overload 174

APRIL 2023 £4.50

In Search of a Better Concurrency Model

Lucian Radu Teodorescu presents current plans for concurrency in the VAL programming language

Drawing a Line Under Aligned Memory

Paul Floyd reminds us about various aligned memory functions

C++20 Concepts: Testing Constrained Functions

Andreas Fertig gives a worked example of testing constraints on functions or classes and other template constructs

Meta Verse

Teedy Deigh turns on, jacks in, and checks out the immersive experience

A magazine of ACCU ISSN: 1354-3172

accu

professionalism in programming

Monthly journals, available printed and online

Discounted rate for the ACCU Conference

Email discussion lists

Technical book reviews

Local groups run by ACCU members

ACCU is a not-for-profit organisation.

Become a member and support your programming community.

www.ACCU.org

April 2023

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

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: Old unused iron railway bridge, Old Coach Road, North Island, New Zealand. By imagoDens, iStock Photo.

Copy deadlines

All articles intended for publication in *Overload* 175 should be submitted by 1st May 2023 and those for *Overload* 176 by 1st July 2023.

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

4 **Drawing a Line Under Aligned Memory**
Paul Floyd reminds us about various aligned allocation functions.

7 **C++20 Concepts: Testing Constrained Functions**
Andreas Fertig gives a worked example of putting constraints on functions or classes and other template constructs.

10 **In Search of a Better Concurrency Model**
Lucian Radu Teodorescu presents current plans for concurrency in the VAL programming language.

16 **Meta Verse**
Teedy Deigh turns on, jacks in, and checks out the immersive experience.

The ACCU

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

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

The Rise and Fall of Almost Everything

Some things go up and up, while others go up and down. Frances Buontempo considers whether the distinction matters and how to spot the difference.

The news in the UK moved on from claiming everything is under pressure, as I mention in the last *Overload* [Buontempo23a], to telling us we're at breaking point. Then we ran out of tomatoes. This is a grand distraction from a variety of important issues, and furthermore has sidetracked me completely from writing an editorial, which is a shame. We will produce extra copies of this edition of *Overload* to hand out at the ACCU conference in Bristol this year, so now would have been a good time to write an editorial. Hello to attendees reading this. For those who have not read *Overload* before, I have a track record of failing to write editorials, so this situation is entirely expected by our regular readers. It's therefore very easy to predict whether I will write an editorial or not. Most situations in life are more difficult to predict, though. Things come and go. Prices or even empires rise and fall. Past performance is no indicator of future results. Of course, this makes a mockery of any attempts at statistics or data science. Both disciplines tend to rely on previous results and values to form a model based on patterns. If we assume the future will be unlike the past, we are in effect saying there is no point in making such models.

The philosopher David Hume grappled with this issue. He suggested that trying to predict the future relies on moving from specific observations to general principles, so is using induction or inductive inference. This hinges on an assumption of what is sometimes referred to as the uniformity principle: "The future will be like the past" [Henderson22]. He [Hume48] claims:

> That the sun will not rise tomorrow is no less intelligible a proposition, and implies no more contradiction, than the affirmation, that it will rise.

We assume unchanging laws govern the universe, and extrapolate from there. In fact, someone, maybe Einstein, once said:

> Insanity is doing the same thing over and over again and expecting different results.

Whether or not the quote is ascribed correctly, we might expect calling the same function to give the same results each time, with some caveats. We might describe such a function as idempotent. We can call it twice and expect the same results. Not all functions behave like this, C's `rand` being the first that springs to mind. Running a program using `rand` twice and getting exactly the same results is not impossible, and confuses coders used to some other languages. As soon as you seed the random number generator, though, you will get a different sequence of numbers. Some languages seed the random number generator for you, but some don't, and this can be a source of confusion.

That we can generate sequences of numbers that appear to be random is quite an achievement. John von Neumann once said:

> Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.

The majority of the 'random' number generators we use are deterministic. Whether we can generate a sequence of truly random numbers is another matter, and I am starting to suspect I don't even know what random actually means. We do, however, have tests describing properties we expect from 'random' numbers, including the *Wald–Wolfowitz runs test*:

> Generate a long sequence of random floats on (0,1). Count ascending and descending runs. The counts should follow a certain distribution. [Wikipedia-1]

This test is more subtle than checking if the number of increasing and decreasing steps are the same, but rather checks the numbers appear to be independent and identically distributed. We know what we want: numbers going up and down, even if we can't define random precisely.

Sometimes people say, "What goes up must come down." This isn't always true. For example, I could throw my keyboard out of the window, and it might go down. If, instead, I launched it out of the window with a suitable rocket, it could either reach escape velocity or orbit the planet. Some things do go up and down though. When values follow such ups and downs, they can be described as seasonal. Many trading strategies fall into either a trend following approach or a seasonal approach. The former tends to show a long term increase or decrease, while the latter cycles over a fixed period. For example, power usage might go up in the winter when the weather is colder and reduce in the warmer summer months. Trying to spot when prices have changed from trend following to seasonal is difficult, and usually relies on time series analysis.

Sometimes, you might think you have found a pattern. If I say 0, 1, 2, 3, 4, 5, you can guess what might happen next. If I then tell you these numbers were generated using the increment operator starting with `unsigned x= 0;` you know the numbers will increase, to a point, then return to zero. What goes up might come down unexpectedly, which you may only realise if you have full details on the context. Some things appear to go up and down at the same time. Escher's impossible staircase in his lithograph *Relativity* immediately springs to mind. Escher had been inspired by the Penrose stairs [Wikipedia-2], which appear to be going up and down simultaneously. Such a staircase is impossible in Euclidean geometry, but not infeasible in some pure mathematical models. As your perspective shifts when you view the picture so too, as your world view or framework changes, up may become down or vice versa.

If you've ever tried learning a new language, or even keeping up to date with new versions of the same language, you will be familiar with the

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.

rise and fall of learning. You might get to a point of considering yourself an expert, only to be confronted with everything moving under your feet, and falling back to feeling like a complete beginner again. You might feel like you are failing with simple things initially. I mused on this a while ago [Buontempo15], recalling the words of Batman's father in *Batman Begins* when the young Bruce Wayne falls down deep into the bat cave:

> And why do we fall, Bruce? So we can learn to pick ourselves up.

I am still excited about trying out new programming languages and technologies, but do sometimes experience a twinge of worry when reading the documents or trying out something for the first time.

I've been writing a C++ book to try to help people catch up if they got left behind with the various new features introduced over the last few standards [Buontempo23b]. There are many books out there which go into full detail, but I wanted to try out some small, self-contained projects showcasing a few of the newer features, partly for self-indulgent reasons and partly to see how well I can explain myself. I moved from feeling excited when the publisher accepted my proposal, to feeling overwhelmed and like a fraud. Imposter syndrome frequently rears its head. You can avoid feeling like this if you never try anything new, but where's the fun in that? Trying to complete any project, be it an editorial or a book, tends to hit a shaky patch in the middle. You might start full of determination and find some extra stamina to make the finish line near the end, but the middle is always difficult. Several of my friends dropped out of university in the second year of a three year course. I keep trying to row 2km on a rowing machine in the gym and almost always grind to a near halt a bit over 1km. If I pace myself a bit and keep going, I get there, but it is hard work. One day I might manage it in less than 10 minutes. We shall see. Maybe latent heat is a good analogy for this sticky middle? If you heat a substance, its temperature increases for a while. It hits a point where the temperature ceases to increase, while the internal state changes, moving from a solid to a liquid or a liquid to a gas. I haven't managed to write any book for the last few days. I shall tell myself it is latent heat. You can't see the page count increase, but something is shaping up in my head. Once I've sorted out where I'm going next, the page count will start increasing again. Some things can be an up-hill struggle for a bit. It's OK to pause and get your breath back.

Moving from physics to mathematics, we have points of inflection. If you draw a plot of $y=x^3$, to the left of the origin the values are negative but increase, getting closer to zero. To the right, the numbers are positive and increase. At the origin, y is zero. This point is neither a maximum nor a minimum, but described as a point of inflection. Minimums, maximums and points of inflection each have a derivative of zero, and are collectively known as stationary points. The point of inflection might look slightly like an S, with the curvature changing from upwards to downwards or vice versa. They are notoriously sneaked into maths questions, because it is very easy to find a derivative of zero, and forget to check it is a maximum or minimum and not an inflection point. Sometimes things are more complicated than they first appear. The same happens when we write software. We might think we found a way to make code quicker, only to find it plateaus at some point. We might download a device driver, to be told we had 17 minutes remaining, 16, 15…, 53, then 2, then 19. And so on. Many things seem to trend in one direction, but then things change. The trick is spotting when you missed some information.

Hooke's law tells us that the force needed to compress or extend a spring or other elastic object, is proportional, or scales linearly, with respect to the distance stretched or compressed. Until it isn't. Wikipedia [Wikipedia-3] says:

> An elastic body or material for which this equation can be assumed is said to be linear-elastic or Hookean.

If the equation "can be assumed" for some things, it cannot be assumed for others. And if a heavy enough weight is put on a spring it will extend and finally break, no longer being Hookean and in fact no long being very springy. You can draw a graph of stress (force) against strain (deformation) and see what happens. It might be linear to a point, known as the elastic limit. Then things might change.

Almost nothing is really linear, though linear models lead to simpler maths, so we often use them as an approximation. The trick is not to believe our own lies. If a simple model does not work, this setback can then be disheartening. We could recall the words in Monty Python's *Life of Brian*:

> Let us not be down-hearted. One total catastrophe like this is just the beginning!

