

Join ACCU

Run by programmers for programmers,
join ACCU to improve your coding skills

- A worldwide non-profit organisation
- Journals published alternate months:
 - *CVu* in January, March, May, July, September and November
 - *Overload* in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website



professionalism in programming

www.accu.org

August 2024

ISSN 1354-3172

EditorFrances Buontempo
overload@accu.org**Advisors**

Paul Bennett
t21@angellane.org

Matthew Dodkins
matthew.dodkins@gmail.com

Paul Floyd
pjfloyd@wanadoo.fr

Jason Hearne-McGuinness
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

Jonathan Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover designOriginal design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Alison Peck. One of the large decorative iron balls at the entrance to the grounds of Culzean Castle, Ayrshire, Scotland.

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

4 User-Defined Formatting in std::format – Part 3

Spencer Collyer finishes his series by showing us how to apply specific formatting to existing classes.

6 In an Atomic World

Lucian Radu Teodorescu reminds us what atomics are, and how and when to use them.

13 Trip report: C++ On Sea 2024

Sandor Dargo shares an overview of his favourite talks and some emergent ideas.

16 Afterwood

Chris Oldwood is testing simplicity.

Copy deadlines

All articles intended for publication in Overload 183 should be submitted by 1st September 2024 and those for Overload 184 by 1st November 2024.

Copyrights and trademarks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request, we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) corporate members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from *Overload* without written permission from the copyright holder.

A Humble Proposal

Are you out of ideas or find it hard to speak up?

Frances Buontempo suggests small ways to get started.

I have been busy writing conference and workshop proposals recently, so haven't got around to writing an editorial. Each conference has a slightly different form to fill in, so even if I want to reuse a talk I need to format the idea with different word counts and different sections. I usually then spot different angles to emphasise and end up tweaking my slides too. It takes ages. As so often happens, a quick task ends up taking a very long time.

Reusing a talk and tweaking the angle is a relatively new experience for me, and even when people do reuse a talk, they need one to begin with. How do you find an initial idea? Most people who have never volunteered a talk think they don't have anything to say. That is not true. Everyone has managed to solve a problem, or wonders how things work, or learnt something in the first place. Consider suggesting a talk if you've never done this before and are in a place to take part in a conference. If you can't find the funds or spend the time travelling, there are hybrid and online only options. Of course, not everyone wants to spend hours listening to people speak about coding or to take part in a workshop. Some people have a life outside IT (if you don't, you might need to broaden your horizons). Nonetheless, speak up at work. You might be able to give a short talk to your teammates one day. Or suggest a different approach to a problem. If you're a team lead, find ways to encourage your team. Some people are too shy to speak up in meetings, so give them another way to share, for example let everyone write ideas on Post-its when you're discussing things.

If you are serious about giving a talk somewhere, watch the recent recording from MeetingCpp about giving talks [MeetingCpp]. The content is much broader than C++, and applies equally well to almost any technical talk. Various people gave short pieces of advice, from pacing the talk to live demo Pro Tips, and dealing with nerves. Tina Ulbrich's advice was called 'But I have nothing to talk about!' She pointed out we all have something to say really, and suggested listing what you do every day as a starting point. Your list proves you do know stuff. What sort of problems have you solved? Someone else might want to know. Alternatively, what do you want to learn more about? Giving a talk gives you a chance to learn and writing a talk will help you learn. Tina also pointed out that it is OK to talk about the basics. Talks don't all need to be new, shiny things. Leave that to the rock stars. Finally, talk about your ideas, otherwise you might talk yourself out of it. Of course, they might talk you into it.

If speaking isn't for you, writing is another option.

Perhaps you could write an article for us. I wrote some guidance a while ago [Buontempo23].

I gave some ideas on how to decide what to write about. You could summarise a

discussion, maybe from accu-general or somewhere on social media if you don't have a new groundbreaking innovation. If you only have the vaguest of ideas, feel free to email me and tell me your humble proposal. Maybe you don't want to write either, but may have ideas about articles you would like to see. Well, let us know. Again, accu-general might be the best place for this, or just tag ACCU somewhere on the socials. There are links on the bottom of the accu pages (<https://accu.org/>) if you want to connect.

I have told you about preparing proposals, creating conference talks and writing blogs. And yet, no editorial? Again? So, why did I decide to become *Overload's* editor? A very good question. Russel Winder asked me that once, and I owned up. He was amused, but understood, so perhaps I'll tell you as well. I had noticed I wasn't managing to read all the articles in *Overload* because other things crowded in. Ric Parkin was editor previously; he asked if anyone was interested in taking over back in 2012. Glad I updated the Wikipedia page so I knew a quick way to look this up [Wikipedia-1]. Unfortunately, since the ACCU website reorg, some links on Wikipedia were no longer correct so I updated them, giving me even less time to write an editorial. Going back to my main point, I volunteered, and Ric paired up with me for a couple of issues while I started to get the hang of (almost) everything. I now do read all the articles. As often happens, stepping up and volunteering to do something opens up more than you might expect. On a few occasions I have had emails from people I consider to be C++ 'Rock stars' which makes me feel like a complete fraud, but also delighted they have written for us or shared blogs to reach a wider audience. Being editor has forced me to stay up to date with various aspects of C++ and learn much more besides. If you would like a chance, we are open to guest editors, both for *Overload* and the members' magazine *CVu*. Get in touch. You could even write an actual editorial, putting the world right, for one edition only! Of course, you can do more than one if you want.

Any small idea can lead to unexpected consequences. I frequently find when writing an article or blog post, or preparing a talk, that I have some gaps in my knowledge or can't fully explain something. To my mind, the latter is often a sign you don't really understand a topic properly. Likewise, if you code alone, and never get input from anyone else, you might be missing out on better ways of doing things, or be relying on something which changes when you upgrade compilers or similar. You don't need a person to review your code, but it can be valuable. Paying attention to warnings, or using a static analysis tool can reveal potential issues. These may not help with the code structure though. Some complexity measures can help, for example cyclomatic complexity *et al* [Wikipedia-2]. A pre-emptive approach might be avoiding `if` statements altogether. If you have never tried this, give it a go. I am on a mission to avoid using booleans at the moment, and Spencer's 'Replacing bool Values'



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 Algorithms 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.

article for *Overload* has many alternatives and reasons why this is a good idea [Collyer21]. It is interesting to watch how one small change can have a big impact.

Not all small changes end up eating all your time or changing the world. You may have noticed a recent C++ proposal for `println` by Alan Talbot [P3142]. It means we will be able to say `println()` rather than `println("")`. The motivation section asks, “Why bother, it’s only two more characters.” It’s a small proposal, but Alan points out why it matters. The number of people who have mentioned the proposal is surprising. The comments usually start with “Why bother?” It is, after all, a very small thing. However, trying to keep things tidy as you go, like putting dirty bowls in hot soapy water to soak while you cook, is always a good thing. If you don’t try to keep things clean and tidy, you end up creating more work for yourself in the long run. Furthermore, the proposal has a great recipe at the end, so read it. If you do have a niggle about something small, do something about it. Whether that means submitting a C++ proposal, reporting what appears to be a compiler bug, or something smaller like reporting a bug in an app you use, you have the chance to make the world a better place.

We rebooted the ACCU Study Groups, using my *Learn C++ by Example* book [Buontempo24] recently. We haven’t had a study group for a while, which was a shame. The idea is for ACCU members to read through a book together and help each other learn. Sometimes the author has joined in, including Scott Meyers and Bjarne Stroustrup. Previously, we only used emails to discuss a book, but it felt like a good way to get to know other people and ask ‘dumb’ questions less publicly. This time we have set up a short video call once a week. Only a small handful of us make these, but it’s a great way to put faces to names. We hope to run another one later this year, using a different book. I suggested the reboot because I noticed several new members asking why they weren’t happening. It has eaten up an amount of time, but I brought this on myself; however, it’s been great. Inevitably, people involved have found typos and mismatches between the code in the repo and a few listings in the book. Fortunately, nothing major. Books always have mistakes. Fact. It is awkward reporting problems, but thanks to everyone who spoke up. I can add these to the errata and fix incorrect listing numbers and typos in comments in the repo. This might help other readers. If you spot a mistake in a book or a blog or an *Overload* article, let the creator know. Obviously, find a kind way to point out the problem, rather than being brash. The chances are someone else was wondering too. Several comments on the `println` proposal were along the lines of “Yeah, that was annoying me too.” It’s almost never just you.

The MeetingCpp session on ‘Talking about C++’ also mentioned questions in talks. These can seem frightening in theory if you aren’t used to public speaking, but it shows the audience are listening. It can also be difficult to ask if you’re in the audience. ‘What if this is obvious to everyone else, and I end up looking foolish?’ and related thoughts are common. Well, like the `println` proposal, you will probably find you are not alone. There’s no such thing as a dumb question. As a speaker, it is worth asking the audience questions too, because it helps you gauge if people are following. Maybe no one will answer, but don’t panic if that happens. Patrick Winston’s ‘How To Speak’ talk [Winston18] gives some advice about this, and other tips for speaking. He says to pause for

several seconds to give people a chance. If no one is willing to speak up, I sometimes go for a show of hands – people are less self-conscious about this.

