`** with a short string that will fit in libc++ 'short string optimization' (SSO). This means that allocating an instance of **`TestClass`** only needs one call to **`operator new`**. Normally when using DHAT you work backwards from the results to the source code and data structures. I'll do that the other way round, working forwards from the code to the results, for explanatory reasons. What is the size of **`TestClass`**? That depends a bit. The structure has 8-byte alignment. So, the total size is:

```
sizeof(int) + 4 hole + sizeof(double) +
sizeof(std:string)
```

Paul Floyd has been writing software, mostly in C++ and C, for about 30 years. He lives near Grenoble, on the edge of the French Alps and works for Siemens EDA developing tools for analogue electronic circuit simulation. In his spare time, he maintains Valgrind. He can be contacted at pjfloyd@wanadoo.fr

pahole is a great tool and I strongly recommend its use in conjunction with DHAT.

```
#include <string>
#include <list>
#include <iostream>
class TestClass
{
  int f1;
  double f2;
  std::string f3;
public:
  TestClass() : f1{}, f3{"small string"} {}
  int getF1() const { return f1; }
};
```

Listing 1

The size of `std::string` depends on the platform. With clang++/libc++ it is 24. With g++/libstdc++ it is 32. Since I'm using FreeBSD amd64 and aarch64, the size that I see is 24, and the size of `TestClass` is 40. You can check your data structure layouts using a tool called *pahole* (part of the dwarves package [Github2]). To use *pahole* you need a binary with debug information. The tool reads the DWARF debug info from the binary and prints a summary of the layouts of all data structures that it finds, including a summary of any wasted space and which blocks of members fit in a cacheline. Figure 1 is the output for `TestClass`.

```
class TestClass {
    int           f1;            /*     0     4 */

    /* XXX 4 bytes hole, try to pack */

    double        f2;            /*     8     8 */
    string        f3;            /*    16    24 */
public:
    void TestClass(class TestClass *);
    int getF1(const class TestClass  *);
    void ~TestClass(class TestClass *);
    /* size: 40, cachelines: 1, members: 3 */
    /* sum members: 36, holes: 1, sum holes: 4 */
    /* last cacheline: 40 bytes */
};
```

Figure 1

The comments at the end of the lines with data members have two numbers. The first is the cumulative size so far and the second is the size of the member on that line. *pahole* is a great tool and I strongly recommend its use in conjunction with DHAT.

```
int main()
{
  std::list<TestClass> tc;

  std::cout << "Size of TestClass "
            << sizeof(TestClass) << '\n';
  std::cout << "Size of std::string "
            << sizeof(std::string) << '\n';
  for (int i = 0; i < 1000; ++i)
  {
    tc.emplace_back();
  }
  int s{};
  for (auto const& elem : tc)
  {
    s += elem.getF1();
  }
  std::cout << "s " << s << '\n';
}
```

Listing 2

The second part of the example code is in Listing 2.

This doesn't do much. It prints out a couple of sizes to confirm what we saw with pahole. It adds 1000 default instances of `TestClass` to a `std::list`. It then iterates over the list reading and summing the `f1` member. Finally, it outputs the sum, which will be 0 since `f1` gets default value initialized.

Running the example

The output that I get is in Figure 2.

```
$ valgrind --tool=dhat ./main
==1148== DHAT, a dynamic heap analysis tool
==1148== Copyright (C) 2010-2024, and GNU GPL'd, by Mozilla Foundation et al.
==1148== Using Valgrind-3.25.0.GIT and LibVEX; rerun with -h for copyright info
==1148== Command: ./main
==1148==
Size of TestClass 40
Size of std::string 24
s 0
==1148==
==1148== Total:     60,096 bytes in 1,001 blocks
==1148== At t-gmax: 60,096 bytes in 1,001 blocks
==1148== At t-end:  4,096 bytes in 1 blocks
==1148== Reads:     29,080 bytes
==1148== Writes:    58,040 bytes
==1148==
==1148== To view the resulting profile, open
==1148==   file:///home/paulf/tools/valgrind/libexec/valgrind/dh_view.html
==1148== in a web browser, click on "Load...", and then select the file
==1148==   /home/paulf/scratch/accu/accu_dhat/dhat.out.1148
==1148== The text at the bottom explains the abbreviations used in the output.
```