However, it is more sensible to remind ourselves that learning and growing is often non-linear. If my attempt on the rowing machine slips by a few seconds one time, I will not give up. I still got some practice in, so I should be pleased with myself. Don't beat yourself up if something doesn't go as planned. Don't give in to despair, like Reginald Perrin [IMDb], who

> Disillusioned after a long career at Sunshine Desserts, Perrin goes through a mid-life crisis and fakes his own death.

Fear not, this old British comedy is very silly, and things work out for him in the end.

Though the weather has been cold recently, I can see signs of spring in our garden. We also still have several leaves rotting on the lawn from the autumn, which some cultures call the fall. Maybe we should call spring the rise? Things change, sometimes for the better, sometimes not. Whatever is going on in your life, hold on to the positives. Look out of the window and see the flowers starting to bloom, and then settle back and read *Overload*.

References

[Buontempo15] Frances Buontempo 'Failure is an Option', *Overload* 129, Oct 2015, https://accu.org/journals/overload/23/129/buontempo_2156/

[Buontempo23a] Frances Buontempo 'Under Pressure', *Overload* 173 Feb 2023 https://accu.org/journals/overload/31/173/buontempo/

[Buontempo23b] Frances Buontempo *C++ Bookcamp* (under development) https://www.manning.com/books/c-plus-plus-bookcamp

[Henderson22] Leah Henderson 'The Problem of Induction' in *The Standford Encylopedia of Philosophy* (Winter 2022 Edition), available at: https://plato.stanford.edu/archives/win2022/entries/induction-problem/

[Hume48] David Hume (1748) *An enquiry concerning human understanding*.

[IMDb] 'The Fall and Rise of Reginald Perrin' *https://www.imdb.com/title/tt0073990/*

[Wikipedia-1] Diehard tests: https://en.wikipedia.org/wiki/Diehard_tests

[Wikipedia-2] Penrose stairs: https://en.wikipedia.org/wiki/Penrose_stairs

[Wikipedia-3] Hooke's law: https://en.wikipedia.org/wiki/Hooke%27s_law

Drawing a Line Under Aligned Memory

When we allocate memory we often forget about alignment. Paul Floyd reminds about various aligned allocation functions.

Recently I've been doing some work with the various Unix-like systems implementations of C functions to allocate aligned memory. These are **`memalign`**, **`aligned_alloc`** and **`posix_memalign`**. Typically, you would use these functions to get memory that is aligned with cache lines or virtual memory pages. As an example of this, imagine a networking application that needs to allocate **`struct msghdr`** and to have the fastest memory access possible. This struct has a size of 56 bytes. If you use **`malloc`** to allocate your memory you are likely to get back a pointer that, depending on the system, is 8- or 16-byte aligned. That means that there is a fair chance that the memory will straddle a 64-byte alignment boundary. That is bad because that is what cache lines map to, meaning that accessing fields of the structure will hit two cache lines. This increases the risk of cache misses, resulting in lower performance.

I'm not going to detail the performance benefits (or drawbacks) of using these functions. Instead in this article I'll be discussing some of the issues that I saw. The implementations that I've looked at are Linux glibc [GNU libc], Linux musl [musl], FreeBSD jemalloc [FreeBSD], macOS [XNU] and Illumos [illumos]. There are other malloc libraries (Illumos umem, tcmalloc, rpmalloc and snmalloc for instance) but I haven't looked at them. Also, (almost) no Windows as I don't use it enough to make fair comment.

History

These functions go back a long way. **`memalign`** goes back to SunOS 4.1.3 (Aug 1992 according to Wikipedia). Despite its age it is not a 'standard' function. The non-standard-ness shows, as we'll see shortly. That means it doesn't figure in either the C standard or the POSIX standard. It doesn't exist on macOS. glibc and musl both have implementations. Finally, FreeBSD gained a version late in the game in 2020 to add glibc compatibility.

`posix_memalign`, as the name implies, is a bona fide part of the POSIX spec. IEEE Std 1003.1d-1999 Additional Realtime Extensions to be precise. All the systems and libraries that I looked at implement **`posix_memalign`**.

`aligned_alloc` was standardized in C11. Again, this was implemented on all the systems that I looked at.

What they claim to do

Here is what the Linux man page says:

> The function posix_memalign() allocates size bytes and places the ad dress of the allocated memory in *memptr. The address of the allocated memory will be a multiple of alignment, which must be a power of two and a multiple of sizeof(void *). This address can later be successfully passed to free(3). If size is 0, then the value placed in *memptr is either NULL or a unique pointer value.
>
> The obsolete function memalign() allocates size bytes and returns a pointer to the allocated memory. The memory address will be a multiple of alignment, which must be a power of two.
>
> The function aligned_alloc() is the same as memalign(), except for the added restriction that size should be a multiple of alignment.

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

That all sounds very reasonable. The POSIX standard has similar wording for **`posix_memalign`**. The spec can be accessed from The Open Group [opengroup], but you need to create an account and log in to access it.

Sadly, C11 does not have very much to say about **`aligned_alloc`**:

> The value of alignment shall be a valid alignment supported by the implementation and the value of size shall be an integral multiple of alignment.

Great, so the alignment can be anything, but the size needs to be a multiple of the same anything. The final draft of C11 can be found here [C11 final].

I can't comment on **`memalign`** since it isn't standardized.

Musl, and more specifically Alpine Linux, doesn't change the man page.

The FreeBSD description for **`posix_memalign`** is very similar. For **`aligned_alloc`** it says:

> The aligned_alloc() function allocates size bytes of memory such that the allocation's base address is a multiple of alignment. The requested alignment must be a power of 2. Behavior is undefined if size is not an integral multiple of alignment.

There is no manpage for **`memalign`** on FreeBSD.

Illumos has the following to say of **`memalign`**:

> The memalign() function allocates size bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of alignment. The value of alignment must be a power of two and must be greater than or equal to the size of a word.

The Illumos wording for **`posix_memalign`** is again similar to the others, but with one exception. This time the behaviour when the **`size`** is zero is specified:

> If the size of the space requested is 0, the value returned in memptr will be a null pointer.

The macOS manpages are quite similar to FreeBSD.

To summarize so far, **`posix_memalign`** is fairly well defined. **`memalign`** is a bit hazy for a size of zero and I'm not sure what Solaris was getting on about saying that the return address will be an even multiple of the alignment. All of the descriptions of **`aligned_alloc`** say that the alignment must be a power of two and the size an integral multiple of the alignment.

So far, no platform has done anything about the "the value of size shall be an integral multiple of alignment" part of the C11 standard

What they actually do?

So how do the implementations match up to the specs? I'm not going to go into internal details – all the functions may allocate more than asked or be aligned to a higher value.

Thus far I've been describing the functions in chronological order. This time I'm going to let **`posix_memalign`** jump the queue. All the implementations behave as specified. Illumos does indeed not allocate if the size is zero. The other implementations allocate some unspecified amount.

The man page for Linux glibc **`memalign`** claimed that the alignment must be a power of two. In fact, any value of alignment will be accepted and silently bumped up to the next power of two.

Two of the **`memalign`** implementations were buggy. FreeBSD would crash if the alignment was zero – I've submitted a patch for that which has been merged. Illumos only restricts the **`memalign`** alignment to being a multiple of four. That can result in some peculiar values for the alignment. I've opened a bug tracker item for that. There was nothing wrong with musl that I could see.

On to the last of the trio, **`aligned_alloc`**. The Linux man page claims that this is the same as **`memalign`** except that the size should be a multiple of the alignment. For glibc, doing that would be an amazing technical feat. The two functions are in fact the same. To be more precise they are both weak aliases of **`__libc_memalign`**. So, there is no extra constraint on the size.

Other platforms also use a lot of code sharing. FreeBSD **`memalign`** calls **`aligned_alloc`** but with the size rounded up to a multiple of alignment. If anything, I would have expected the opposite, but anything goes when functions are non-standard, or implementation defined. Musl **`memalign`** just calls **`aligned_allloc`**. And with a nice bit of symmetry, Illumos **`aligned_alloc`** just calls **`memalign`**.

Just when I thought I'd covered everything, I discovered that if you use a huge value of alignment with musl **`aligned_alloc`** then it will crash with a segfault. The crash is in version 1.2.2 and it has apparently been fixed in 1.2.3.

So far, no platform has done anything about the "the value of size shall be an integral multiple of alignment" part of the C11 standard. macOS is the remaining platform and it DOES do something about it. If the size isn't

What is a 'weak alias'?

It is a mechanism that allows one or more symbols to refer to the same object or function. I shall now digress into the world of the link editor and the link loader. I expect that everyone reading this is familiar with compiling and linking libraries and executables. You compile some source files into object files and then link them. There isn't always a 1:1 relationship between names in your source and symbols in object files. There are several ways in which this can happen. One way that this can be done is to explicitly request a 'weak alias'. These aliases can refer to any other symbol, and unlike regular symbols it is not an error if weak aliases do not get resolved. That makes them ideal to use for functions such as the **`malloc`** family that are specified to be replaceable.

Consider this small program:

```
#include <iostream>
extern "C" void hello()
{
  std::cout << "Hello from " << __func__
    << " address " << std::hex << (size_t)hello
    << '\n';
}
extern "C" void hello_alias() __attribute__
  ((weak, alias ("hello")));
int main()
{
  hello();
  hello_alias();
}
```

As you can see, **`main()`** calls two functions, but only one is defined!