Perhaps we don’t always step up because we get overwhelmed. Some people always barge in, either because they like the sound of their own voice and have missed the point, not realizing they don’t know much, or because most people are painfully aware they know little and prefer to keep quiet. Take a moment, and follow Tina’s advice to list what you do every day, maybe ignoring some of the distractions and rabbit holes you go down. You do know a lot. It might only be 20% of the language you use day to day, or how to fix a broken build, but you are an expert in a small section of your realm. We all know the Pareto principle: 80% of outcomes come from 20% of causes. (That’s about a fifth correct, but you get the idea.) You don’t have to know ALL the things, just some. If you don’t even have 20%, fear not. A European folk story tells of a group of travellers who arrive at a village with an empty cooking pot, which they fill with water and put over a fire. They put a single stone in the pot, and the villagers notice and ask what they are doing. “Making stone soup, but it could do with some extra flavour.” A villager adds carrots, and the next villager to come by asks a similar question. Eventually, you guessed it, the pot turns into a grand stew, so the stone is removed and everyone gets fed. Variants involve nails or an axe instead of a stone but they all make the same point [Wikipedia-3]. So, if you think you have nothing to offer, you are wrong. Speak up, question, or even volunteer to be guest editor. It’s about time *Overload* had an editorial.

References

- [Buontempo23] Frances Buontempo ‘How to write an article’ in *Overload* 178, December 2023, <https://accu.org/journals/overload/31/178/overload178.pdf#page=16>
- [Buontempo24] Frances Buontempo (2024) *Learn C++ by Example*, available at <https://www.manning.com/books/learn-c-plus-plus-by-example> or via O’Reilly’s subscription: <https://www.oreilly.com/library/view/learn-c-by/978163438330/>
- [Collyer21] Spencer Collyer ‘Replacing ‘bool’ values’ in *Overload* 163, June 2021, <https://accu.org/journals/overload/29/163/collyer/>
- [MeetingCpp] ‘Speaking about C++’ (<https://meetingcpp.com/>) On YouTube: https://www.youtube.com/watch?v=O-hR-u_jFIM
- [P3142] Alan Talbot: ‘Printing Blank Lines with `println`’, available at <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3142r0.pdf>
- [Wikipedia-1] *Overload* (magazine): [https://en.wikipedia.org/wiki/Overload_\(magazine\)](https://en.wikipedia.org/wiki/Overload_(magazine))
- [Wikipedia-2] Cyclomatic complexity: https://en.wikipedia.org/wiki/Cyclomatic_complexity
- [Wikipedia-3] Stone Soup: https://en.wikipedia.org/wiki/Stone_Soup
- [Winston18] Patrick Winston, ‘How to Speak’, transcript available at https://ocw.mit.edu/courses/res-tll-005-how-to-speak-january-iap-2018/bc92763ffa0dad0cafe44967e834e16_Unzc731iCUY.pdf



If you are thinking of writing for *Overload* (or *CVu*) but don’t know where to start, get in touch with our friendly editorial team. They will help you get from initial idea to finished article. We’re always looking for smaller articles, too – less than a page is OK – so start small! Contact overload@accu.org or cvu@accu.org

User-Defined Formatting in `std::format` – Part 3

We've seen formatting for simple classes and more complicated types. Spencer Collyer finishes his series by showing us how to apply specific formatting to existing classes.

In the previous articles in this series [Collyer24a], [Collyer24b] I showed how to write classes to format user-defined classes and container classes using the `std::format` library.

In this article I will show you how to create format wrappers, special purpose classes that allow you to apply specific formatting to objects of existing classes.

A note on the code listings: The code listings in this article have lines labelled with comments like `// 1`. Where these lines are referred to in the text of this article it will be as 'line 1' for instance, rather than 'the line labelled `// 1`'.

Format wrappers

I'd now like to introduce a type of class which I call 'format wrappers'. A format wrapper is a very simple class which wraps a value of another type. They exist purely so that we can define a `formatter` for the format wrapper. The idea is that the `formatter` will then output the wrapped value using a specific set of formatting rules. Hopefully this will become clearer when we discuss the `Quoted` format wrapper later.

A format wrapper is a very simple class, which normally consists of just a constructor taking an object of the wrapped type, and a public member variable holding a copy or reference to that value. They are intended to be used in the argument list of one of `std::format`'s formatting functions, purely as a way to select the correct `formatter`.

Quoted value

In C++14, a new I/O manipulator called `quoted` was added [CppRef]. When used as an output manipulator, it outputs the passed string as a quoted value. The value output has delimiter characters (the default is `"`) at the start and end, and any occurrences of the delimiter or the escape character (the default is `\`) have an extra escape character output before them.

For example, the following shows input strings on the left and what they would be output as on the right:

```
abcdef → "abcdef"
abc"def → "abc\"def"
ab\cd"ef → "ab\\cd\"ef"
```

If we want the same ability using `std::format`, we can create a format wrapper to do the work for us. An example of such a format wrapper and its associated `formatter` struct is `Quoted`, which is given in Listing 1, with sample output in Listing 2.

```
#include <format>
#include <iostream>
#include <string>
#include <string_view>

using namespace std;
struct Quoted // 1
{
    Quoted(string_view str) // 2
        : m_sv(str)
    {}
    string_view m_sv; // 3
};
template<>
struct std::formatter<Quoted>
{
    constexpr auto parse(format_parse_context&
        parse_ctx)
    {
        auto iter = parse_ctx.begin();
        auto get_char = [&]() { return iter
            != parse_ctx.end() ? *iter : 0; };
        char c = get_char();
        if (c == 0 || c == '\')
        {
            return iter;
        }
        m_quote = c; // 4
        ++iter;
        if ((c = get_char()) != 0 && c != '\') // 5
        {
            m_esc = c;
            ++iter;
        }
        if ((c = get_char()) != 0 && c != '\') // 6
        {
            throw format_error(
                "Invalid Quoted format specification");
        }
        return iter;
    }
    auto format(const Quoted& p,
        format_context& format_ctx) const
    {
        string_view::size_type pos = 0;
        string_view::size_type end = p.m_sv.length();
        auto out = format_ctx.out(); // 7
        *out++ = m_quote; // 8
        while (pos < end) // 9
        {
            auto c = p.m_sv[pos++];
            if (c == m_quote || c == m_esc) // 10
            {
                *out++ = m_esc;
            }
            *out++ = c; // 11
        }
        *out++ = m_quote; // 12
        return out; // 13
    }
}
```

Listing 1

Spencer Collyer Spencer has been programming for more years than he cares to remember, mostly in the financial sector, although in his younger years he worked on projects as diverse as monitoring water treatment works on the one hand, and television programme scheduling on the other.

```
private:
    char m_quote = '"';
    char m_esc = '\\';
};
int main()
{
    cout << format("{}\n",
        Quoted(R"(With "double" quotes)"));
    cout << format(":{:}'\n",
        Quoted("With 'single' quotes"));
    cout << format(":{:}'~'\n",
        Quoted("With 'single' quotes,
            different escape character"));
    cout << format(":{:}'"\n",
        Quoted(R"(Mixed "double" and 'single'
            quotes)"));
    cout << format("{}\n",
        Quoted(R"(Escaped escape character '\')"));
    cout << format(":{:}'~'\n",
        Quoted("Escaped escape character '~'"));
}
```

Listing 1 (cont'd)

The Quoted format wrapper

Line 1 starts the format wrapper. Because it is all public, we define it as a **struct** rather than a **class**. Line 2 defines the constructor which just copies the given **str** to **m_sv**, defined in line 3.

The parse function

The format specification for **Quoted** has the following form:

```
[ quote [ escape ] ]
```

The *quote* element is a single character that is the quote character to use as delimiters on the string. If not given it defaults to `"`.

The *escape* element is a single character that gives the escape character to use on the string. If not given it defaults to `\`. Note that you can only give an *escape* if you have already given a *quote*.

The first part of the **parse** function should be familiar from examples in my previous articles.

Line 4 picks up the first character and assigns it as the quote character.

Line 5 checks if we have reached the end of the *format-spec*, and if not it picks up the next character and assigns it as the escape character.

Line 6 is our normal check for reaching the end of the *format-spec*.

The format function

Before describing the **format** function in detail, please be aware that it is not optimised for speed, but has been kept simple as we are just using it as an example¹.

Line 7 picks up the current value of the output iterator from **format_ctx**, and then line 8 writes the delimiter preceding the string to it.

Starting at line 9, we iterate over all the characters in the string. For each character, line 10 checks if it is a quote or escape character, and if so outputs an escape character. Then line 11 outputs the actual character. As noted above, the speed of this loop could be improved although at the cost of making it more complicated.

Finally, line 12 outputs the delimiter after the string, and then line 13 returns the output iterator as the value of the function, as required.

Why use format wrappers?

After reading the description above you will hopefully understand the purpose of format wrappers. But you may ask the question, why would

¹ An optimised version would not process the string one character at a time. It would split the string up into substrings delimited by occurrences of the quote and escape characters, and output those substrings using `std::format_to`. That would generally be faster than checking and outputting each character individually

```
"With \"double\" quotes"
'With \'single\' quotes'
'With ~\'single~\' quotes, different escape
character'
"Mixed \"double\" and 'single' quotes"
"Escaped escape character '\\'"
"Escaped escape character '~'"
```

Listing 2

you want to use them? After all, you could just create a function that takes an object of the wrapped value and returns a value which can then be written to the output, without having to create a format wrapper class and the associated **formatter** struct.

