accu
professionalism in programming

April 2025
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

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

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Daniel James. Paper lanterns at a Korean Buddhist temple.

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 Writing Senders
Lucian Radu Teodorescu describes how to implement senders.

10 C++26: Erroneous Behaviour
Sandor Dargo explains how and why uninitialised reads will become erroneous behaviour in C++26, rather than being undefined behaviour.

12 constexpr Functions: Optimization vs Guarantee
Andreas Fertig explores the use of constexpr functions and when a constexpr expression might not be evaluated at compile time.

14 UML Statecharts Formal Verification
Aurelian Melinte demonstrates how to model statecharts in Promela.

19 P271828R2: Adding mullptr to C++
Teedy Deigh attempts to help the evolution of C++ by sharing her proposal for a new state for pointers, which may not get traction, but might make you smile.

Copy deadlines

All articles intended for publication in *Overload* 187 should be submitted by 1st May 2025 and those for *Overload* 188 by 1st July 2025.

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.

self->doubt

A lack of self-confidence can be incapacitating. Frances Buontempo considers why it's important to acknowledge your doubts and remind yourself of the things that you can manage.

Do you ever freeze up and fail to get something done, because you're not certain how to proceed? For example, writing an editorial. I suspect you have noticed I frequently don't know how to proceed, making excuses every time for the lack of editorial. Imagine if you never had this problem. You could carry on with no stumbling blocks. Nothing would stop you. If you found yourself in a deep hole, you would carry on digging forever. If you were 'refactoring' a code base you would plough on for days, without wasting time trying to build the code and run the tests. Nothing could possibly go wrong, right? Hmm, maybe some self-doubt from time to time is a good thing. Perhaps a balance between freezing up and continuing regardless is what's needed.

I got stuck while writing my talk for ACCU. I mentioned my grand plan in the last *Overload* [Buontempo25a]. I intend to get my machine to play Snake, learning how via reinforcement learning. I revisited my proposal to see what I promised. I said will "start with a **`std::vector`** and progress from there" [Buontempo25b], so previous me had been helpful to current me. Starting simple is a good idea. I have managed to get a blob to move right or left, corresponding to a position in a vector. It can learn to go to the end, and get a reward. With the one-dimensional case working, I could move on to the two-dimensional case. Pleased to report I have a mini-game of snake working on a five by five grid. I represented the snake with a `+`, an apple with an `O`, and show a `*` when/if the snake crashes. It can manage to eat a few apples, but still tends to crash into itself:

```
| ++*+|
| ++  |
|    O|
|     |
|     |
-------
```

In this example, instead of going down and eating the `O`, it went up and hit itself. But, the snake had eaten a few apples first. Finding a simpler starting place made the problem seem less overwhelming.

Andrew Drakeford spoke at *CppOnline* [Drakeford25] the other week. He talked about Data-Oriented design, using data layouts and better algorithms and similar to optimize a program. He reminded me about George Pólya's book *How to solve it: A new aspect of mathematical method*. Pólya gave four steps:

1. Understand the problem,
2. Devise a plan,
3. Carry out the plan,
4. Look back and check your work.

If this fails, Pólya suggests [Wikipedia-1]:

> If you cannot solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem?

Trying a one dimensional version of snake first helped me to start thinking. Well, I say snake. There was no snake, and no apples to eat to make the snake grow. However with no apples, I can claim the single square is a hungry snake, and it does move from side to side. I still have doubts that this will extend to a proper game, but I might just about have enough to wire together to give a talk. I still have lots to learn about reinforcement learning. But I have managed something. You may well have found yourself in a similar situation, not being sure how to achieve a task. If you can break it down into smaller steps, or try a simpler proof of concept, you might manage to progress. As a Chinese proverb reminds us, "A journey of a thousand miles begins with a single step" [Wikipedia-2].

Now, solving a different problem is no guarantee you will actually be able to solve the harder problem. And a proof of concept doesn't prove much, at least in mathematical terms. Both can be helpful, at least by giving you more information, and maybe sparking a few thoughts about the wider problem. We often don't get much proof in code. We can write some tests, but might not think of all the edge cases. We can use property-based testing, getting your computer to generate values for you. If you're not familiar with property-based testing, listen to Steve Love's talk [Love24]. On the face of it, the testing framework generates random values to try, but the randomness is actually guided, trying to flush out the smallest possible examples that fail tests. The tests are written as properties, like "nothing is negative". You can regard property-based testing as a subset of fuzzers, though these are often used to try to code, looking for potential vulnerabilities. Maybe the property tests are not certainty, but they can increase confidence.

You can use other tools too, like sanitizers and linters to try to find potential problems. Recently, many people have been advocating use of **`constexpr`** and similar to attempt to eliminate undefined behaviour in C++. I believe the standard [C++] says:

> An expression E is a core constant expression unless the evaluation of E … would evaluate … an operation that would have undefined behavior.