If I compile and run this, I get

```
paulf> ./weak_alias
Hello from hello address 202740
Hello from hello address 202740
```

As you see, both calls print the same function address, confirming that the weak alias calls the original strong function. The **`nm`** tool can show this in the binary:

```
paulf> nm weak_alias | grep hello
0000000000202740 T hello
0000000000202740 W hello_alias
```

`T` means a global function and **`W`** a weak alias. Getting back to the GNU libc case of weak aliases, **`nm`** can again be used to show them. First of all, **`aligned_alloc`**

```
paulf> nm /lib64/libc.so.6 | grep aligned_alloc
000000000009a6f0 W aligned_alloc
```

Then all symbols with the same address:

```
paulf> nm /lib64/libc.so.6 | grep 000000000009a6f0
000000000009a6f0 W aligned_alloc
000000000009a6f0 t __GI___libc_memalign
000000000009a6f0 T __libc_memalign
000000000009a6f0 W memalign
000000000009a6f0 t __memalign
```

Here, **`__libc_memalign`** is the real, private, implementation and **`aligned_alloc`** and **`memalign`** are the public aliases for **`__libc_memalign`**.

an integral multiple of the alignment, then it will return NULL and set errno to EINVAL.

One thing that is generally not documented is that most of the functions will fail if the alignment is huge (over half the memory space). In that case they will return **`NULL`** and set **`errno`** to **`EINVAL`**.

Windows almost got away without a mention. Whilst Windows doesn't have any of the Unix aligned allocation functions (not even C11 **`aligned_alloc`**), it does have its own variation. It's called **`_aligned_malloc`** [Microsoft].

Other than having an underscore and an extra 'm', Microsoft also has the order of the alignment and the size arguments reversed. That seems to me a source of confusion and potential bugs. I'm not sure if **`_aligned_alloc`** predates **`memalign`**, I see references to it going as far back as VC++ 6.0 (1998). That means that by the time C11 came around there were already functions with different argument ordering.

Advice

Whilst I must say that I was quite underwhelmed by the quality of what I saw, I don't think that in practice these are big issues. I do recommend that you avoid using an alignment that is zero or a non-power of two. Unfortunately, Hyram's law [hyrum] says that there is probably code out there that is taking advantage of Linux glibc working out the next power of two for the alignment. For portability, **`posix_memalign`** and **`aligned_alloc`** have the edge, and of the two, **`aligned_alloc`** is easier to adapt to its Windows counterpart, **`_aligned_malloc`**. However, you still need to take care that the size is an integral multiple of the alignment if you also port to macOS. ■

References

[C11 final] International Standard: https://open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf

[FreeBSD] Source for freebsd: https://github.com/freebsd/freebsd-src

[GNU libc] Source for glibc v2.37: https://elixir.bootlin.com/glibc/glibc-2.37/source

[hyrum] Hyrum's Law: https://www.hyrumslaw.com/

[illumos] Illumos is the continuation of OpenSolaris: https://github.com/illumos/illumos-gate

[Microsoft] _aligned_malloc: https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/aligned-malloc?view=msvc-170

[musl] Source for musl: https://elixir.bootlin.com/musl/v1.2.3/source

[opengroup] Open Group Library: https://publications.opengroup.org

[XNU] Source browser: https://opensource.apple.com/source/xnu/ (there are also GitHub mirrors)

Idalia is a freelance artist operating at the intersection of art and geek, using a myriad of techniques and styles to produce works that both delight and entertain.

The best way to reach her is via idalia.ku@hotmail.com

C++20 Concepts: Testing Constrained Functions

Concepts and the requires clause allow us to put constraints on functions or classes and other template constructs. Andreas Fertig gives a worked example including how to test the constraints

The difference between a requires-clause and a requires-expression

In July 2020 [Fertig20], I showed a requires-clause and the three valid places such a clause can be: as a requires-clause, a trailing requires-clause, and when creating a concept. But there is another requires-thing: the requires-expression. And guess what, there is more than one kind of requires-expression. But hey, you are reading an article about C++; you had it coming.

A requires-expression has a body, which itself has one or more requirements. The expression can have an optional parameter list. A requires-expression therefore looks like a function called **requires**, except for the return-type which is implicitly **bool**. See Figure 1.

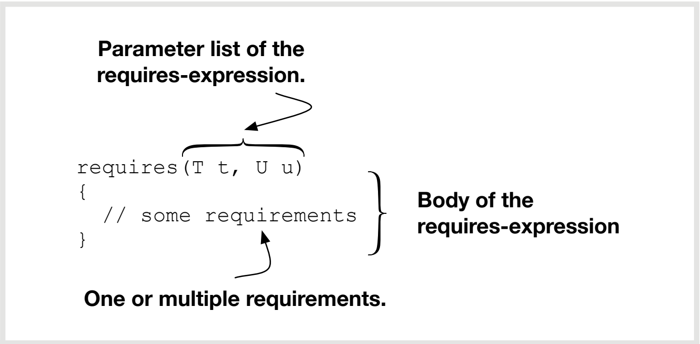

Parameter list of the requires-expression.
requires(T t, U u)
{
// some requirements
}
Body of the requires-expression
One or multiple requirements.

Figure 1

Now, inside of a requires-expression we can have four distinct types of requirements:

- Simple requirement
- Nested requirement
- Compound requirement
- Type requirement

Simple requirement

This kind of requirement asserts the validity of an expression. For example, **a + b** is an expression. It requires that there is an **operator+** for these two types. If there is one, it fulfils this requirement; otherwise, we get a compilation error.

Nested requirement

A nested requirement asserts that an expression evaluates to **true**. A nested requirement always starts with **requires**. So, we have a **requires** inside a requires-expression. And we don't stop there. With a nested requirement, we can apply a type-trait to the parameters of the requires-expression. Beware that this requires a boolean value, so either use the **_v** version of the type-trait or **::value**. Of course, this is not limited to type-traits. You can supply any expression which evaluates to **true** or **false**.

Compound requirement

With a compound requirement, we can check the return type of an expression and (optionally) whether the expressions result is **noexcept**. As the name indicates, a compound requirement has the expression in curly braces, followed by the optional **noexcept** and something like a trailing return-type. This trailing part needs to be a concept against which we can check the result of the expression.

Type requirement

The last type of requirement we can have inside a requires-expression is the type requirement. It looks much like a simple requirement, just that it is introduced by **typename**. It asserts that a certain type is valid. We can use it to check whether a given type has a certain subtype, or whether a class template is instantiable with a given type.

An example: A constrained variadic function template, add

Let's let code speak. Assume that we have a variadic function template **add**.

```
template<typename... Args>
auto add(Args&&... args)
{
  return (... + args);
}
```

It uses a fold expression to execute the plus operation on all values in the parameter pack **Args**. We are looking at a binary left fold. This is a very short function template. However, the requirements to a type are hidden. What is **typename**? Any type, right? But wait, it must at least provide **operator+**. The parameter pack can take values of different types, but what if we want to constrain it to all types be of the same type? And do we really want to allow a throwing **operator+**? Furthermore, as **add** returns **auto**, what if **operator+** of a type returns a different type? Do we really want to allow that? Oh yes, and then there is the question of whether **add** makes sense with just a single parameter which leads to an empty pack. Doesn't make much sense to me to add nothing. Let's bake all that in requirements.

We have:

1. The type must provide **operator+**
2. Only the same types are passed to **args**
3. At least two parameters are required, so that the pack is not empty

Andreas Fertig is a trainer and lecturer on C++11 to C++20, who presents at international conferences. Involved in the C++ standardization committee, he has published articles (for example, in iX) and several textbooks, most recently Programming with C++20. His tool – C++ Insights (https://cppinsights.io) – enables people to look behind the scenes of C++, and better understand constructs. He can be reached at contact@andreasfertig.com

It is surprising how many requirements we had to write for just a one-line function-template

```
// First type struct which retrieves and stores
// the first type of a pack❶
template<typename T, typename...>
struct first_type
{
  using type = T;
};

// Using alias for clean TMP❷
template<typename... Args>
using first_type_t =
  typename first_type<Args...>::type;

// Check whether all types are the same❸
template<typename T, typename... Ts>
inline constexpr bool are_same_v =
std::conjunction_v<std::is_same<T, Ts>...>;

// Concept to compare a type against the first
// type of a parameter pack ❹
template<typename T, typename... Args>
concept same_as_first_type =
  std::is_same_v<std::remove_cvref_t<T>,
  std::remove_cvref_t<first_type_t<Args...>>>;
```

Listing 1

4. **`operator+`** should be **`noexcept`**
5. **`operator+`** should return an object of the same type.

Before we start with the requires-expression, we need some additional type-traits. The function template signature only has a parameter pack. For some of the tests, we need one type out of that pack. Therefore, a type-trait **`first_type_t`** helps us to split the first type from the pack. For the check whether all types are of the same type, we define a variable template **`are_same_v`** using **`std::conjunction_v`** to apply **`std::is_same`** to all elements. Thirdly, we need a concept **`same_as_first_type`** to assert the return type with a compound requirement. It can use **`first_type_t`** to compare the return type of the compound requirement to the first type of the parameter pack. Listing 1 is a sample implementation[1].

As you can see, we expect that the compiler inserts the missing template parameter for **`same_as_first_type`** as the first parameter. In fact, the compiler always fills them from the left to the right in case of concepts.

Now that we have the tools let's create the requires-expression (see Listing 2).

The numbers of the callouts in the example match the requirements we listed earlier, which is the first step. We now have a constraint function template using three out of four possible requirements. You are probably accustomed to the new syntax, as is **`clang-format`**, but I hope you can see that we not only have constrained **`add`**, we also added documentation to it. It is surprising how many requirements we had to write for just a one-line function-template. Now think about your real-world code and how hard it is there sometimes to understand why a certain type causes a template instantiation to error.