Figure 2

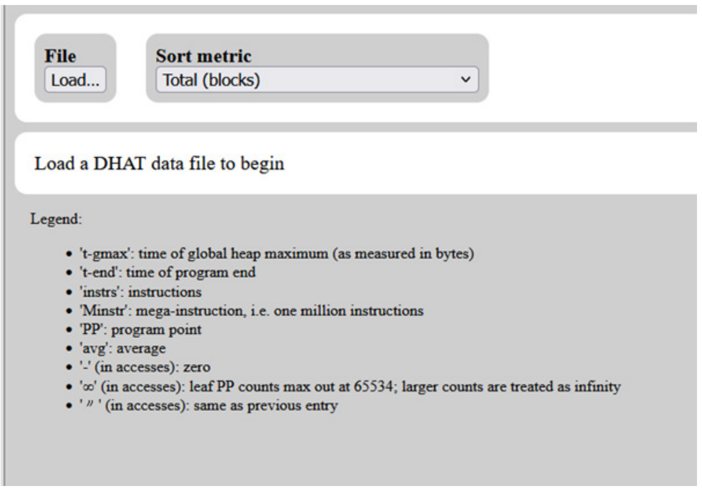
The number of bytes written are roughly the same as the number of bytes allocated, which makes sense. I'm not sure where all the bytes are being read.

Lines that start with `==1148==` are the console output from DHAT. The other lines are from the 'main' test executable. We can see most of what is happening from the summary. The `Total` is the total amount of memory allocated and the number of allocated blocks. I'll skip a line to `t-end`. DHAT uses its own terminology that can take some getting used to. `t-end` is at program end, and at that point there is one block of 4096 bytes. That block is allocated by libc by `fwrite` during the call to `std::cout` and FreeBSD libc does not free it.

Getting back to the `Total`, if `fwrite` uses 4096 bytes in 1 block that leaves 56000 bytes in 1000 blocks for `main()`. That is exactly what I was expecting. 1000 elements get added to the list, so each element is 56 bytes. We've already seen that `TextClass` is 40 bytes. The other 16 bytes are used by the `next` and `previous` pointers of the `std::list` nodes. `t-gmax` is the value at the global maximum, and it happens to be the same as the `Total`. Finally, there are the totals of the numbers of bytes read and written. The number of bytes written are roughly the same as the number of bytes allocated, which makes sense. I'm not sure where all the bytes are being read. I expect that the list traversal to calculate them `s` reads the list `next` (8 bytes) and `f1` (4 bytes) and the list destructor also does another traversal. That's 20 bytes. I guess that there is a 1 byte read per element to work out if the `f3` string needs to be deleted or not. There must be one more 8-byte read per element somewhere, giving a total of 29 per `TestClass` instance.

Viewing the results

I followed the instructions and opened the link in Firefox.



File
Load...

Sort metric
Total (blocks)

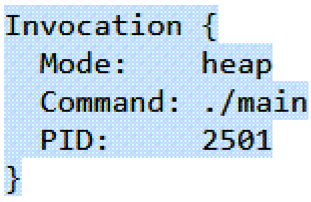
Load a DHAT data file to begin

Legend:

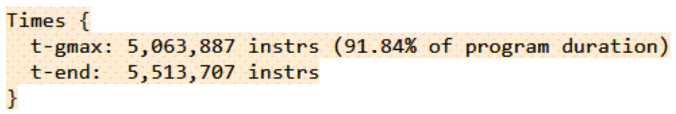
- 't-gmax': time of global heap maximum (as measured in bytes)
- 't-end': time of program end
- 'instrs': instructions
- 'Minstr': mega-instruction, i.e. one million instructions
- 'PP': program point
- 'avg': average
- '-' (in accesses): zero
- '∞' (in accesses): leaf PP counts max out at 65534; larger counts are treated as infinity
- ' 〃 ' (in accesses): same as previous entry

Note the **Legend**. I'll cover the **Sort metric** drop-down later.

Clicking **Load...** and opening a results file gives a complex screen even for this small example, so I'll break it up into small pieces.



```
Invocation {
  Mode:     heap
  Command:  ./main
  PID:      2501
}
```



```
Times {
  t-gmax: 5,063,887 instrs (91.84% of program duration)
  t-end:  5,513,707 instrs
}
```

Off to an easy start. That's just a summary of the executable and the PID that ran.

This is still quite simple. **Times** are really instruction counts, and this tells us when the peak memory occurs, and the total number of instructions executed.

Now for the hard bit. Before I treat you to some pretty colours[1], I need to make a stab at explaining what DHAT is doing. Basically, it is just doing two things.

1. Recording heap allocations (address, length, callstack). I'll call these allocation contexts.
2. Counting accesses to the heap allocations.