There are two main reasons for using a format wrapper rather than a function, as follows.

1. If you use a function to do the formatting of the value, you have to return a value which is then output. Using a format wrapper this interim object is not required – the **format** function for the format wrapper can write the value directly to the output.
2. If your format wrapper's **formatter** allows for different formatting to be applied (as the one for **Quoted** does), that is simply done using a *format-spec* in the normal way. If you were to use a function to do the formatting instead, you could of course pass parameters to it indicating any changes required to the formatting, but there are a couple of disadvantages with doing that:
 - The formatting parameters for the value are separate from the format string, meaning anyone reading the code wouldn't see all the formatting instructions at a glance. You may think this is just an aesthetic problem and you aren't concerned about it, but to me it looks tidier to have all the formatting instructions in one place.
 - If or when you decide you need to internationalize your program, you may need to handle different formatting for the value, depending upon the expectations of various countries / languages as indicated by the locale. If the formatting instructions are embedded in the format string as a *format-spec* it is easy for translators to update them as required. However, if you are passing them as parameters to a function, you would need to add code to select the correct values to pass in based on locale. ■

References

[Collyer24a] Spencer Collyer, 'User-Defined Formatting in `std::format`', *Overload* 180, April 2024, available at <https://accu.org/journals/overload/32/180/collyer/>

[Collyer24b] Spencer Collyer, 'User-Defined Formatting in `std::format` – Part 2', *Overload* 181, June 2024, available at <https://accu.org/journals/overload/32/181/collyer/>

[CppRef] CPP Reference: `std::quoted`.

Available at <https://en.cppreference.com/w/cpp/io/manip/quoted>

Advertise in CVu & Overload

80% of readers make purchasing decisions or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for information.

In an Atomic World

Atomics form a relatively low level, but fundamental part of sharing data across threads. Lucian Radu Teodorescu reminds us what atomics are and how and when to use them

We often discuss mutexes as the basic building blocks of concurrency. However, there are more fundamental concepts upon which concurrent programs and synchronization primitives are constructed. The C++ language defines a *memory model*, which describes how programs behave when multiple threads are involved. Additionally, C++ introduces atomic operations that serve as foundation for working with data across threads, ensuring both safety and performance. The goal of C++ atomics is to closely align with the hardware and eliminate the need for lower-level operations that must work across threads.

The topic of atomics is often overlooked, and the prevailing advice is to avoid them. While this advice is generally sound, there are occasions when we need to use atomics to fully leverage the language's capabilities. This article aims to give atomics the attention they deserve, as they have yet to be featured in an *Overload* article.

The subject of atomics is extensive. For a comprehensive exploration, readers are encouraged to consult books by Anthony Williams [Williams19] and Mara Bos [Bos23]. While the Bos book primarily focuses on Rust, there is still much to be learned about atomics for C++ programmers. The reader can also consider cppreference.com for a quick reference to the atomics library [cppreference-1]. In this article, we will examine various memory ordering models and illustrate their usage through simplified practical examples.

Introduction to atomics

Many C++ applications share data between threads. If multiple threads are accessing the same data at the same time, and one of them is modifying the data, we have a *race condition*. In C++, a race condition leads to undefined behaviour.

A common solution for avoiding race conditions is the use of mutexes; when properly used, these can prevent race conditions. However, sometimes they are too heavy for some shared objects. For example, if we have a boolean flag that indicates when we need to stop a process, creating a mutex for this would be overkill. Instead of using mutexes, one can directly use `std::atomic<bool>` or `std::atomic_flag` for this example. In general, if the data being shared is just a primitive type (boolean, chars, integers, pointers), using atomics makes more sense.

An *atomic operation* is an indivisible operation. This means that no thread can observe partial results of applying that operation. If one writes an integer into a memory location, each thread that reads the value will either find the initial value or the new value.

An *atomic type* is a type that guarantees that all the operations that can be performed on the values of the type are atomic operations.

The C++ standard defines a series of atomic types. First, there is `atomic_bool` which is essentially a lock-free wrapper for atomic

operations on booleans. Then, we have the templated `atomic<T>`, which has default specialisations for integers, `bool`, `char` types, and pointers. In C++23, we also have dedicated specialisations for `shared_ptr` and `weak_ptr`. In its general form, `atomic<T>` can be instantiated when `T` is a trivially copyable type. For more information, please see [cppreference-2].

For most of the article, we will focus on types like `atomic<int>` and `atomic<bool>`.

Let's look at an example. Listing 1 shows a program that has two threads that communicate through an atomic object. The first thread waits for the atomic object to have the value `true`, and then prints something to the console. The second thread, after sleeping, sets the atomic object to the value `true`. If `notified` had not been an atomic object, this would have caused a data race, and thus, undefined behaviour.

Table 1 (next page) lists the main members of the `atomic<T>` class, where `T` is an integer type. All the operations are atomic, including those that contain multiple actions in their description.

Depending on the type `T` given to an `atomic<T>` instantiation, and depending on the platform, the atomic may or may not be lock-free. To determine whether an atomic object is lock-free, users can call the `is_lock_free` function on the object (there is also a `static constexpr` variant to indicate if an atomic type is always lock-free). The standard guarantees that `atomic_flag` is always lock-free; all other atomic types may not be lock-free. That is, the library may use mutexes to implement these atomic types.

Most common platforms have atomic types for integers and pointers that are lock-free. For this article, we will assume this is the case.

Atomics can be used as primitives to build all the other synchronisation primitives. We will show some examples in this article.

Thread views

Let's consider a function in a single-threaded application. In a naive world, the compiler would emit instructions that map exactly to the written program. But that would be slow. To prevent that, the standard allows the compiler to transform the code so that it can 'optimise' the

```
std::atomic<bool> notified{false};
std::thread t1{[&notified] () {
    while (!notified.load())
        std::this_thread::yield();
    std::print(
        "received signal from other thread");
}}
std::thread t2{[&notified] () {
    std::this_thread::sleep(100ms);
    notified = true;
}}
```

Listing 1

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

if a program writes multiple times to a variable without reading it, the compiler is allowed to generate code that will only write once to that variable

Member	Description
<code>is_lock_free</code>	check if the atomic object is lock-free
<code>operator=</code>	stores a value in the atomic object
<code>store</code>	stores a value in the atomic object
<code>load</code>	obtains the value of the atomic object
<code>operator T</code>	obtains the value of the atomic object
<code>exchange</code>	replaces the value of the atomic object and returns the previous value
<code>compare_exchange_weak</code> <code>compare_exchange_strong</code>	compare the value of the atomic object with a given value; if equal perform an exchange, otherwise a load
<code>wait</code>	block the thread until notified and the object value changes (C++20)
<code>notify_one</code>	notify at least one thread waiting on the atomic object (C++20)
<code>notify_all</code>	notify all threads waiting on the atomic object (C++20)
<code>fetch_add</code> <code>fetch_sub</code> <code>fetch_and</code> <code>fetch_or</code> <code>fetch_xor</code>	perform addition/subtraction/bit operation between the atomic object and a given value, and then return previous value
<code>operator++</code> (pre/postfix) <code>operator--</code> (pre/postfix)	increments/decrements the atomic object

Table 1

code, as long as the transformed code has the same observable behaviour as the original code. This is called the *as-if* rule [cppreference-3].

For example, if a program writes multiple times to a variable without reading it, the compiler is allowed to generate code that will only write once to that variable, the final value. Similarly, if the program reads from memory more than it should, the compiler is allowed to remove some of the unnecessary reads. The compiler is also allowed to reorder instructions so that memory accesses have patterns that would lead to faster operations or reorder instructions to minimise the latency to some of the memory operations.

After the compiler generates the binary code, the program can be further transformed. The CPU can reorder instructions when executing them. Again, execution reordering also happens under the *as-if* rule. Another type of code transformation is the effect that processor caches have on the execution of the code: not all the memory operations will reach the main memory in the order they were written.

We discuss all this to point out that there is a big difference between how we write the code, what's executed on a CPU core, and how the memory is actually accessed.

Things get a bit more complex when we look at multi-threaded applications, but the same *as-if* rule applies, with some caveats. One of these caveats is that there is no single 'observable behaviour' of the program. Each thread will have its own observable behaviour, its own *view* of the program execution. Then, we have a few rules that say how these views interact with each other. Besides the rules that apply when starting and ending a thread, C++ describes most of the interaction between views with the use of the *synchronises-with* relation.

We say that *A synchronises-with B* if operation *A* is a release store operation that produces results visible in operation *B*, which is an acquire load. We will cover the meaning of *release store* and *acquire load* below. When the two operations happen in different threads, the second thread has guarantees about the operations that happened before the store and the first thread has guarantees about the operations that happened after the load in the second thread. This *synchronises-with* relation is a fundamental relation that allows us to connect the views of different threads.

We will not describe the complex terminology that C++ standard has on this matter, but the goal is to be able to say that operation *A* happens before operation *B* (*A happens-before B*), when *A* and *B* are operations that may be executed on different threads. This allows us to have a partial ordering of the operations on different threads, even if we have multiple views.

Once again, the reader should bear in mind that one thread may see operation *A* happening before operation *B*, while another thread may see *B* happening before *A*.