1 Please note, C++20 ships with a concept **`same_as`**. This one here is a version which ignores cvref qualifiers and is a variadic version to retrieve the first type of a parameter pack.

```
template<typename... Args>
requires requires(Args... args)
{
  (... + args);           // Simple requirement❶
  requires are_same_v<Args...>;       // Nested
                // requirement with type-trait❷
  requires sizeof...(Args) > 1;       // Nested
       // requirement with a boolean expression
       // asserts at least 2 parameters❸
  {
    (... + args)
  }
  noexcept      //Compound requirement ensuring
                // noexcept❹
    ->same_as_first_type<Args...>;     // Same
     // compound requirement ensuring same type❺
}
auto add(Args&&... args)
{
  return (... + args);
}
```

Listing 2

Testing the constraints

Great, now that we have this super constrained and documented **`add`** function, would you believe me if I said that all the requirements are correct? No worries, I expect you not to trust me; so far, I wouldn't trust myself.

What strategy can we use to verify the constraints? Sure, we can create small code snippets which violate one of the assertions and ensure that the compilation fails. But come on, that is not great and is cumbersome to repeat. We can do better!

Whatever the solution is, so far we can say that we need a mock object that can have a conditional **`noexcept operator+`** and that that operator can be conditionally disabled. Rather than copy and paste parts, we can use a class template. We can conditionally disable a method using a NTTP and **`requires`**. Passing the **`noexcept`** status as another NTTP is simple. A mock class can look like Listing 3.

❶ we create a class template called **`ObjectMock`**, taking two NTTP of type **`bool`**. It has an **`operator+`** ❷, which has the conditional **`noexcept`** controlled by **`NOEXCEPT`**, the first template parameter and a matching return-type. The same operator is controlled by a trailing requires-clause, which disables it based on **`hasOperatorPlus`**, the second template parameter. The second version ❸ is the same, except that is returns a different type and with that does not match the expectation of the requires-expression of **`add`**. A third NTTP, **`validReturnType`**, controls two different operators ❷and ❸; it enables only one of them. In ❹, we define three different mocks with the different properties. With that we have our mock.

```
// Class template mock to create the different
// needed properties❶
template<bool NOEXCEPT, bool hasOperatorPlus,
  bool validReturnType>
class ObjectMock
{
  public:
  ObjectMock() = default;

  // Operator plus with controlled noexcept can
  // be enabled❷
  ObjectMock& operator+(const ObjectMock& rhs)
    noexcept(NOEXCEPT)
    requires(hasOperatorPlus&& validReturnType)
  {
    return *this;
  }
  // Operator plus with invalid return type❸
  int operator+(const ObjectMock& rhs)
    noexcept(NOEXCEPT)
    requires(hasOperatorPlus &&
    not validReturnType)
  {
    return 3;
  }
};
// Create the different mocks from the class
// template ❹
using NoAdd = ObjectMock<true, false, true>;
using ValidClass = ObjectMock<true, true, true>;
using NotNoexcept =
  ObjectMock<false, true, true>;
using DifferentReturnType =
  ObjectMock<false, true, false>;
```

Listing 3

A concept to test constraints

The interesting question is now, how do we test the **add** function? We clearly need to call it with the different mocks and validate that is fails or succeeds but without causing a compilation error. The answer is, we use a combination of a concept wrapped in a **static_assert**. Let's call that concept **TestAdd**. We need to pass either one or two types to it, based on our requirement that **add** should not work with just one parameter. That calls for a variadic template parameter of the concept. Inside the requires-expression of **TestAdd** we make the call to **add**. There is one minor thing, we need values in order to call **add**. If you remember, a requires-expression can have a parameter list. We can use the parameter pack and supply it as a parameter list. After that we can expand the pack when calling **add** (see Listing 4).

Wrap the test concept in a static_assert

Nice! We have a concept which evaluates to **true** or **false** and calls **add** with a given set of types. The last thing we have to do is to use **TestAdd** together with our mocks inside a **static_assert** (Listing 5).

```
template<typename... Args>
concept TestAdd =
  requires(Args... args)  // Define a variadic
                          // concept as helper ❶
{
  add(args...);           // Call add by
                          // expanding the pack❷
};
```

Listing 4

```
// Assert that type has operator+❶
static_assert(TestAdd<int, int, int>);
static_assert(not TestAdd<NoAdd, NoAdd>);

// Assert, that no mixed types are allowed❷
static_assert(not TestAdd<int, double>);

// Assert that pack has at least one parameter❸
static_assert(not TestAdd<int>);

// Assert that operator+ is noexcept❹
static_assert(not TestAdd<NotNoexcept,
  NotNoexcept>);

// Assert that operator+ returns the same type❺
static_assert(not TestAdd<DifferentReturnType,
  DifferentReturnType>);

// Assert that a valid class works❻
static_assert(TestAdd<ValidClass, ValidClass>);
```

Listing 5

In ❶, we test with **int** that **add** works with built-in types but refuses **NoAdd**, the mock without **operator+**. Next, the rejection of mixed types is tested by ❷. ❶ already ensured as a side-effect that the same types are permitted. Disallowing a parameter pack with less than two values is asserted by ❸ and therefore **add** must be called with at least two parameters. ❹ verifies that **operator+** must be **noexcept**. Second last, ❺ ensures that **operator+** returns an object of the same type, while ❻ ensures that a valid class works. We are already implicitly testing this with other tests and this is there for completeness only. That's it! We just tested the constraints of **add** during compile-time with no other library or framework! I like that.

Summary

I hope you have learned something about concepts and how to use them, but most of all, how to test them.

Concepts are a powerful new feature. While their main purpose is to add constraints to a function, they also improve documentation and help us make constraints visible to users. With the technique I have shown in this article, you can ensure that your constraints are working as expected using just C++ utilities, of course at compile-time.

If you have other techniques or feedback, please reach out to me on Twitter or via email. If you would like a more detailed introduction into Concepts, let me know. ■

Reference

[Fertig20] Andreas Fertig 'How C++20 Concepts can simpolify your code', published 7 July 2020 at https://andreasfertig.blog/2020/07/how-cpp20-concepts-can-simplify-your-code/

This article was published on Andreas Fertig's blog in August 2020 (https://andreasfertig.blog/2020/08/cpp20-concepts-testing-constrained-functions/) as a short version of Chapter 1 'Concepts: Predicates for strongly typed generic code' from his latest book *Programming with C++20*. The book contains a more detailed explanation and more information about this topic.

In Search of a Better Concurrency Model

Concurrency can get confusing quickly. Lucian Radu Teodorescu presents current plans for concurrency in the VAL programming language, comparing them with other languages.

Concurrency is hard. Really, really hard. Especially in languages like C++. First, there are the safety and correctness issues: deadlocks, race condition bugs and resource starvation. Then, we have performance issues: our typical concurrent application is far from being as fast as we hoped. Lastly, concurrent code is far harder to understand than single-threaded code.

Despite being an old problem (since 1965, older than Software Engineering), it appears that it is largely unsolved. We lack concurrent solutions that are safe, fast, and easy. This article tries to lay down the characteristics of a good concurrent model and presents a possible model that fulfils these characteristics.

The reader may be familiar with my previous work on *Structured Concurrency in C++* [Teodorescu22], which argued for a good concurrency model. Here, we aim at going beyond that model, improving on the ease-of-use side.

A concurrency model

Before diving into our analysis, we have to set some definitions. These definitions may not be universally accepted; they are tailored for the purposes of this article.

Concurrency is the set of rules that allows a program to have multiple activities (tasks) start, run, and complete in overlapping time periods; in other words, it allows the activities to be executed out-of-order or in partial order. Parallelism is the ability to execute two activities simultaneously. Concurrency enables parallelism, but we can have concurrency without parallelism (e.g., a system with a single threaded hardware that implements time slicing scheduling).

Concurrency concerns only apply at program design time. One can design concurrent software that is independent of the hardware it runs on. On the other hand, parallelism is an execution concern: whether parts of a program run in parallel or not depends on the underlying hardware (both in general and at the time of execution). For this reason, we focus on concurrency, not on parallelism.

A hardware thread (core) is an execution unit that allows code to be executed in isolation from other execution units on the hardware. Multiple hardware threads enable parallelism. Software threads (or, simply, threads) are logical execution streams that allow the expression of concurrency at the OS or programming level.

Arguably, the most important concern in concurrent design is the analysis of the overlapping activities. These activities can be repressed in the form of a directed acyclic graph, possibly with more constraints that cannot be expressed with graph links. For example, the constraint that two activities cannot be executed concurrently, without specifying which activity needs to be executed first, cannot be directly represented in the graph.

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

Race conditions, deadlocks and starvation

Race condition bugs happen when two threads share a single resource (most often, a memory location), and one thread writes to that resource while the other thread is accessing the resource. This usually means that one thread will invalidate the invariants that are assumed to be true by the other thread. Often, these problems are solved by protecting the access to the resources with mutexes (not the best approach, though).

Deadlocks happen when two threads/tasks are waiting for each other; each is waiting for the other one to resume, and both are stuck forever. This frequently happens when a resource is protected by multiple mutexes that are taken in different orders by tasks/threads.

Starvation happens when at least one thread tries to acquire a resource multiple times, and each time the access to the needed resource is denied. If person A constantly tries to get money from an ATM at noon or afternoon, and person B always gets all the money from the machine early in the morning, then person A will be starved regarding using the ATM.