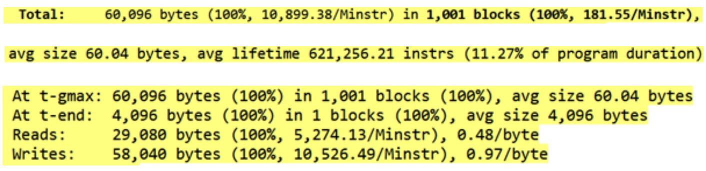
DHAT calls these allocation contexts 'Program Points' (PPs). The PPs get organized as a tree. The root of the tree represents the entire execution of the executable. Each PP is colour coded with darker colours meaning more blocks or memory. There is a threshold of 1% below which PPs do not get displayed.

Below the root there are three kinds of PP nodes:

1. The root node, coloured like the interior nodes.
2. Interior nodes. These are for allocation contexts that also contain other allocation contexts. They are coloured yellow if their child nodes are collapsed and blue of their child nodes are expanded.
3. Leaf nodes for functions that allocate by do not call any other allocating functions. They are colour coded green.

In the example that I'm using there is only a root node and a leaf node.

The following few pictures are of the root node. Before taking the pictures, I collapsed the children, making this yellow. Unfortunately, the viewer does not allow line wrapping.



```
 Total:     60,096 bytes (100%, 10,899.38/Minstr) in 1,001 blocks (100%, 181.55/Minstr),

avg size 60.04 bytes, avg lifetime 621,256.21 instrs (11.27% of program duration)

 At t-gmax: 60,096 bytes (100%) in 1,001 blocks (100%), avg size 60.04 bytes
 At t-end:  4,096 bytes (100%) in 1 blocks (100%), avg size 4,096 bytes
 Reads:     29,080 bytes (100%, 5,274.13/Minstr), 0.48/byte
 Writes:    58,040 bytes (100%, 10,526.49/Minstr), 0.97/byte
```

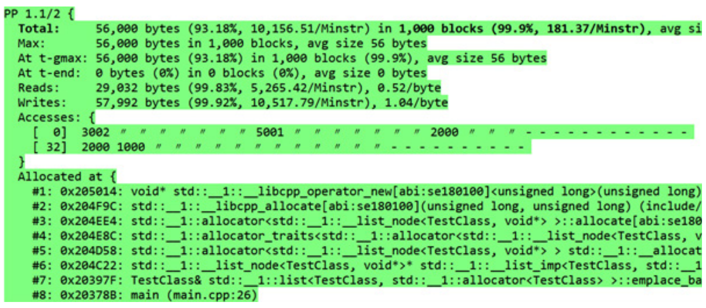
This looks quite like what we saw in the summary on the console, with some extra information.

1 If you access the online version of this article, all the screenshots are in colour.

the histogram or access map is only produced for allocations of 1024 bytes and less. This means that you won't see these maps for any large array-type allocations

Section	Data	Meaning
Total	Bytes 10,899.38/ Minstr	This is how many bytes get allocated per million instructions. Lower is better
Total	Blocks 181.55/ Minstr	This is the number of memory blocks allocated per million instructions. Lower is better.
Total	avg size 60.04 bytes	The average size per allocation.
Total	avg lifetime 621,256.21 instrs	This is the average number of instructions per block between allocation and deallocation. Lower is better, also shown as a %.
Reads	5,274.13/ Minstr	The average number of reads per million instructions. Higher is better. Very low or zero indicates a problem.
Reads	0.48/byte	The average number of reads per byte allocated.
Writes	10,526.49/ Minstr	The average number of writes per million instructions. Higher is better. Very low or zero indicates a problem. A value of one may mean objects are getting constructed and initialized and having no subsequent writes.
Writes	0.97/byte	The average number of writes per byte allocated.