Let's look at this graphically. In Figure 1, we have two threads (represented by two grey boxes), each containing two operations: the first thread has two stores for variables *x* and *y*, while the second thread loads the values of these variables from memory. Let's assume that both variables are initialised with 0 before the two threads run. The figure indicates with a

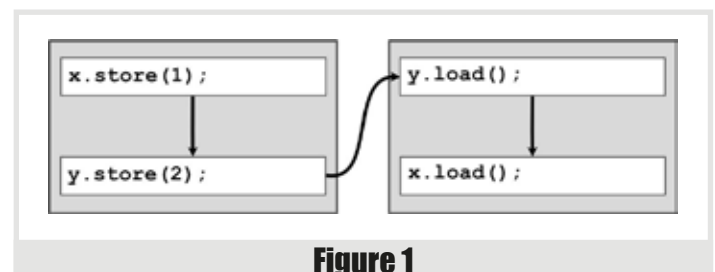


Figure 1

relaxed atomics allow us to use the same variable in multiple threads without race conditions

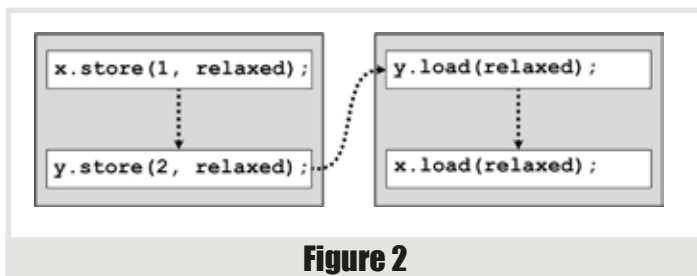


Figure 2

thick cross-thread arrow a *synchronizes-with* relation between the store of the `y` variable and the load from the second thread; that is, for this particular execution, `y.load()` will obtain the value that was written by `y.store(2)`. Using a sequentially-consistent ordering (more on this later), we make sure that the store to `x` happens before the store to `y`, and that the load from `y` happens before the load from `x`. All these ordering relations are seen by both threads and are represented by thick full arrows. Thus, in all the views, the load from `x` will see the value 1 as written from the first thread (assuming there is no other intermediate store to this variable by a different thread).

Figure 2 shows the same type of instructions, but with relaxed memory ordering (to be covered shortly). In this case, the relation between the operations on the same thread are enforced only in the local view of that thread; we denote this by using thinner, dotted arrows. For the first thread, the store to `x` happens before the store to `y`. But the second thread might see those writes out of order; it might see the store to `y` before the store to `x`. Thus, the second thread might load value 2 for `y`, but a value of `x == 0`.

This illustrates the idea that threads might have different views on the order of operations. One of the most frequent causes for this behaviour is the use of processor caches.

```
std::atomic<int> x = 0;
std::atomic<int> y = 0;
std::thread t1{[&] {
    for (int i = 0; i < 1000000; ++i) {
        x.store(i, std::memory_order_relaxed);
        y.store(i, std::memory_order_relaxed);
    }
}};
std::thread t2{[&] {
    int count_mismatch = 0;
    for (int i = 0; i < 1000000; ++i) {
        auto yy = y.load(std::memory_order_relaxed);
        auto xx = x.load(std::memory_order_relaxed);
        if (xx < yy) count_mismatch++;
    }
    printf("Mismatch count: %d\n", count_mismatch);
}};
t1.join();
t2.join();
```

Listing 2

The difference between different thread views can be seen by running the code in Listing 2. From the perspective of the first thread, we have either `x == y` or `x == y+1`; thus `x >= y`. This inequality doesn't translate well in the view of the second thread. Here, we load `y` before `x`. Thus, `x` should always be greater than `y`, either because that was the state of `x` and `y` when the second thread started reading, or because `x` got the chance to have a newer (greater) value between the two loads. However, when we look at the result, we see that occasionally we have `xx < yy`, thus we get the counter incremented, and the program prints a non-zero mismatch count. A non-zero value printed out proves that the two threads might have different views. On my machine (MacBook Pro, Apple M2 Pro processor), for a given run, I've obtained 258 mismatches. (Note, a non-zero result is not guaranteed, but on many machines, it is a probable result).

Memory ordering

The way that the thread views are aligned for atomic operations depends on the *memory ordering* applied to the atomic operations. The C++ standard defines the following memory ordering values: *relaxed*, *consume*, *acquire*, *release*, *acquire-release*, and *sequentially-consistent*. We won't discuss the *consume* memory ordering here; it is not well supported by the compilers, and the rules surrounding it are hard to reason about.

Each atomic operation uses one of these memory ordering values. The most relaxed, as the name suggests, is the *relaxed* mode; this mode doesn't add any ordering guarantees. On the other side, the *sequentially-consistent* model, which is the default, will ensure that all the threads have the same views on the ordering.

To discuss memory ordering, we will refer to how different instructions are translated to machine code. For this, we will focus on just two architectures commonly found in practice: x86-64 and ARM64. The x86-64 platform is a good example of a strongly-ordered architecture, while the ARM64 platform is a good example of a weakly-ordered architecture [Preshing12]. As we shall see below, strongly-ordered architectures have stronger guarantees than weakly-ordered architectures for regular operations.

Relaxed memory ordering

Atomic operations that use relaxed memory ordering will typically generate code as if no atomics were used. Table 2 (on next page) shows the generated code for x86-64 and ARM64 for storing 0 in an integer in two different ways: using a plain `int` type and using `atomic<int>` with a relaxed store. On neither platform is there a difference between the two operations.

A similar thing happens for loads. Table 3 shows that there is no difference between the generated code for loading the value from an `int*` object or from an `atomic<int>` using relaxed memory order, on neither x86-64 nor ARM64. From these, we can conclude that relaxed atomic load/store operations behave like regular loads/stores.

C++ code	x86-64	ARM64
<pre>void s1(int* x) { x = 0; } void s2(atomic<int>& x) { x.store(0, memory_order_relaxed); }</pre>	<pre>s1(int*): mov DWORD PTR [rdi], 0 ret s2(atomic<int>&): mov DWORD PTR [rdi], 0 ret</pre>	<pre>s1(int*): str wzr, [x0] ret s2(atomic<int>&): str wzr, [x0] ret</pre>

Table 2

C++ code	x86-64	ARM64
<pre>int ld1(int* x) { return *x; } int ld2(atomic<int>&x) { return x.load(memory_order_relaxed); }</pre>	<pre>ld1(int*): mov eax, DWORD PTR [rdi] ret ld2(atomic<int>&): mov eax, DWORD PTR [rdi] ret</pre>	<pre>ld1(int*): ldr w0, [x0] ret ld2(atomic<int>&): ldr w0, [x0] ret</pre>

Table 3

If there is no difference in the generated code between a relaxed atomic and the regular code, the reader might rightfully ask why we need relaxed atomics. The answer is twofold. First, relaxed atomics allow us to use the same variable in multiple threads without race conditions. Second, the usage of atomics will prevent the compiler from doing certain operations: it can't omit memory loads and memory stores, and it cannot reshuffle the code to generate more instructions than needed. For example, for regular integers, an operation like `x = 1` can be transformed by the compiler to look like `x = 0; x++;` – such a transformation is forbidden when using atomics.

In terms of performance, relaxed atomics don't incur additional costs compared to regular instructions. They do prevent certain compiler transformations, so using relaxed atomics, depending on their use, might be slower than not using atomics at all. In general, we should limit the usage of atomics to just essential variables needed for synchronisation between threads.

As mentioned above, using relaxed atomics won't make the operations viewed by a thread consistent with what other threads view. See Figure 2 and Listing 2.

A typical usage for relaxed operations is in incrementing counters, since most of the time we only require atomicity; note, however, that decrementing counters typically require acquire-release synchronisation.

Acquire and release memory ordering

Acquire and release typically work together as they create *synchronises-with* relations. Acquire memory ordering only has meaning for load operations, while release memory ordering only applies to stores. For operations that perform both stores and loads (like `compare_exchange_strong`), one can use the `std::memory_order_acq_rel` memory order.

When one performs a store that has a release ordering, it forces all the previous operations to make their effect visible before the store is actually visible. The C++ standard states that no read or write operations before a release operation can be reordered to happen after the release store.

When one performs a load that has an acquire ordering, it forces all the subsequent memory loads to use data obtained after the acquire operation.

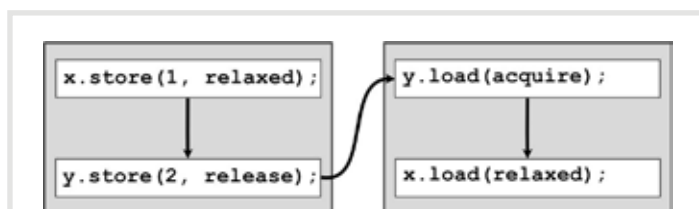


Figure 3

The C++ standard states that no read or write operations after the acquire operation can be reordered to happen before the acquire load.

A pair of a release store and an acquire load can form a *synchronises-with* relation. Figure 3 shows this relation, being similar to the examples shown in Figures 1 and 2. If we make the store to `y` use release memory ordering, and the load from `y` use acquire memory ordering, then the two threads can form a *synchronises-with* relation when the second thread reads the value that was published by the first thread. That means we can gain consistency between the views of the two threads. Even if the store to `x` is relaxed in the first thread, because the store to `y` has release memory ordering, the second thread will have to see its effect when acquiring the value of `y`; this means that the second thread will always see `x` stored before `y`. This means that the four operations in Figure 3 are perfectly ordered, and both threads see the same thing.