To handle concurrency, there are abstractions that deal with the start and the end of these activities. We call the code executed in these abstractions *concurrency-control code*. For example: creating a thread, spawning a thread, joining a thread, waiting for a task to complete, etc. These abstractions are usually implemented in a concurrency framework.

A concurrency model is the set of programming rules and abstractions that allows us to build concurrent applications.

For simplification, this article uses the word *threads* as if all threads are CPU threads. The same ideas can be extended if we think of threads of execution as being part of other execution contexts (GPU threads, compute resources in a remote location, etc.)

Safety in a concurrent world

Goal S1. The concurrency model shall not allow undefined behaviour caused by race conditions.

For most practical purposes, this means that there shouldn't be two threads in the concurrent system that are sharing a resource in an *unprotected* way, in which at least one thread is writing to that resource.

Let us take a simple example. Suppose we have a shared variable in a C++ codebase of type **`shared_ptr<string>`**. One thread could be trying to read the string from the shared pointer, while another thread could replace the string object. The first thread assumes that the string object is still valid, while the second thread just invalidates the object. This will lead to undefined behaviour.

Adding mutexes around the accesses to the shared resource solves this problem, but creates other problems, as detailed below.

Goal S2. The concurrency model shall not allow deadlocks.

In concurrent systems with threads and locks, deadlocks are a common issue. This is not actually a safety issue *per se*; it is a correctness issue (and also a performance issue). We don't generate undefined behaviour, but we reach a state in which at least one of the threads cannot progress any further.

***

Having these two goals, concurrent programs are as safe as programs that don't have any concurrency.

One would want to also add lack of starvation as a goal here. While this is a noble goal, there is no concurrent system that can guarantee the lack of starvation. This is because this is highly dependent on the actual application. One cannot eliminate starvation without constraining the types of problems that can be expressed by the concurrent model. In other words, for all possible general concurrent systems, one can find an application that has starvation.

Fast concurrency

Goal F1. For applications that express enough concurrent behaviour, the concurrency model shall guarantee that the performance of the application scales with the number of hardware threads.

If there is enough independent work in the application that can be started, then adding more hardware threads should make the application execute faster. Amdahl's law tells us that the scale-up is not linear, but, nevertheless, we should still see the performance improvements.

Let's say, for example, that we need to process a thousand elements, and processing them can be done in parallel. If processing one element takes about 1 second, then processing all the elements on 10 cores should take about 100 seconds, and processing all the elements on 20 cores should take about 50 seconds. There are some concurrency costs when we create appropriate tasks and when we finish them, but those should be fairly insignificant for this problem.

Goal F2. The concurrency model shall not require blocking threads (keeping the threads idle for longer periods of time).

If the number of software threads is equal to the number of hardware threads, then blocking a software thread means reducing the amount of actual parallelism, thus reducing the throughput of the application.

And, because of the next goal, we generally want to have the number of software threads equal to the number of hardware threads. This implies that blocks are typically a performance problem. This means that the concurrency model should not rely on mutexes, semaphores, and similar synchronisation primitives, if they strive for efficiency.

As concurrency models usually require some synchronisation logic, this goal implies the use of lock-free schemas. Note, however, that implementations may still choose to use blocking primitives for certain operations in cases where they work faster than lock-free schemas.

Goal F3. The concurrency model shall allow limiting the oversubscription on hardware threads.

For many problems, if we have one hardware core, it's more than twice as slow to run two threads on it than to run a single thread. Creating two software threads on the same hardware thread, will not make the hardware twice as fast; thus, we start from twice as slow. Then, we have context switching, which will slow down both activities; this is why the slowdown will be more than doubled.

Try reading two books at the same time; one phrase from one book, one phrase from another book. Your reading speed will be extremely slow.

Goal F4. The concurrency model shall not require any synchronisation code during the execution of the tasks (except in the *concurrency-control code*).

If one has a task in which no other tasks are created, and no tasks are expected to be completed, then adding any synchronisation code will just slow the task down. The task should contain the same code as if it were run in an environment without concurrency.

Goal F5. The concurrency model shall not require dynamic memory allocation (unless type-erasure is requested by the user).

Dynamic memory allocation can be slow. The concurrency model shall not require any dynamic memory allocation if there is no direct need for it. For example, futures require dynamic memory allocation, even if the user doesn't need any type-erasure.

Easy concurrency

Goal E1. The concurrency model shall match the description of *structured concurrency*.

In the previous article 'Structured Concurrency in C++' [Teodorescu22], we said that an approach to concurrency is structured if the principles of structured programming [Dahl72] are applied to concurrency:

- use of abstractions as building blocks
- ability to use recursive decomposition of the program
- ability to use local reasoning
- soundness and completeness: all concurrent programs can be safely modelled in the concurrency model

If these goals are met, then writing programs in the concurrency model will be easier.

Goal E2. Concurrent code shall be expressed using the same syntax and semantics as non-concurrent code.

If there is no distinction between the concurrent code and the code that is not meant for concurrent execution, then the user will find it easier to read and reason about the concurrent code.

Of course, these principles assume that we don't make non-concurrent code more complex than it needs to be.

Goal E3. Function colouring shall not be required for expressing concurrent code.

This can be thought of as a special case of the previous goal, but it's worth mentioning separately, as it has a special connotation for concurrency. The term *function colouring* comes from Bob Nystrom's article 'What Color is Your Function?' [Nystrom15]. Bob argues that different asynchrony frameworks apply different conventions to functions (different colours), and it's hard to interoperate between such functions (colours don't mix well).

For example, in C#, one would typically add the `async` keyword to a method to signal that the method is asynchronous; then, the callers would call the function with `await` to get the real result from such an asynchronous function. This makes the interoperability between regular methods and asynchronous methods harder, making the whole concurrency model harder for the programmer.

Goal E4. Except for *concurrency-control code*, the user shall not be required to add any extra code in concurrent code versus non-concurrent code.

This goal is very similar to F4, but seen from a different perspective. It is obvious that if writing concurrent programs requires us to add extra logic, then things aren't as easy as they could be.

Ideally, the extra complexity required when writing concurrent code should be in the design of the activities that can be run concurrently, and the setting of the proper relationships between them. This is always harder than sequential code, where total ordering of activities is guaranteed.

Goal E5. The concurrency model should have a minimum set of rules for the user to follow to stay within the model.

There is no concurrency model that would not require extra thought from the programmer to write good concurrent applications. However, the fewer those thoughts are, the fewer the rules that the programmers must follow and the easier the concurrent programming would be.

CONCURRENCY MODEL	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
Locks and threads	-	-	-	-	-	-	/	-	-	+	-	-
Tasks	/	+	+	+	+	+	-	-	/	-	+	/
C# asynchronous model	/	+	+	-	-	+	-	+	/	-	+	/
C++ Coroutines	/	+	+	/	+	+	-	+	/	-	+	/
Senders/Receivers	/	+	+	+	+	+	/	+	-	-	+	/
Rust's fearless concurrency	+	/	/	-	+	-	-	-	-	+	-	-

TABLE 1

A review of past concurrency modes

We will now take a few existing concurrency models and see how they match the above goals. Please see Table 1. We mark with '+' the fact that a concurrency model fully meets a goal; we use '/' to indicate that the goal is only partially met; lastly, we use '–' to indicate that the goal is not met.

For the E5 goal, we try to give a subjective measure of the concurrency rules of the model (how many there are / how complex they are), and how easy it is for an average user to master them.

Locks and threads

This is the classic model of concurrency, in which one would explicitly create threads and use locks (mutexes, semaphores and other synchronisation primitives) to synchronise between threads.

This mode is, as one would expect, one of the worst possible models. It tends to generate a lot of safety/correctness issues, and it's hard to use correctly. Although one can create fast programs in this model, this is not guaranteed by the model itself. In fact, average concurrent code written with threads and locks is far from performant, as having numerous locks will severely slow down the application (see [Parent16, Henney17]).

Using tasks

Tasks are much better than locks and threads. If done correctly, they promise safety (although safety cannot be guaranteed by the compiler). In general, tasks can be fast, except for a dynamic memory allocation that is needed when the task is created. They don't form a model of concurrency that meets our definition of structured concurrency [Teodorescu22]. It may not be the easiest model to work with, but it is not particularly bad either.

C# asynchronous model

For the C# world, this is not a bad model. It keeps the safety guarantees of the language, it makes some efficiency tradeoffs that are fairly common, and it has a decent ease of use. Not ideal, but in line with the general philosophy of the language.

In terms of efficiency, the model has 2 problems: it requires blocking calls and it doesn't properly limit oversubscription. The two are closely related. Calling an asynchronous function from a regular one requires the use of a blocking call (e.g., `Task.Wait` or `Task.WaitAll`). Because threads can be blocked, the framework can associate multiple threads for a hardware core. To maintain good performance, the thread pool needs to do a balancing act, starting and stopping threads at runtime. This has a performance penalty, but it's considered acceptable for most C# applications.

The main reason I've included this model in my analysis is because it's a perfect example of function colouring. We have asynchronous functions (to be used in concurrent contexts), and synchronous ones (to be used in contexts in which concurrency is not required). Instead of having one syntax for both concurrent code and non-concurrent one, we have two syntaxes, each with slightly different rules. This hurts the ability to read the programs. But, to be honest, it is not that bad, if we consider the performance implications of synchronously waiting on tasks.

C++ coroutines

The concurrency model with C++ coroutines is very similar to the asynchronous model in C#. In C++, we just have language support for coroutines, without any library support for enabling a concurrent model. However, we can assume that people will be able to create one.