Not so bad? On with an interior node.



```
PP 1.1/2 {
  Total:     56,000 bytes (93.18%, 10,156.51/Minstr) in 1,000 blocks (99.9%, 181.37/Minstr), avg si
  Max:       56,000 bytes in 1,000 blocks, avg size 56 bytes
  At t-gmax: 56,000 bytes (93.18%) in 1,000 blocks (99.9%), avg size 56 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     29,032 bytes (99.83%, 5,265.42/Minstr), 0.52/byte
  Writes:    57,992 bytes (99.92%, 10,517.79/Minstr), 1.04/byte
  Accesses: {
    [  0]  3002 ″ ″ ″ ″ ″ ″ ″ 5001 ″ ″ ″ ″ ″ ″ ″ 2000 ″ ″ ″ - - - - - - - - - - - -
    [ 32]  2000 1000 ″ ″ ″ ″ ″ ″ ″ ″ ″ ″ ″ ″ - - - - - - - - - -
  }
  Allocated at {
    #1: 0x205014: void* std::__1::__libcpp_operator_new[abi:se180100]<unsigned long>(unsigned long)
    #2: 0x204F9C: std::__1::__libcpp_allocate[abi:se180100](unsigned long, unsigned long) (include/
    #3: 0x204EE4: std::__1::allocator<std::__1::__list_node<TestClass, void*> >::allocate[abi:se180
    #4: 0x204E8C: std::__1::allocator_traits<std::__1::allocator<std::__1::__list_node<TestClass, v
    #5: 0x204D58: std::__1::allocator<std::__1::__list_node<TestClass, void*> > std::__1::__allocat
    #6: 0x204C22: std::__1::__list_node<TestClass, void*>* std::__1::__list_imp<TestClass, std::__1
    #7: 0x20397F: TestClass& std::__1::list<TestClass, std::__1::allocator<TestClass> >::emplace_ba
    #8: 0x20378B: main (main.cpp:26)
```

This is quite similar to the root PP node for most of the information. In order for the text to fit the **`Total`** line has been truncated as have the standard library function names in the **`Allocated at`** section. The **`Total`** line is similar to the previous PP. There are a few extras.

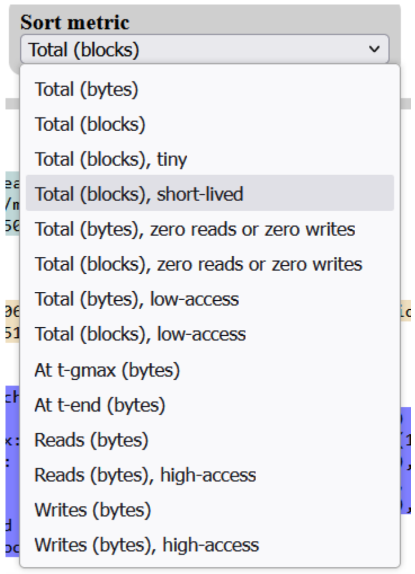
The **`Max`** line, showing the maximum memory for that leaf PP.

A summary of **`Accesses`**. This is the sum of all accesses for all allocations done at that callstack. This does not distinguish between reads and writes. This displays 32 bytes on a line with the access count for each byte. Ditto marks mean that the count is the same as the previous byte. A dash means a count of zero. The first 8 bytes have a count of 3002, probably the list **`previous`** pointer. The next 8 bytes were accessed 5001 times, probably the list **`next`** pointer. Then there are 4 bytes with an access count of 2000 – that's the **`f1`** member, each is zero initialized and read once in the sum loop. After that there are 12 bytes without any accesses. 4 of those bytes are the hole in the structure and 8 are for double **`f2`** that I deliberately did not initialize. The second line is the **`std::string f3`**. I guess that the first byte is being used as a tag to indicate SSO use with an access count of 2000. Then there are 13 bytes with an access count of 1000 corresponding to **`"small string\0"`**. Lastly there are 10 bytes with an access count of 0, the unused bytes in the SSO **`std::string`**. There isn't much that can be done in that case. Note that the histogram or access map is only produced for allocations of 1024 bytes and less. This means that you won't see these maps for any large array-type allocations (like **`std::vector`**).

The third thing is that there is the callstack that tells you where the allocations of this kind were done.

Sorting

Now I'll get back to the **Sort metric** dropdown list. This allows you to change how the display is ordered and filtered. Using this you can concentrate on specific things like peak memory, small allocations, high and low access rates.


Sort metric
Total (blocks)
Total (bytes)
Total (blocks)
Total (blocks), tiny
Total (blocks), short-lived
Total (bytes), zero reads or zero writes
Total (blocks), zero reads or zero writes
Total (bytes), low-access
Total (blocks), low-access
At t-gmax (bytes)
At t-end (bytes)
Reads (bytes)
Reads (bytes), high-access
Writes (bytes)
Writes (bytes), high-access

Larger 'access' maps

If you see a block of memory that is too big for the 1024-byte access map limit, but you would still like to look 'inside' it to see how it is being used there is way. You will need to instrument the code to enable this.

The first thing that you need to do is to include **`valgrind/dhat.h`**.