To prove this, we can slightly adapt the code in Listing 2. For the line containing `y.store`, we can use `std::memory_order_release` instead of relaxed ordering; also, we can use `y.load(std::memory_order_acquire)` instead of relaxed memory order. Making these two changes will ensure that the code in Listing 2 guarantees that the printed number of mismatches is zero. The reasoning is exactly the same as the one we had for Figure 3.

Taking a lock has acquire semantics: no instructions after the point where the lock is taken can be seen by threads that use the lock prior to taking

```
std::atomic<bool> locked_flag = false;
int counter1 = 0;
int counter2 = 0;
auto f = [&] {
    for (int i = 0; i < 1000000; ++i) {
        // Simulate acquiring the lock.
        while (locked_flag.exchange(true,
std::memory_order_acquire))
            ;
        // Protected operations.
        counter1++;
        counter2 = counter1;
        // Simulate releasing the lock.
        locked_flag.store(false,
std::memory_order_release);
    }
};
std::thread t1{f};
std::thread t2{f};
std::thread t3{f};
t1.join();
t2.join();
t3.join();
assert(counter1 == 3000000);
assert(counter2 == 3000000);
```

Listing 3

C++ code	x86-64	ARM64
<pre>void store3(atomic<int>& x) { x.store(0, memory_order_release); } int ld3(atomic<int>&x) { return x.load(memory_order_acquire); }</pre>	<pre>store3(atomic<int>&): mov DWORD PTR [rdi], 0 ret ld3(atomic<int>&): mov eax, DWORD PTR [rdi] ret</pre>	<pre>store3(atomic<int>&): stlr wZR, [x0] ret ld3(atomic<int>&): ldar w0, [x0] ret</pre>

Table 4

C++ code	x86-64	ARM64
<pre>void store4(atomic<int>& x) { x.store(0, memory_order_seq_cst); } int ld4(atomic<int>&x) { return x.load(memory_order_seq_cst); }</pre>	<pre>store4(std::atomic<int>&): xor eax, eax xchg eax, DWORD PTR [rdi] ret ld4(std::atomic<int>&): mov eax, DWORD PTR [rdi] ret</pre>	<pre>store4(std::atomic<int>&): stlr wZR, [x0] ret ld4(std::atomic<int>&): ldar w0, [x0] ret</pre>

Table 5

the lock. Similarly, releasing a lock has release semantics: no instructions before the lock is released can be reordered to be seen by the threads using the lock after the release point. This is demonstrated by the code in Listing 3. This code will ensure that the modifications to the two counter variables are done in protected regions, without any race conditions.

Let’s now look at the translation of acquire loads and release stores on the two major platforms. Table 4 shows how the translations look. Comparing this with the translation from Table 3, we conclude that on x86-64 there is no difference between relaxed loads and acquire loads, and between relaxed stores and release stores; the same code is generated. However, on ARM64, the astute reader will spot the differences. A relaxed store is translated into **str** (store register), while a release store is translated into an **stlr** instruction (store-release register); similarly, a relaxed load is translated into **ldr** (load register), while an acquire load is translated into **ldar** (load acquire register).

On weakly-ordered platforms like ARM64, release/acquire operations are more expensive than relaxed atomic operations. On strongly-ordered platforms like x86-64, there is no difference between relaxed atomics and release/acquire atomics. One can say that release/acquire operations are as cheap as relaxed memory operations, but also that relaxed memory operations are as expensive as release/acquire operations.

Sequentially consistent memory ordering

There is another ordering that provides more guarantees than acquire-load memory ordering, but can also be more expensive. This is sequentially-consistent memory order. In C++, this is represented by the `std::memory_order_seq_cst` enum value, and it is the default memory ordering for atomic operations.

Table 5 shows the generated instructions for the two selected platforms. Here, it’s interesting to notice that on ARM64, the same instructions are generated as for acquire/release operations. However, there are differences on x86-64 platforms; while the load operation generates the same instruction, the store translates into different code. For this, the **xchg** (exchange register/memory with register) instruction is used; this is typically used for swaps and performs a store and a load operation. This indicates that a sequentially-consistent store on x86-64 is more expensive.

Sequentially-consistent memory ordering provides more guarantees compared to acquire/release semantics. In particular, it guarantees a single total modification ordering of all the operations that are tagged. By contrast, acquire/release semantics only add constraints between two threads, and they are not concerned with global ordering.

To spot the difference let’s look at the code from Listing 4 (assuming each function is called in parallel on a different thread). The **read_x_then_y** function waits until **x** becomes true, and then loads **y**; if **y** is true, then it increments **z**. The **read_y_then_x** function does the same things but

swaps **x** and **y**. They are just reading the atomic values, so using acquire/release semantics we can’t make those two functions agree on the order in which **x** and **y** become **true**. In the acquire/release world, there is no relation that synchronises their view of the variables. Thus, if this code were to use acquire/release semantics, after running all the functions in parallel, we might end with a zero value of **z**. That is, **read_x_then_y** would see **x==true** and **y==false**, while **read_y_then_x** would see **y==true** and **x==true**.

This cannot happen in sequentially-consistent memory ordering (the memory ordering used in the example); both read functions would see the same order in which **x** and **y**. To repeat, in the acquire/release model, we can only synchronise between the view of the thread that does the release and the view of the thread that does the acquire. In the sequentially-consistent model, we synchronise between all thread views.

Some techniques and real-life examples

Mutexes and busy loops

We’ve seen in Listing 3 a sketch of building a mutex on top of atomic operations. If there is no contention on the lock, this implementation might be fast and appropriate. But, as soon as one thread needs to wait, we have a performance problem; the thread will spin in a tight loop and consume all its CPU quota.

Possible alternatives here include introducing loop instructions that will briefly pause the thread. Processor yields, thread scheduler yields, or even sleeps are good strategies for pausing the threads (each having different pausing times and characteristics).

Luckily for us, C++20 provides a portable way to do this waiting on spin-loops. Listing 5 (next page) shows how the code needs to be modified

```
atomic<bool> x = {false};
atomic<bool> y = {false};
atomic<int> z = {0};
void write_x() {
x.store(true);
}
void write_y() {
y.store(true);
}
void read_x_then_y() {
while (!x.load());
if (y.load()) ++z;
}
void read_y_then_x() {
while (!y.load());
if (x.load()) ++z;
}
```

Listing 4

```
// Acquire the lock.
while (locked_flag.exchange(true,
    std::memory_order_acquire))
    locked_flag.wait(true,
        std::memory_order_relaxed); // NEW
...
// Release the lock.
locked_flag.store(false,
    std::memory_order_release);
locked_flag.notify_one(); // NEW
```

Listing 5

to incorporate this into our spin-mutex implementation. In this code, the `wait` call waits until `locked_flag` is not `true` anymore; this will use the best strategy known for the detected processor/OS type (often a combination of the above). The `notify_one` call will ensure that the `wait` call wakes up on platforms where `wait` sleeps in kernel space. See [Doumler20] for a discussion on building spin-mutexes and [Giroux] for a possible implementation of `wait`. Also see [Pikus22] for a quick discussion on building spin mutexes.

Reference counting

Listing 6 shows how a reference counting mechanism can be implemented, similar to what we find in `std::shared_ptr`. As long as we have a valid reference to the object (at least one `inc` is called without `dec`), the object is kept alive; as soon as we drop all the references to this ref-counted object, the inner object will be destructed too. Here, the assumption is that the first time we call `inc()` to have a non-zero reference, we can have at most one thread; this typically happens in the constructor.

When calling `inc`, we don't care about synchronising thread views. There are no loads and stores that depend on the ordering of this (assuming that `dec` is properly ordered after `inc`, which is ensured by other mechanisms). This means that a relaxed memory order is enough for what we need.

When calling `dec`, things are slightly different because we are also touching `data_`. In the thread that calls `dec`, all the memory access to `data_` before the call to `dec` should not be moved after the call to `dec`; this implies release semantics. Also, the code that deletes `data_` (when the counter is decremented to zero) which accesses its content should not be reordered before the `fetch_sub` call on the atomic; this implies `acquire` semantics.

In this example, we've shown the use of `memory_order_acq_rel`, that is commonly used with atomic operations that perform both reads and writes. Prime examples of such instructions are `fetch_add` and `fetch_sub`, also shown in this example.

CAS loops and a simple stack example

The C++ standard provides a few examples of atomic operations that do both reads and writes. However, it can't provide all the operations that a user might need. There needs to be a technique for users to implement atomic operations out of simpler operations.

Such a technique is Compare-and-swap (CAS) loop [Wikipedia]. In C++ this can be achieved with the use of `compare_exchange_weak` and

```
template <typename T> class ref_counted {
    atomic<int> count_;
    unique_ptr<T> data_;
public:
    void inc() { count_.fetch_add(1,
        memory_order_relaxed); }
    void dec() {
        if (count_.fetch_sub(1,
            memory_order_acq_rel) == 1)
            data_.reset();
        }
    // ...
};
```

Listing 6

`compare_exchange_strong` methods of `atomic<T>`; these methods change the atomic to a desired value if they have an expected value. To make this work, these instructions are typically put inside a loop.