To interact between functions and coroutines, we need a blocking wait call, so the same tradeoffs apply as with the C# model. Although we can implement the same strategies in C++, the performance expectations for C++ are higher, so the model may not be considered very efficient for C++ applications.

The way coroutines are specified in the language, the compiler cannot guarantee (for most cases) the absence of dynamic memory allocation. For many problems, this is not acceptable.

In terms of ease of use, C++ coroutines behave similarly to C# asynchronous functions (ignoring the fact that in C++ everything is, by default, harder to use).

Senders/Receivers

The senders/receivers model is a C++ proposal named **`std::execution`** that is currently targeting C++26 [P2300R6]. It contains a set of low-level abstractions to represent concurrent computations efficiently, and it follows a structured approach.

This model really shines on efficiency. The proposal targets composable low-level abstractions for expressing concurrency, and there isn't any part of the proposal that has inherent performance costs. There is no required blocking wait. For simple computations, when expressing the entire computation flow can be stored on the stack, there is no required dynamic memory allocation. However, for most practical problems we would probably have some form of dynamic tasks, and thus we need heap allocations in one way or another.

The problem with this model is its ease of use. All the computations need to be expressed using primitives provided by the proposal. Listing 1 provides an elementary example of using this framework. Ideally, the same logic should have been encoded in a form similar to Listing 2.

The code presented in Listing 2 is not that far from how one would write concurrent code with coroutines, but we removed any syntax associated with coroutines.

Rust's fearless concurrency

Rust's default concurrency model has an odd place in our list of concurrency models [Rust]. It promises safety, but it doesn't deliver any guarantees to

```
scheduler auto sch = thread_pool.scheduler();
sender auto begin = schedule(sch);
sender auto hi = then(begin, [] {
  std::cout << "Hello world! Have an int.";
  return 13;
});
sender auto add_42 = then(hi, [](int arg) {
  return arg + 42; });
auto [i] = this_thread::sync_wait(add_42).value();
```

Listing 1

```
int hello() {
  std::cout << "Hello world! Have an int.";
  return 13;
}
void concurrent_processing() {
  thread_pool pool = ...;
  pool.activate(); // move to a different thread
  int i = hello() + 42;
  // ...
}
```

Listing 2

be free of deadlocks. It tries to provide access to lower-level primitives, but that typically leads to slow code. Moreover, it also fails to deliver on ease of use, as the low-level abstractions used for concurrency tend to cloud the semantics of concurrent code. The concurrent applications are cluttered with code needed for synchronisation.

The model is based on channels as a means of communicating between threads and mutexes for blocking threads to access protected shared resources. Mutexes are, by definition, blocking primitives; that is, they tend to slow down the applications. Channels allow two threads to communicate. The main problem with channels is that they often require blocking primitives; moreover, in most of the cases in which blocking primitives are not used, some kind of pooling is needed, which typically also leads to bad performance.

For some reason, whenever I hear the syntagm *fearless concurrency*, my mind flows to Aristotle's ethics. Aristotle defines virtues as being the golden means between two extremes, one being excess and one being deficiency. He actually gives the example of the virtue of courage as being the right balance between being a coward and being rash (fearless). Having no fear frequently means seeking danger on purpose.

While this is just wordplay, I can't help but think that being this fearless is not a virtue.

A possible future for concurrency

The model that I am about to describe is still in the inception phase. It is purely theoretical, there is no real implementation for it. I may be arguing for some model that cannot be built, but, for discovery purposes, I hope the reader will not mind my approach.

I am basing this model on the evolution and the ideas of the Val project [Val, Abrahams22a, Abrahams22b, Racordon22, Teodorescu23]. While this model was discussed in the Val community, and got support from the community, there is no official buy-in of the community towards this model. We see this as an experiment.

For the rest of this article, I will call this model the *proposed Val concurrency model*.

The concurrency model has three pillars:

- Val's commitment to safety would imply safe concurrency.
- The model aims at being as efficient as senders/receivers, but uses coroutine-like syntax.
- Reduce function colouring to make concurrent code look similar to non-concurrent code.

An example

Listing 3 shows an example of a concurrent program in Val. We have three innocent-looking functions. The **`long_task`** and **`greeting_task`** functions are regular functions; they produce integer values on the same thread they were started on. If the **`greeting_task`** function is called on the main thread, the **`long_task`** function is called on a worker thread (as it is started with **`spawn`**). The **`concurrency_example`** function is different: it starts executing on one thread (the main thread) and most probably it finishes executing on a worker thread. From the user's perspective, all three are still functions, but there is a slight generalisation regarding threading guarantees.

Figure 1 shows a graphical representation of the tasks involved in this program.

```
fun long_task(input: Int) -> Int {
  var result = input
  for let i in 0 ..< 42 {
    sleep(1)
    &result += 1
  }
  return result
}
fun greeting_task() -> Int {
  print("Hello world! Have an int.")
  return 13
}
fun concurrency_example() -> Int {
  var handle = spawn long_task(input: 0)
  let x = greeting_task()
  let y = handle.await() // switching threads
  return x + y
}
fun main() {
  print(concurrency_example())
}
```

Listing 3

Computations, senders, coroutines, and functions

In the 'Structured Concurrency in C++' article [Teodorescu22], we defined a *computation* as a chunk of work that can be executed on one or multiple threads, with one entry point and one exit point. We argued that the exit point doesn't necessarily have to be on the same thread as the entry point. This article tries to show that, if we consider computations to be the basis of computing, we can build any concurrent program on top of them.

In C++, the senders/receivers proposal can be built upon the computation model to provide structured concurrency. Every computation can be modelled with a sender. This provides an efficient basis for implementing computations.

Moreover, we can use C++ coroutines to represent computations. In fact, the senders/receivers model [P2300R6] provides a (partial) equivalence relation between senders and coroutines. Coroutines can be easier to use than senders, and thus it makes sense to consider them when designing a user-friendly concurrency model.

In the proposed Val concurrency model, all functions are coroutines, even if the user doesn't explicitly mark them. This means that Val functions can directly represent computations, and thus can be used for building concurrent programs.

With this in mind, the code from Listing 3 can be translated to C++ as something similar to the code from Listing 4. The **`long_task`** function is automatically put through a coroutine, and waiting for its result looks like calling **`co_await`** on the coroutine handle. This indicates that the proposed Val concurrency model can be easy to use.

Concurrency operations

If concurrency is defined as being the execution of activities in partial order, one of the main activities in concurrency design should be establishing the relationships between activities. That is, the rules that govern when activities can be properly started, or what needs to happen when an activity is completed. If we see the execution of the program as a directed acyclic graph, the main focus of concurrency should be on the links of the graph.

As argued above, a good concurrency model shall not require any synchronisation code except the start/end of the activities (what we called *concurrency-control code*).

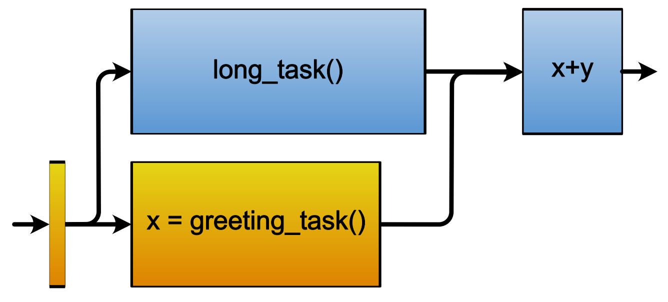


Figure 1

```
int long_task(int input) {
  int result = input;
  for (int i=0; i<42; i++) {
    sleep(1);
    result += 1;
  }
  return result;
}
task<int> long_task_wrapper(int input) {
  co_await global_task_pool.enter_thread();
  co_return long_task(input);
}

int greeting_task() {
  std:: cout << "Hello world! Have an int.";
  return 13;
}

task<int> concurrency_example() {
  auto handle = long_task_wrapper(0);
  auto x = greeting_task()
  auto y = co_await handle;
  co_return x + y;
}
```

Listing 4

In our proposed model, starting new activities is signalled by the `spawn` construct. By default, new work goes on a default thread pool. The user can change this behaviour and provide a *scheduler* (term borrowed from [P2300R6]), so the new activity will start in the specified execution context.

Calling `spawn  f()` in Val would be equivalent to a C++ sender `schedule(global_scheduler) | then(f)` and starting that sender. The result of such an operation would be represented by something like `Async<T>`. This result can then be *awaited* to get the actual value produced by the computation.

It is important to note that awaiting on such an asynchronous result may switch the current thread. This is considered an acceptable behaviour in our model.

It turns out that spawning new work, and awaiting the result of that work, are the only two activities needed to express concurrency. This is consistent with the other async/await models [Wikipedia].

C++ coroutines are not scalable

If we want programmers to avoid synchronous waits, then coroutines are pervasive. If a coroutine switches threads, then the caller coroutines also switches threads. Unless at the end of the caller we do a synchronous wait for the initial thread, the caller also needs to be a coroutine. The same reasoning then applies to its caller, and so on until we reach the bottom of the stack.

Thus, avoiding synchronous waits implies that we have to transform a large part of our functions into coroutines. This is obviously bad.

First, the function colouring problem affects most of the code. The language stops being concurrency-friendly.

Then, we have performance implications. For every coroutine we create, there is a potential heap allocation. If most of our functions need to be coroutines, this cost is much too high.

Another approach for coroutines

C++ coroutines are stackless. They don't require the entire stack to be available, and all the local variables of the coroutine will be placed on the heap (most probably). This limits the ability of the coroutine to suspend; it can only suspend at the same level as its creation point; it cannot suspend inside a called function.