Secondly you need to use the **`DHAT_HISTOGRAM_MEMORY`** Valgrind client request macro, for instance:

```
  std::vector<uint8_t> vec(2000, 0);
  DHAT_HISTOGRAM_MEMORY(vec.data());
```

The macro just takes the address of the allocated block. In the example above, the limit has been raised to 2000. There is still a hard coded limit of 25600 on these user-specified access maps (25× the normal default).

This is still fairly limited for general use with C++ containers. For instance, if you have an **`std::vector`** that is not allocated up-front like in the example above then it's tricky to know when the allocated memory needs to grow and the new allocation flagged for profiling. You could track the vector **`capacity()`**. Or you could write a custom allocator – please contact me if you do!

Using the results

To round off this article, here are some ways that you can use DHAT.

1. Identify small, short-lived allocations and convert them to using the stack.
2. Identify 'dead data' (like dead code). These are data fields and entire structures that are never used. You may need to run several tests to get better 'data coverage' (like code coverage).
3. Improving cache hit rate. Look for high access counts with similar values in the access map that are more than 2 text lines in the report apart (corresponding to 64 bytes or 1 cacheline). Use *pahole* as a check, and tools like Linux *perf stat* and *perf record* to verify any performance changes.
4. Reduce the peak memory. Look for large allocations that have a long lifetime and see if that memory can be freed earlier. The kind of change that you will be looking to make is to patterns that look like

   alloc A; use A; alloc B; use B; free A; free B;

   where A is no longer needed after 'use A'. This can be transformed into

   alloc A; use A; **free A;** alloc B; use B; free B;

Don't forget that the 'free' might be due to the implicit destructor of a standard library container stored in an automatic variable. That means that 'free A;' may mean that you need to take explicit actions like **`A.clear(); A.shrink_to_fit();`** and 'free B' may just be the end of the scope.

Conclusion

In my opinion DHAT is a little-known hidden gem amongst the Valgrind tools. It is very slow, and the results can be difficult to read. There are no alternatives that I am aware of (other than instrumenting your own code to do the same sort of things). ■

References

[DHAT] DHAT: https://valgrind.org/docs/manual/dh-manual.html

[Floyd] Paul Floyd, DHAT viewer files (repo: paulfloyd/accu_dhat) https://github.com/paulfloyd/accu_dhat

[Floyd12] Paul Floyd, 'Valgrind Part 5 – Massif' in *Overload* 112, December 2012, available at https://accu.org/journals/overload/20/112/floyd_1884/

[Floyd13] Paul Floyd, 'Valgrind Part 6 – Helgrind and DRD' in *Overload* 114, April 2013, available at https://accu.org/journals/overload/21/114/floyd_1867/

[Github1] Heaptrack: https://github.com/KDE/heaptrack

[Github2] Dwarves: https://github.com/acmel/dwarves

[Gregg] Brendan Greg, 'Memory Leak (and Growth) Flame Graphs', available at https://brendangregg.com/FlameGraphs/memoryflamegraphs.html#Linux

[Massif] Valgrind user manual: https://valgrind.org/docs/manual/ms-manual.html

[Valgrind] Valgrind developers: https://valgrind.org/info/developers.html

[Wikipedia] List of performance analysis tools: https://en.wikipedia.org/wiki/List_of_performance_analysis_tools#C_and_C++

And this year's winners are...

In Overload:

1st place:
C++ Safety, In Context
by Herb Sutter

2nd place:
C++20 Concepts Applied – Safe Bitmasks Using Scoped Enums
by Andreas Fertig

3rd place:
User-Defined Formatting in std::format – Part 3
by Spencer Collyer

In CVu

1st place:
Private but not Hidden
by Pete Cordell

2nd place:
Array Thinking
by Francis Glassborow

3rd place:
Eleven C++11 Features Worth Knowing About
by Silas S. Brown

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

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

If you're reading this online, the article titles link to the articles. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones. If you're not a member yet, why not join?

Afterwood

Learning can be a lonely experience. Chris Oldwood tells us why he prefers learning in person.

I don't know if it was a New Year's resolution to resurrect the ACCU Cambridge meet-up, but 2025 will start with exactly that happening, as organiser Phil Nash kicks off the reboot with his own talk about the past, present, and future state of C++. Now that I'm working remotely practically full-time, having an ACCU meet-up in my neck of the woods is most welcome. A lawyer might argue that the pre-reboot social at a pub in Cambridge just before Christmas was the real reboot event, but January sees the actual return of the traditional format – a talk, book-ended with some socialising/networking.