The signature of these methods, in their most generic form, is `bool compare_exchange_strong(T& expected, T desired, memory_order success, memory_order failure) noexcept;` (same for the `_weak` version). This will compare the value in the atomic object to `expected`; if they have the same value, it will store `desired` in the atomic object and return `true`. If they have a different value, it would update `expected` to the current value in the atomic object and return `false`. Updating the `expected` value in case of a failure can be very useful in CAS loops, as most of the time we need to check the current value of the atomic object in case of a failure. The `success` memory order is the one that should be applied to the read-modify-write if the operation succeeds; the `failure` memory order is the one that should be applied to the load operation in the case of a failure.

Unlike the `_strong` version, the `_weak` version is allowed to fail spuriously; that is, it can return `false` even if the atomic object has a value equal to `expected`. On some architectures, the only way to implement the `_strong` version is to have inner loops; but, as we often put these constructs inside other loops, we can directly utilise the `_weak` version. Thus, the rule of thumb is that, if the compare-and-swap is put in a loop, one should use the `_weak` version, and if not, one should use the `_strong` version.

Listing 7 presents an example of implementing the operation of pushing into a stack. After creating the new node, we need to chain it to the head of our singly-linked list. This requires two changes: the `next_` pointer of the new node needs to point to the current head, and the head needs to point to the new node. These two changes need to happen atomically. To make this happen, we try to set the current list head as the element following the new node, and then atomically compare-and-swap the head node. If the `compare_exchange_weak` operation succeeds, we still have the same `head_` value, and the connection is made correctly; if the operation fails, then we will load the new head, putting it directly into `new_node->next_` (as a side effect of the `compare_exchange_weak` instruction) and try again. The chances of converging the loop in a small number of iterations are very high even in the case of contention.

The reader might remark that `next_` is not an atomic type. It doesn't need to be, as we always interact with it through the `head_` pointer. Storing any value to it will synchronise with other threads that might read data from it (not shown in our example), as the store of `head_` has release semantics; this ensures that all the previous stores will be visible before the store to `head_` (for the threads that synchronise with this thread on an acquire-load of `head_`).

We don't have any subsequent load operations that need to be synchronised with the exchange of the head pointer. This means that we don't need acquire semantics for all the loads we have. This means that we can safely use relaxed loads when reading the content of `head_`.

```
template <typename T> struct node {
    T data_;
    node* next_;
    node(const T& data) : data_(data), next_(nullptr) {}
};
template <typename T> class stack {
    atomic<node<T>*> head_;
public:
    void push(const T& data) {
        node<T>* new_node = new node<T>(data);
        new_node->next_ =
            head_.load(memory_order_relaxed);
        while (!head_.compare_exchange_weak(
            new_node->next_, new_node,
            memory_order_release,
            memory_order_relaxed));
    }
};
```

Listing 7

We should reiterate a key point of this algorithm: the `new_node->next_` is updated when the `compare_exchange_weak` operation fails. This actually prepares the new node for adding it again to the head of the list, thus ensuring the precondition needed for changing the head of the stack. Without this, the loop of the `while` instruction would have some more operations.

Some discussions

Atomics are very powerful. They allow programmers to exploit low-level primitives for the platform in order to generate fast concurrent code. One can build all concurrency primitives on top of the atomics library provided by the C++ standard, and they would probably be very fast. Atomics lack deep integration with the OS thread scheduler, but other than that, they have the potential to express most concurrency primitives in the most efficient way.

However, with great power comes great responsibility and a higher chance of shooting yourself in the foot. If C++, in general, is considered a language in which one can easily shoot themselves in the foot, using atomics is like juggling with hand grenades. In this section, we cover a few reasons why we should avoid using atomics for most of the programs we write.

Atomics break local reasoning

Atomic primitives are like `goto` instructions. They are fundamental at a lower level (e.g., generated machine code), but should be avoided in high-level programming. One of the reasons for this is that they both break local reasoning.

Local reasoning, a fundamental idea in structured programming, is the ability to look at a code fragment in isolation and determine its meaning, without knowing what the rest of the program does.

One key aspect of reasoning about atomics is understanding how different thread views are correlated. That is, to reason about a code fragment that runs on a thread, we need to understand how the other threads are viewing different operations. Local reasoning is possible only when thread views are either equivalent or they don't intersect at all.

Using release/acquire semantics, one needs to reason about which stores are "published" on a release operation, and which loads are guaranteed to be ordered on an acquire operation. Oftentimes, this reasoning expands beyond the local code that uses atomics. A good example of this is the primitives used to build a spin mutex (see Listing 3). The acquire semantics that apply to the step of acquiring a lock need to apply to the protected region; similarly, the effects of the protected region need to be "published" by the release semantics of the unlocking part.

As it's easy to misuse atomics, and as bugs are typically hard to catch (the code can run in production for years before the bugs manifest), reasoning about the code that uses atomics is of utmost importance. And, at this point, we don't seem to have good tools to help us dealing with atomics.

Release/acquire is better than sequentially consistent semantics

The C++ standard takes the stand that the default memory ordering is sequentially consistent, even if it is not needed most of the time. It is the ordering that provides the most guarantees, thus it's safer. While this is true, it may encourage less reasoning, which may lead to bugs.

As argued above, reasoning about atomics is crucial. The reasoning process needs to include the guarantees that an operation needs in order to ensure correctness. But, if we fully reason about the use of atomics, then it makes little sense to use a suboptimal setting when it's clear that it's suboptimal.

Using sequential consistency atomics is often a sign that the reasoning about the atomics was not carried through to the end. Thus, similar to the position of Mara Bos [Bos23], we recommend using the appropriate memory ordering, instead of sticking to the default of sequentially consistent. And, of course, documenting the rationale helps.

Document the choice of memory ordering

The reasoning performed when choosing the memory model should not be easily lost. Thus, we recommend adding comments to document such reasoning. For example, after an acquire load, document what other loads are dependent on this load. Also, for a release store, document which other stores need to be published. If using sequential consistency, document the cases in which release/acquire semantics are not enough.

As using atomics often implies non-local reasoning, documenting expected behaviour, especially in relation to non-local items, is important for later readings of the code.

Prefer using higher-level concurrency abstractions

Using mutexes is generally easier than using atomics for most use cases. Using barriers and latches can be easier than using mutexes and conditional variables. Although atomics can be more efficient, not all code needs to be optimised to the maximum. We should measure the performance of a concurrent system before deciding to fully optimise it. Thus, for most projects, using higher-level concurrency primitives is the right thing to do.

Continuing on this line of thought, we should use concurrency primitives that invite users to express high-level concurrency constraints, such as dependencies between two or more work chunks (see [Teodorescu24]). In C++ one can use the senders/receivers framework [P2300R10] (maybe with extra utilities on top of it)¹.

Using fast low-level concurrency is typically a worse strategy than using appropriate high-level concurrency primitives for many applications. This is true both in terms of maintainability and performance.

Atomics are powerful primitives, but perhaps too powerful for most applications. Maybe raw power is not always the answer. Having good strategies often produces better results. ■

References

- [Bos23] Mara Bos, *Rust Atomics and Locks: Low-Level Concurrency in Practice*, O'Reilly Media, 2023.
- [cppreference-1] *cppreference*, 'Concurrency support library', <https://en.cppreference.com/w/cpp/thread>, accessed June 2024.
- [cppreference-2] *cppreference*, `std::atomic`, <https://en.cppreference.com/w/cpp/atomic/atomic>, accessed June 2024.
- [cppreference-3] *cppreference*, 'The as-if rule', https://en.cppreference.com/w/cpp/language/as_if, accessed June 2024.
- [Doumler20] Timur Doumler, 'Using locks in real-time audio processing, safely', posted on Timor.Audio on 14 April 2020, <https://timur.audio/using-locks-in-real-time-audio-processing-safely>.
- [Giroux] Olivier Giroux, 'Sample implementation of C++20 synchronization facilities' on GitHub, https://github.com/ogiroux/atomic_wait/, accessed June 2024.
- [P2300R10] Michał Dominiak, et al, P2300R10: `std::execution`, ISO/IEC JTC1/SC22/WG21, <https://wg21.link/P2300R10>.
- [Pikus22] Fedor Pikus, A Spinlock Implementation (Lightning Talk), CppCon 2022, <https://www.youtube.com/watch?v=rmGJc9PXpuE>.
- [Preshing12] Jeff Preshing, 'Weak vs. Strong Memory Models' posted on *Preshing on Programming* (blog) on 30 Sept 2012, <https://preshing.com/20120930/weak-vs-strong-memory-models/>
- [Teodorescu24] Lucian Radu Teodorescu, 'Concurrency: From Theory to Practice', *Overload* 181, June 2024, <https://accu.org/journals/overload/32/181/teodorescu/>.
- [Wikipedia], Wikipedia, Compare-and-swap, <https://en.wikipedia.org/wiki/Compare-and-swap>.
- [Williams19] Anthony Williams, *C++ concurrency in action (2nd edition)*, Manning, 2019.

1 If the reader hasn't heard yet, it is my great pleasure to announce that the P2300 paper was voted, in the June 2024 plenary in St. Louis, to be included in the upcoming C++26 standard.