Another way to model coroutines is to use stackful coroutines [Moura09]. Boost libraries provide support for such coroutines [BoostCoroutine2]. For these types of coroutines, we keep the entire stack around. We can suspend such a coroutine at any point. This alone is a big win in terms of usability.

Coming back to our initial problem with coroutines, calling stackful coroutines doesn't require special syntax or special performance penalties.

The stackful coroutines have performance downsides too. They require memory for the full stack. If, for example, a function with a deep stack creates many coroutines, we need memory to fit multiple copies of the original stack. There are ways to keep the costs under control, but there are nevertheless costs.

Using stackful coroutines for our concurrency implementation, the good news is that we would only pay such costs when we spawn new work, when we complete concurrent work, and whenever we want to switch threads. The performance costs would be directly related to the use of concurrency, which is what most users would expect.

More performance considerations

This model is consistent with other async/await models [Wikipedia]. Thus, it doesn't need any synchronisation primitives during the execution of tasks. This means that Goal F4 is met by design. Furthermore, using functional composition for asynchronous operations, our proposed model meets Goal F1, which requires that performance scales with the number of performance threads (if the application exhibits enough concurrent behaviour).

Moreover, because calling `await` doesn't imply blocking the current thread, Goal F2 is also met. The proposed model allows limiting oversubscription by using an implicit thread pool scheduler for the `spawn` calls; this means Goal F3 is also met .

Finally, there is Goal F5, which requires the model not to perform heap allocation for creation of work and getting the results out of the work. This is trickier.

As we just discussed, the stackful coroutines model may require heap allocations when a new coroutine is created. While implementations can perform tricks to amortise the cost of allocations, in essence we still require memory allocation.

Thus, our proposed model only partially meets Goal F5. All the other goals are met.

We believe that the compromise put forward by the model will turn out to be efficient for the majority of practical problems. But, time remains to tell whether we are correct.

Easiness and safety

As the reader may expect, the proposed Val concurrency model meets all the goals for ease of use. In essence, the functions of the language are also the primitives to be used for concurrency. The simplest model possible.

The set of rules that the programmer needs to know for concurrent programming is almost identical to the rules for non-concurrent programming. The difference is the semantics of `spawn` (including the use of schedulers) and the semantics of the `await` functions.

One thing that needs highlighting is local reasoning. Unlike most imperative languages, with Val's Mutable Value Semantics, a variable cannot be used in a code block that is mutated outside that code block. This takes local reasoning to its maximum: to reason about a code block, one doesn't need to consider other, non-related code. This is true for single-threaded code as well as for concurrent code.

```
fun f1() {}
fun f2() {
  var handle = spawn f1()
  handle.await() // switching threads
}
fun f3() {
  f2() // switching threads
}
```

Listing 5

```
fun process_request(c: Connection) {
  io_thread.activate() // ensure we are on
                       // I/O thead
  let incoming_request = c.read_and_parse()
  cpu_thread_pool.activate() // move to the
                             // CPU thread
  let response = handle_request(incoming_request)
  io_thread.activate() // go back to the
                       // I/O thread
  c.write_response(response)
}
```

Listing 6

In terms of safety, the model builds upon the safety guarantees of Val. This means that there cannot be any race conditions. Furthermore, because we are not using locks, there cannot be deadlocks. As both of these conditions are met, concurrent programming gets to be free of headaches, or, at least, as free as single-threaded programming.

Analysis of the proposed model

In the previous section, we focused on describing the proposed concurrency model and showing how well it meets our goals. In this section, we look at some other implications for the model.

Threads are not persistent

Imagine the code from Listing 5. Because of the **spawn** / **await** constructs from **f2**, the thread is most likely switched inside **f2** . If **f2** switches threads, then **f3** also needs to switch threads. All the functions on the stack can possibly switch threads.

In this model, threads are not persistent for the duration of a function. Functions can be started on one thread, and they may exit another thread. This might be surprising for some users, but we believe that most people will not be affected by this.

This fact can also give us some advantages. One can use sequential code that switches execution contexts, like the code from Listing 6. In this snippet, one can move between execution contexts when processing data, clearly expressing the transformation stages for processing requests, and their execution contexts.

No thread-local storage

If threads are not persistent, one cannot properly talk about thread-local storage. With every function call, the current thread can change, so thread-local storage becomes an obsolete concept.

Removing the ability to use thread-local storage is also needed for a different reason: it breaks the law of exclusivity [McCall17]. Thus, this construct should be present in languages like Val.

Interoperability

The use of stackful coroutines would make the language harder to interoperate with other languages. If other languages call Val functions, the functions need to be wrapped/adapted to ensure that the assumptions of the other languages are met. That is, the wrapped functions must guarantee that the threads won't change while executing these functions.

The problem with such adaptation of function calls is that it most likely introduces performance penalties.

Conclusions

We started this article as a search for a better concurrency model. One that allows us to write safe programs, that are fast and easy to reason about. We defined the goals for an ideal concurrency model, and we looked at past concurrency models to see how they meet our goals. Not surprisingly, most models have drawbacks, in at least one area.

In the other half of the article, we tried to sketch a new concurrency model that would (almost fully) meet all our criteria. This model is based on the goals of the Val programming language, and it aims at fully delivering a safe and easy experience, being as fast as possible.

Val is an experimental language. This concurrency model is an experiment within the Val experiment. This word play seems appropriate, as one of the goals for being structured is to be able to recursively decompose problems.

Regardless of the success or failure of the experiment, the goal appears to be worthwhile: searching for a better concurrency model. And, if everything goes well, and our experiment succeeds, the search will end with the implementation of this model. Time will tell. ■

References

[Abrahams22a] Dave Abrahams, 'A Future of Value Semantics and Generic Programming (part 1)', *C++ Now* 2022, https://www.youtube.com/watch?v=4Ri8bly-dJs

[Abrahams22b] Dave Abrahams, Dimitri Racordon, 'A Future of Value Semantics and Generic Programming (part 2)', *C++ Now* 2022, https://www.youtube.com/watch?v=GsxYnEAZoNI&list=WL

[Aristotle] Aristotle, *Nicomachean Ethics*, translated by W. D. Ross, http://classics.mit.edu/Aristotle/nicomachaen.mb.txt

[BoostCoroutine2] Oliver Kowalke, Boost: Coroutine2 library, https://www.boost.org/doc/libs/1_81_0/libs/coroutine2/doc/html/index.html

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

[Henney17] Kevlin Henney, 'Thinking Outside the Synchronisation Quadrant', *ACCU 2017 conference*, 2017, https://www.youtube.com/watch?v=UJrmee7o68A

[McCall17] John McCall, 'Swift ownership manifesto', 2017. https://github.com/apple/swift/blob/main/docs/OwnershipManifesto.md

[Moura09] Ana Lúcia de Moura, Roberto Ierusalimschy. 'Revisiting coroutines' *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2009.

[Nystrom15] Bob Nystrom, 'What Color is Your Function?', http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

[P2300R6] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, **std::execution**, 2023, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2300r4.html

[Parent16] Sean Parent, 'Better Code: Concurrency', *code::dive 2016 conference,* 2016, https://www.youtube.com/watch?v=QIHy8pXbneI

[Racordon22] Dimitri Racordon, 'Val Wants To Be Your Friend: The design of a safe, fast, and simple programming language', *CppCon 2022*, https://www.youtube.com/watch?v=ELeZAKCN4tY&list=WL

[Rust] Steve Klabnik, Carol Nichols, community 'Fearless Concurrency', https://doc.rust-lang.org/book/ch16-00-concurrency.html

[Teodorescu22] Lucian Radu Teodorescu, 'Structured Concurrency in C++', *Overload* 168, April 2022, https://accu.org/journals/overload/30/168/teodorescu/

[Teodorescu23] Lucian Radu Teodorescu, 'Value-Oriented Programming', *Overload* 173, February 2023, https://accu.org/journals/overload/31/173/teodorescu/

[Val] The Val Programming Language, https://www.val-lang.dev/

[Wikipedia] Async/await: https://en.wikipedia.org/wiki/Async/await

Meta Verse

What’s life like in cyberspace? Teedy Deigh turns on, jacks in, and checks out the immersive experience.

The sky above port 80 is the colour of a dead Slack channel.
It is said the future is already here;
It’s just not evenly distributed.
Our heroine protagonist looks out across the bleak landscape:
Soulless and chintzy.
If this place had weather, there would be wind.
If this place had tumbleweed, it would be everywhere.
The future is clearly distributed somewhere else.

There are few structures dotted across the imaginary plane.
They are echoes of another world,
Constructed by the power of marketing,
Hollowed by contact with reality.
This blasted realm is not a place that once was;
It is a place that never was,
Except in some far away land
Imagined as the Cold War turned to perestroika.
A retrofuturist synthwave vision of cyberpunk
That is as virtual and outdated as its name suggests,
A future that never came to pass because Web.

Satisfied that she has adequately set the scene –
And gazed sufficiently on the pixel-rendered bleakness before her –
Our heroine protagonist considers heading
To the nearest depopulation centre.
If this were a film, around now a single chord would be struck.
It would sound big, overproduced and decisive.