The ACCU Cambridge meet-up holds a special place in my heart as it was the first meet-up I ever attended. Way back in late 2007 (not long after I joined ACCU) Jez Higgins gave an amusingly titled talk, 'Iteration: It's just one damn thing after another'. Up until that point, my only real sources of learning about the craft of programming were books, dedicated printed magazines such as *Dr Dobbs*, *C++ Report*, *MSJ*, etc. and – increasingly – articles on the Internet, such as the Artima Weblogs (sic).

Whichever way you look at it, it was all about the written word – a very solitary and passive experience. Sure, I talked with colleagues in the office, and we shared views on the best written content we came across, though primarily when it had practical implications for the kinds of systems we were building. When you're young and all working for the same company for a long time, it can create a form of echo chamber. Unless fresh blood joins the ranks and brings in experiences from farther afield – other cultures and industries – there is a danger of groupthink setting in, which is neither optimal for the employer or employee.

It would be easy to go along to the meet-up, listen to the presentation, and then leave; all without saying a word to anyone else. But then this would be no different to reading the transcript or watching the video later (not that that was really even an option back then). What I found most enjoyable from that meet-up experience was the interactivity, both with the speaker and the other attendees. Being Cambridge, there were a number of people from the embedded arena, a sector with very different constraints to those I'd personally experienced in a professional capacity. (Writing assembly language in your bedroom as a teenager might give you some technical empathy but does not prepare you for the commercial pressures of real-world software development.)

One consequence of that experience, and those meet-ups which followed, was that I attended my first ACCU Conference the following year in 2008. This was almost like back-to-back meet-ups, but where you also shared breakfast, lunch, and dinner with the other attendees too. I didn't write a review of my 2008 conference experience, mostly because I wasn't into writing back then (in fact, I abhorred it). However, Steve Love (amongst others), clearly helped me overcome my shyness a year later and I concluded my 2009 ACCU Conference review for *CVu* with "I know it's only my second year, but it lost none of the magic I experienced last year." Words to that effect appear to be my closing remark on my five subsequent ACCU conference reviews for *CVu* too.

Over 15 years later and I still find attending meet-ups and conferences a hugely enjoyable part of my learning process, whether hosted by ACCU or otherwise. Conferences in particular have provided a level of diversity of content that I might not have been exposed to if each session had been a separate article, book, or meet-up to attend. A conference allows you to leverage the locality of reference and amortise the cost of each session across the whole event, making it cheaper to step outside your comfort zone and attend talks which may not directly influence your current role, but could well contribute to your overall well-roundedness as a programmer. On some occasions an over-subscribed talk forced me to seek refuge elsewhere and I have subsequently been enlightened by a topic I didn't even know existed. I've never written a line of Scala, Clojure, Ruby, Lisp, or Haskell in my life, either professionally or for fun, but spending 45 to 90 minutes watching a talk on them moved those subjects (and related concepts) from the level of 'unconscious incompetence' to 'conscious incompetence'.

Naturally, The Four Stages of Competence is a learning model I first heard about through a meet-up, and the meta subject of 'learning about learning' usually makes one or two appearances at the meet-ups and conferences that cover a wider spectrum of programming topics than at ones focusing on a single technology. While I remember soaking up everything I could about C++ during those first few ACCU conferences (because it was my bread-and-butter) I purposefully attended talks about testing, databases, system's thinking, requirements analysis, architecture, etc. to help add colour to the craft that I knew I'd probably have to embrace anyway at some point in my career, even if I wanted to remain a hands-on software developer. (Conference keynotes can fill this void to some extent too, when used effectively, but it's becoming more common for them to remain technical, which I believe is a lost opportunity.)

For most of us, collaboration plays a significant role in our daily lives, and being able to communicate ideas well to a variety of stakeholders, whether they be customers, management, operations, testers, fellow developers, etc. makes us more productive if we can anticipate their problems ahead of time – to some degree – because we have an appreciation for their discipline too. Socialising with other people in a dedicated learning environment allows you to explore that without the pressure of the business setting biasing the conversation.

Even if you do choose to stick with what you know best and only attend events for your technology stack of choice, you'll still be greeted with an endless supply of interactive programming war stories to prepare you for the future, ones that you'd rarely get from reading incident post mortems or The Daily WTF. Plus, meeting some of your favourite authors and contributors can do wonders for confronting your Imposter Syndrome as you realise they're all mere mortals and don't, in fact, know everything there is to know about everything. ■

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

accu 2025

Conference 1–4 April

Pre-conference workshops 31 March

Online workshops 29 & 30 March, 12 April

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