Trip report: C++ On Sea 2024

C++ On Sea took place in Folkestone again in February this year. Sandor Dargo shares an overview of his favourite talks and some emergent ideas.

Last week, between the 3rd and 5th of July, I had the privilege of attending and presenting at *C++ on Sea 2024* [CPPoS-1] for the 5th time in a row! I'm grateful that the organizers accepted me not simply as a speaker, but that they granted me a double slot to deliver a half-day workshop about how to reduce the size of a binary. I'm also thankful for my management that they gave me the time to go to Folkestone and share our knowledge on binary size. Last but not least, great thanks goes to my wife, who took care of the kids alone that week.

Let me share with you a few thoughts about the conference.

First, I'm going to write about the 3 talks that I liked the most during the 3 days, then I'm going to share 3 interesting ideas I heard about and then I'm going to share some personal impressions about the conference.

The recordings will be on YouTube when they are available.

My favourite talks

Last year, I wrote that I was pondering what makes a good talk for me and that I enjoyed more the talks that covered beginner topics in depth. I still feel the same way, but I'm not 100% sure if my selection represents that. There are a couple of presenters who due to their outstanding knowledge and wonderful presenter skills can captivate any audience full of C++ enthusiasts.

The talks are ordered by their scheduled time.

Understanding The constexpr 2-Step: From Compile Time To Run Time by Jason Turner

The conference had a very strong start. Right after the keynote by Dave Abrahams, four incredible speakers were on stage at the same time in the four different tracks. Jason Turner, Walter E Brown, Nico Josuttis, Mateusz Pusz...



If a conference could have these people spread throughout the whole program, it would be a strong conference. *C++ On Sea* proposed such a strong line-up that these people could be scheduled at the same time. It didn't make my decision easier, so I chose based on the topic, and I wanted to grow my knowledge on `constexpr` so I stayed in the `main()` room.

Let's talk about the talk [CPPoS-2].

C++20 brought us `constexpr std::vector` and `std::string`. Yet, the simple piece of code below doesn't compile.

```
#include <vector>
int main()
{
    constexpr std::vector<int> data{1,2,3,4,5};
}
```

`constexpr` variable data must be initialized by a constant expression...

In Jason's talk, we learned about why that's the case and how we can use those `constexpr` constructs. The key to answering that question is in understanding that to instantiate objects that (might) allocate dynamic memory, *memory allocated at compile time must be freed at compile time*.

I don't want to go through all of Jason's reasoning and examples: I simply share the two steps that he referred to in the title. For the rest, watch the recording once it's available:

1. Do all the dynamic storage stuff you want to at compile-time.
2. Copy the dynamic storage stuff to static stuff at compile-time (and make sure you free the dynamic thing at compile-time).

While I don't want to give away all of his talk, I do want to share a compiler bug that Jason shared with us. If you use `llvm` and you want to write a `constexpr` function, be aware that you cannot have static local variables: the compiler removes them. Here is a minimal example taken from the corresponding Github issue [Github]:

```
constexpr const int* a() {
    static constexpr int b = 3;
    return &b;
}
int c() { return *a(); }
/*
GCC assembly:
c():
    mov     eax, 3
    ret

Clang assembly:
c():
    xor     eax, eax
    ret
*/
```

Now you are aware. Let's see the other talks.

Core and other guidelines. The good, the bad, the... questionable? by Arne Mertz

Once again, it was a tough call to decide whether I should attend Peter Muldoon's talk on dependency injection [CPPoS-3] or Arne's on guidelines. As my team had recently had some discussions about some of the core guidelines, I decided to attend Arne's talk [CPPoS-4].

Arne has worked on a dozen different projects as a consultant during the last 9 years. Based on what he's seen, he shared how guidelines have been misused. In the standard, or in the core guidelines – at least in the titles –

Sandor Dargo is a passionate software craftsman focusing on reducing maintenance costs by applying and enforcing clean code standards. He loves knowledge sharing, both oral and written. When not reading or writing, he spends most of his time with his two children and wife in the kitchen or travelling. Feel free to contact him at sandor.dargo@gmail.com



Arne Mertz at C++ On Sea

every word is important. If you skip one, or half a sentence, the meaning may be something completely different.

Arne brought some examples of guidelines set in different companies that were most often based on the core guidelines or some other companies' (in)famous policies, but they were misinterpreted. For example, in one place, they had this guideline:

Define or delete all copy, move, and destructor functions.

Does it remind you of the rule of 5? It should. But they forgot to add that you should only define all these special member functions if you have to define at least one of them...

Another company declared that you should not use exceptions. There was no good rationale behind it, except that Google famously banned them. Google also said that if they started over, they would probably do things differently.

When it comes to adopting guidelines, it's important to put things into context. A guideline that makes sense in a certain context might even be harmful in a different environment.

There is also the question of whether rules are just guidelines. The secret is in the name. Guidelines are guidelines. Sometimes you might go against them. At the same time, if you do so, it's better to document why. Otherwise, you'll make people waste too much time figuring that out. Worse, they might even undo a good decision.

Also, keep your style guide short. At the same time, automate as many checks as possible with the right tooling. A subject that I'll mention in later sections of this trip report.

There is no Silver Bullet by Klaus Iglberger

Klaus had the task of keeping the audience engaged at the end of Friday afternoon with his closing keynote. I think I'm not alone in saying that he fulfilled his job with his talk 'There is no silver bullet' [CPPoS-5].

Most of us can agree that 13 years after the release of C++11, using the term 'modern C++' is probably not the best idea. Ivan Čukić came up with the term 'progressive C++', which Klaus likes too [Dargo20]. We'll see if that term sticks.

Klaus used an anonymous comment on one of his earlier talks to give a structure for this presentation. According to the commenter:

object-oriented programming and especially its theory is overestimated. ... C++ always had templates, and now also has `std::variant`, which makes most of the use of inheritance unnecessary.

Heck, even Jon Kalb said at *CppCon 2019* [Kalb19] that:

object-oriented programming is not what the cool kids are doing in C++. They are doing things at compile time, functional programming, ... Object-oriented programming, this is so 90s.

So, Klaus went on with the good old shapes example and implemented it in various ways including the old school OO way, with variants and with templates, and compared them.

He indeed managed to get a nice speedup, but it's not all black and white. While it's easy to add new operations to the variant solution, it's relatively difficult to add new types. With the OO solution, it's the other way around. Moreover, due to the reversed dependencies, the `std::variant` solution is an architectural disaster.

It doesn't mean that OO doesn't have a place and that it cannot be used in certain situations. It simply means that different solutions have different pros and cons.

These solutions can even be combined into a value-based object-oriented solution, which is still not a silver bullet but can be well used in high-level architecture.

Learn about it by watching the recording once it's released!

My favourite ideas

Now let me share three interesting ideas from four different talks.

'We all write bad code sometimes' – Jan Baart

Jan Baart gave a very useful talk about code modernization and unit testing [CPPoS-6]. Talks like this are useful for several kinds of audiences. For junior developers, they gave actionable tasks, and for seniors, a reminder. A reminder of what is important and what message we have to share and distribute.

While modernizing and adding unit tests to legacy code is important, the most important message of Jan was about humility:

No blaming, we all write bad code sometimes.

We don't have to condemn others because a piece of code is bad. Probably they did their best writing it. In the past, I wrote much worse code than today and hopefully in the future, I'll write better than nowadays. That's probably true for you as well. Besides, we can simply have bad days. Don't blame others for bad code. Help them grow.

(Let's leave aside the question of someone not even trying to do a good job. That's a different problem to be dealt by management.)

'You should write tests' – Robert Leahy

In my opinion, Robert Leahy was among the best presenters at *C++ On Sea 2024* [CPPoS-7]. His points were clear and he was exceptionally energetic on stage.

His message was clear. We should write tests. Even for components or bugs that seem to be too small, simple or trivial to be tested. He brought several examples from his code to support his points. Indeed a one-liner function calling `std::min` can have a bug in it, so it's worth adding tests.

Even though I would argue with Robert whether some of his tests are actually unit tests, there is nothing to argue with his main message. Write tests, not only because of delivered wisdom, but because it levers up your output, and improves your design and code.

'Write your own clang-tidy checks' – Mike Crow and Svyatoslav Feldsherov

There were two talks about writing your own `clang-tidy` checks or even refactoring actions. One by Mike Crow [CPPoS-8] and another by Svyatoslav Feldsherov [CPPoS-9]. I liked the message of both of these talks.

`clang-tidy` and the Abstract Syntax Tree (AST) look sometimes a bit too 'abstract' to most of us. And if we look at some sample code we'd have to write, it doesn't make things any better. These talks bring these tasks a little bit closer to everyone and they clearly tell us that even though they are not the simplest things, they're not rocket science either.

Until the recordings are available, it's worth looking into AST Matcher Reference [clang].

Personal impressions

Finally, let me talk about some more personal feelings about the conference.

Starts to feel like home

The first two occasions I attended *C++ On Sea*, I did so online, but this was already the third time I had attended in person. It is starting to feel a bit like home. I don't need a map for anything: I know where to find things at the conference and in the town. I have my favourite places.

More importantly, with more and more people, we are greeting each other with a big smile. Organizers, speakers and attendees included. This feels right and makes me appreciate the social aspects of a conference even more. I meet some of these people more than I do my colleagues.

It also made me realize that, so far, *C++ On Sea* has been the only C++ conference where I have given a talk in person. This might change in a few months, though.