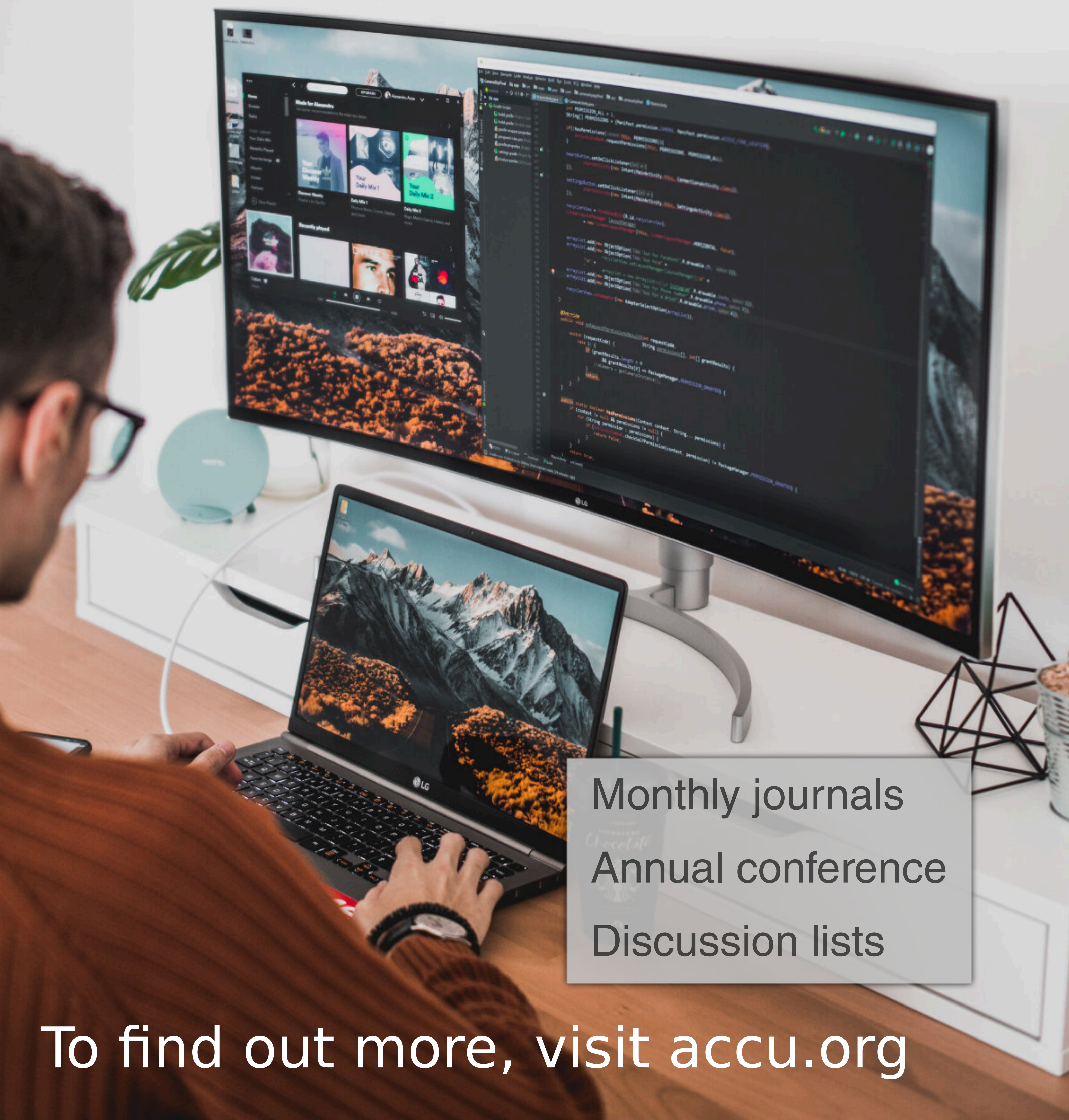
Our protagonist is not the heroine we deserve.
It’s not clear she’s even the heroine we need.
But we’re going to have to make do,
Because she’s all we’ve got and she’s got the keyboard.
What she needs now that she hasn’t got is a drink.
Even without libation, though, she is legless.
Literally.
She looks down at her avatar.
She looks the post-apocalyptic part:
Goggles, slightly too pristine made-ragged clothes,
Non-descript weapon slung across her back...
But no legs.
She would file a bug report,
But she has neither the time
Nor the patience to fight the FAQs and Contact Us system.
Perhaps that’s what the non-descript weapon is for?
She pulls up a menu to enter main street.
She ‘strolls’ past the shops.
The unreal estate speaks of squandered budgets.
The whole place reeks of hype and VCs.
She happens across an avatar.
“What’s the value proposition here, friend?”
“Am I your friend? Did you send me a request?”
“What’s all this for?”
She waves at the branding and pop-ups around them.
At her question, he lights up,
All neon and animated,
More NPC than human.
“When here in this realm,
We shall be able to get together
With friends and family.
We shall be able to work, learn, play, shop and create.
It is the promised land,
Thus spake the Book of Faces.”
Wide-eyed and fanatical,
He’s probably high on NFTs or some other Web3 scheme.
“You just described what everyone’s been doing online for years.”
“But it’s not like this, is it?”
“No, mercifully not.
I don’t get headaches from having to wear goggles.
I just click on links and get things done instead of having to
Journey through janky skeuomorphisms.”
“You are not a believer?!”
“I believe I need a drink and some aspirin.
I believe the only place I’m going to get some is IRL.”
“Pray tell, where is that?
Is it a bar in another ‘verse?
You have been to other realms?
What are they like?
Are there people?
Are they like us? Or do they have legs?
Wait... you are going?
When will you return?
Will you return?”

Our heroine protagonist heads towards the port.
There is no reason to be found here, nor any rhyme.

Although ostensibly based in the real world, **Teedy Deigh** spends most of her time living in codebases and in her head. She’s not sure what to do with the unmanaged technical debt in either case but, should the debt collectors call, cyberspace is one place she could flee where no one would be bothered to follow.

accu

professionalism in programming

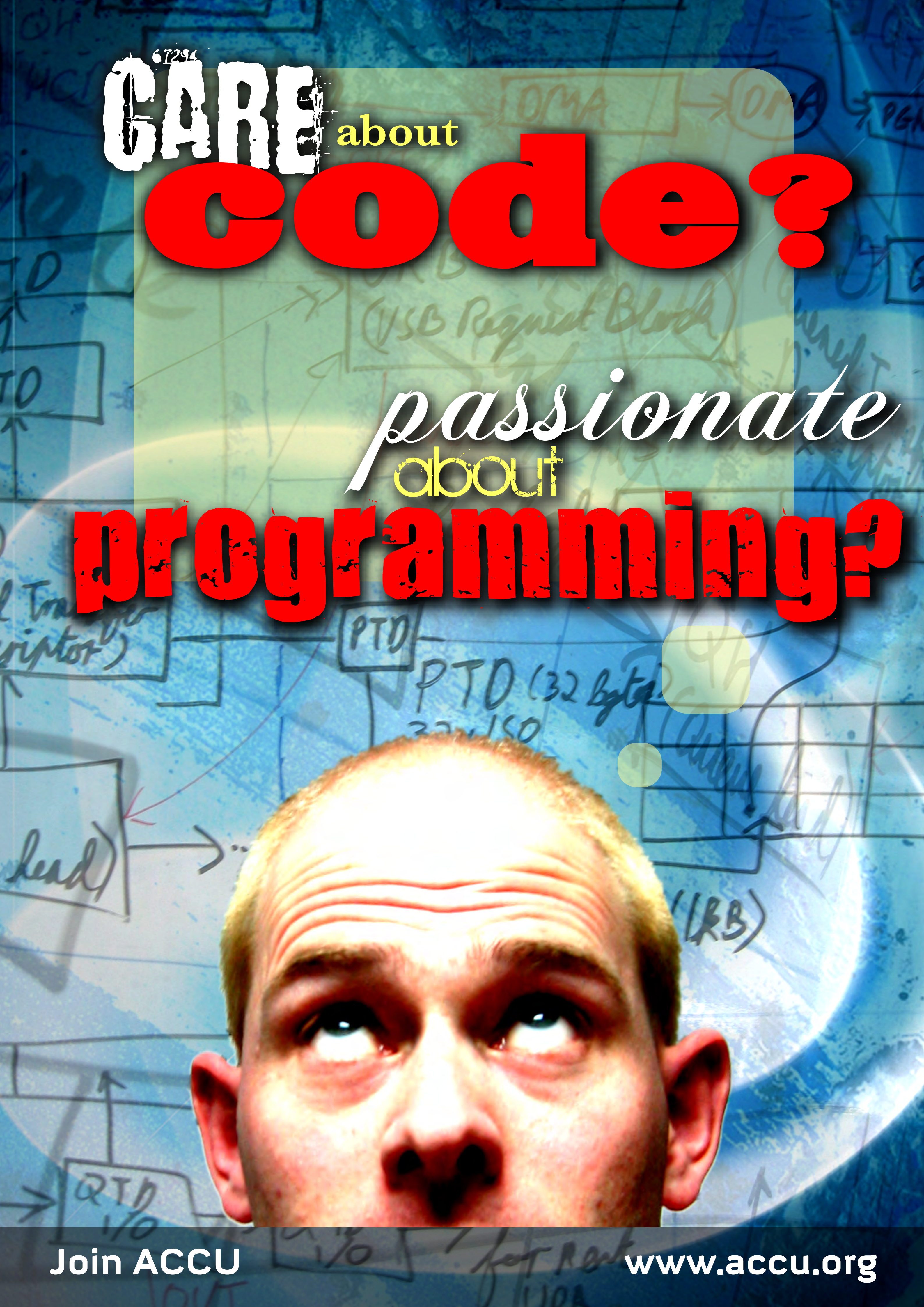


Monthly journals

Annual conference

Discussion lists

To find out more, visit accu.org


CARE about code?
passionate about programming?
Join ACCU
www.accu.org