New ideas keep coming

It's nice to see that organizers pay close attention to detail. If something doesn't work as expected one day, they improve it for the next day. And the venue staff members are partners in this, too.

There were two novelties this year that I want to mention.

Wednesday evening, there was a movie night hosted by Walter E Brown with some short or longer clips about the history of computer science. The organizers also provided pizza for everyone who wanted to stay around so that we couldn't claim that we needed to find dinner somewhere else. Sadly, I couldn't stay until the end as I had my sessions the next morning. Nevertheless, I liked what I saw.

There was a fun buzzword bingo to win some C++ books. The idea was that we had C++ buzzwords on a paper such as `const_cast`, `ADL`, `void`, etc. just to name a few. At each session, we could tick two of them that we heard and the first few people who got 5 words in a single row or column could get a book. Although I didn't intend to take the book as I already have it, I liked the idea and played along.

My talks

On the second morning, I had two consecutive slots to deliver a half-day workshop about how to keep your binaries small. During the pandemic, I had delivered a half-day workshop online... but obviously, that was a vastly different experience.

Last year, I left my clicker at home. This year I had it, but for some strange reason, it stopped working properly. So in the end, I was very static on

stage, as I had to stay close to my laptop and its space button. Apart from that, it went quite well.

The first part of the first session was about binary formats. I was afraid that it would be too boring for most people, but as it turned out many appreciated it and the great majority of people turned up for the second session as well.

Overall, I received good feedback from some attendees and also some follow-up questions, such as how dynamic linkage affects binary sizes.

I had a second commitment as well. I am someone who feels obligated to give a lightning talk, if that is possible. I feel obligated to go on stage and practise whenever there is an opportunity. So I did, talking about whether engineering teams really resemble sports teams [Dargo24]. I ran a few seconds over time, but I finished what I wanted to share.

Conclusion

C++ On Sea was as great an experience in 2024 as during the previous years [Dargo]. Three days packed with great presentations about various topics, including performance, tooling, design and best practices.

The best we can do is to spread the word so that maybe even more people join next year and also to just share what we learned.

I hope to be back in Folkestone in 2025. ■

References

- [clang] AST Matcher Reference: <https://clang.llvm.org/docs/LibASTMatchersReference.html>
- [CPPoS-1] *C++ on Sea* website: <https://cpponea.uk/>
- [CPPoS-2] Abstract: Jason Turner: <https://cpponea.uk/2024/session/understanding-the-constexpr-2-step-from-compile-time-to-run-time>
- [CPPoS-3] Abstract: Peter Muldoon: <https://cpponea.uk/2024/session/dependency-injection-in-cpp-a-practical-guide>
- [CPPoS-4] Abstract: Arne Metz: <https://cpponea.uk/2024/session/core-and-other-guidelines-the-good-the-bad-the-questionable>
- [CPPoS-5] Abstract: Klaus Iglberger: <https://cpponea.uk/2024/session/there-is-no-silver-bullet>
- [CPPoS-6] Abstract: Jan Baart: <https://cpponea.uk/2024/session/who-needs-unit-tests-anyway-modernizing-legacy-code-with-0pc-code-coverage>
- [CPPoS-7] Abstract: Robert Leahy: <https://cpponea.uk/2024/session/fantastic-bugs-and-how-to-test-them>
- [CPPoS-8] Abstract: Mike Crowe: <https://cpponea.uk/2024/session/building-on-clang-tidy-to-move-from-printf-style-to-stdprint-style-logging-and-beyond>
- [CPPoS-9] Abstract: Svyatoslav Feldshero: <https://cpponea.uk/2024/session/pets-cattle-and-automatic-operations-with-code>
- [Dargo] *C++ on Sea* trip reports, listed at <https://www.sandordargo.com/tags/cpponea/>
- [Dargo20] 'Functional Programming in C++ by Ivan Cukic' posted on 27 May 2020 at <https://www.sandordargo.com/blog/2020/05/28/functional-programming-in-cpp>
- [Dargo24] 'Do engineering teams really resemble sports teams?', posted on 21 May 2024 at <https://www.sandordargo.com/blog/2024/05/22/are-we-a-sports-team>
- [Github] 'Permitting static constexpr variables in consteval functions', <https://github.com/llvm/llvm-project/issues/82994>
- [Kalb19] 'Why don't cool kids like OOP?', a Lightning Talk at *Meeting C++ 2019*, available at <https://www.youtube.com/watch?v=x1xkb7Cfo6w>



Me at *C++ On Sea*

This article was previously published on Sandor Dargo's Blog on 10 July 2024, available at <https://www.sandordargo.com/blog/2024/07/10/cpponea2024-trip-report>

Afterwood

Have things become unnecessarily complicated?
Chris Oldwood is testing simplicity.

Einstein famously told us to keep things as simple as possible, and no simpler. Hence, you'd think that being presented with a small snippet of simple code would generate a flurry of correct answers along with lots of metaphorical shrugging of shoulders and cries of "meh?" as people wonder what all the fuss is about. Jason Gorman, someone who specialises in training and coaching software teams, recently posted the following question on his various social media accounts [Gorman-X] [Gorman-M]:

Quick experiment: what's the expected result that's been blanked out in this test code?

```
[Test]
public void FundsTransferDebitsPayer ()
{
    // Arrange
    Account payer = new Account ();
    payer.Credit(1000.0);
    Account payee = new Account ();

    // Act
    payer.Transfer(250.0, payee);

    //Assert
    Assert.That(payer.Balance, Is.EqualTo(_____));
}
```

He doesn't say what programming language this is written in but it's fair to say it's either Java or C# – a pair that are so similar these days they are easier to mix-up than Pluto and Goofy, or The Munsters and The Addams family. Either way, both these languages generally hold very few surprises, or dark corners that could lead to the pit of undefined behaviour. For those playing along at home, the answer really is just 750.0.

My own response was a somewhat guarded "750, right?", followed by a flippant suggestion that if this was an interview there would almost certainly be some gotcha that I hadn't accounted for. There wasn't, this time, but nobody ever posts a snippet of code on social media that does exactly what it says on the tin, so is it any wonder that nearly every reply was caveated in some way? We programmers can be such a jaded bunch at times, our bodies metaphorically covered from head-to-toe in scar tissue.

Naturally I blame all those ridiculous interviews we go through where the apparent goal is not to hire someone capable of doing the job but to allow the interviewer to show the candidates how clever they are. If this code snippet was presented in a code review or generated during a pair / ensemble session I would hardly think twice about it. (That's not entirely true because of the apparent ill-advised use of floating-point numbers for monetary values instead of using decimal in C#, BigDecimal in Java, scaled integer, etc. but that wasn't the point of the question.)

Many of the teams I've worked in over the years have generally been of the 'you build it, you run it' stance and so your current teammates provide a useful barometer for what passes as 'simple enough'. In a few cases, though, I've worked in The Enterprise™ where they have a different approach to maintenance, one where they hire a bunch of experienced developers to create a new service or application and then, when the first release is done, they hand it over to 'the support team' for any future

bug fixes or enhancements. What surprised me was that this team mostly spends their time doing system administration but only gets to don their software developer hat occasionally.

This difference in maintenance approach made me consider the readers of my code more deeply than usual and this popular quote from *The Elements of Programming Style* [Kernighan78] took on a new perspective:

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Consequently, patterns and idioms which, up to that point I assumed to be perfectly acceptable, I now felt less comfortable using and started questioning whether I was carrying over anachronisms instead of adopting more modern idioms that would make the code more accessible to this different audience. For example, as an industry we now discourage people from writing manual loops, and so maybe a retry loop in C# like:

```
var attempt = 5;
while(attempt-- > 0) { ... }
```

is now becoming trickier to reason about when it comes to mentally checking the maximum number of times around the loop than:

```
foreach(var attempt in
    Enumerable.Range(1, 5)) { ... }
```

While the former technique might have been inherited from C by several popular languages, that doesn't guarantee universal comprehension. It might be marginally less typing but as we embrace more functional idioms the latter style of code is becoming the new norm.

On the other hand, am I falling foul of Einstein's advice and heading into oversimplification territory and as a consequence verging on patronisation? Can code become too simple?

What I liked about the replies to Jason's experiment was seeing people answer their own questions about the code without realising it. As Alex Horne often says to contestants in this situation on Taskmaster, "all the information is on the task", though in Jason's example it's in the names of the test, class, methods, and variables. Any other assumptions, like the default balance being 0.0, feel like a no-brainer too.

At some point we have to put our spade down and trust that the reader will engage their own brain. The trouble seems to be that the reader doesn't always trust us to hold up our end of the bargain. ■

References

[Gorman-M] Jason Gorman's Mastodon account:

<https://mastodon.cloud/@jasongorman/111601548580386385>

[Gorman-X] Jason Gorman's X account:

<https://x.com/jasongorman/status/1736733070726721935>

[Kernighan78] Brian Kernighan and P. J. Plauger (1978)

The Elements of Programming Style 2nd Edition,
McGraw-Hill Education,
ISBN-13 978-0070342071

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](https://twitter.com/chrisoldwood)



accu



Monthly journals, printed and online
Local groups run by ACCU members
Discounted rate for the ACCU Conference
Email discussion lists

accu.org

To connect with
like-minded people
visit accu.org



accu