Of course, there is a 'but'. There is an example on *Stack Overflow* of calling a library with a core constant expression that did not get a diagnostic [stackoverflow]. Although this example was actually a bug (now fixed) in GCC [GCC], using **`constexpr`** can find problems. Šimon Tóth wrote a 'Daily bit(e) of C++' on *Medium* and posted about this [Tóth24]. He gave an example of a naïve attempt at midpoint:

```
constexpr int midpoint(int a, int b) {
  return (a + b)/2; // can overflow,
                    // int overflow is UB
}
```

With input just smaller than **`std::numeric_limits<int>::max()`**, this can overflow. His example uses:

```
constexpr int a =
  std::numeric_limits<int>::max();
constexpr int b = a - 2;
constexpr int c = a - 1;
```

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.

Using a **static_assert**:

```
static_assert(midpoint(a, b) == c);
```

gives a compile error:

```
error: overflow in constant expression
[-fpermissive]
  5 |     return (a + b)/2; // can overflow,
                            // int overflow is UB
```

The **constexpr** has helped find UB. Nice. As warned by the Stack Overflow link, there will be exceptions, but the use of **static_assert** can provide less doubt.

Eliminating uncertainty is a great aspiration. However, sometimes you can't remove all doubt. There are always unknown unknowns, and often known unknowns. Making these more apparent can be useful. Seb Rose talked about this in his 'User Stories and BDD' series. In particular he said [Rose23]:

> We feel deeply uncomfortable with uncertainty and will do almost anything to avoid having to admit to any level of ignorance. Rather than focus on what we know (and discreetly ignore what we're unsure of), we should actively seek out our areas of ignorance.

Kevlin Henney dug into this in 'The Uncertainty Principle' [Henney13]. He encouraged us to see uncertainty as part of the solution rather than a problem. He suggested structuring your code so the decision doesn't matter. For example (my example), you can introduce an interface and switch between a database or lookup table. The code using the data won't need to change, then. Having more than one idea can be a good thing. Being certain there's only one solution is suboptimal, at least, and delusional at best.

Is confidence overrated? Maybe. It is often misunderstood. You may be familiar with confidence intervals from statistics. If someone claims to be 100% confident, they are either analyzing a simple problem, like the probability of a coin toss being a head or a tail, or they have missed the point. Claims are sometimes made that, assuming a normal distribution, a hypothesis is valid at a 95% confidence interval. This means an observation falls between the mean minus 1.96 standard deviations and the mean plus 1.96 standard deviations. If the data is normally distributed, and the mean and standard deviation have been calculated accurately, 95% of the observations would lie within these bounds. Confidence might be a misleading word to use at this point. Sometimes other disciplines use different words, for example medicine might say a 'reference range' [BMJ], which allows you to talk about 'normal' or 'abnormal'. Be careful with the word 'normal', though. I've previously written about this being misleading too [Buontempo21]. Saying 'abnormal' to refer to measurements is useful, but be careful when discussing anything non-numeric, like people.

Statistics are fundamentally imprecise. Actually, they are very precise, but always start with phrases like 'assuming a normal distribution'or similar. If you provide stats, for example when benchmarking, it's useful to provide a mean as well as a standard deviation or variance. This provides more information. The sample size can be useful too. If you use n=1, I'll doubt your results. This is equivalent to saying, "It works on my machine." OK, fine, but it might not work elsewhere. Providing more than one number is important for statistics, and having more than one result or perspective is useful. In fact, getting someone else to run your code often reveals potential improvements, or might help pin down the cause of a problem more quickly. Sharing is caring, as they say.

Now, my title, 'self->doubt', could suggest a member function. Whether it is a getter or setter is unclear. In fact, you can cause yourself doubt, calling your own setter as it were, if you're not careful. And that can become a negative feedback loop. On a computer, you would get a stack overflow. For a human, you can end up in a state of despair or burn out. If this chimes with you at the moment, take a moment if something doesn't seem to be working. Remind yourself what's gone OK – maybe you have some tests that pass. If you can narrow down the problems, either as tests or as a TODO list, you might end up with smaller, easier to solve problems. Or maybe you won't, but you have still learnt something.

It's OK to abandon a problem. You might be able to form a Plan B and solve the actual problem in a different way. Experiments that fail are still informative. However, sometimes the self-doubt comes from imposter syndrome [Wikipedia-3]. Despite evidence that you can do something, or are knowledgeable about a subject, you might feel like a fraud. Knowing you still have much to learn is a good thing. The Dunning-Kruger effect reminds us "people with limited competence in a particular domain overestimate their abilities" [Wikipedia-4]. Conversely, people who know their stuff may underestimate their abilities. Imposter syndrome goes beyond this underestimation. It can lead to anxiety, neurotic behaviour and depression. If you find yourself feeling like this, find some good friends to talk to. It's OK to say you need support.

If you find yourself doubting your knowledge or ability, don't panic. Some self-doubt can be better than overconfidence or downright arrogance. Some things are hard, and we all get stuck from time to time. A maths lecturer once told me to relax and enjoy a problem if I got stuck. He was probably quoting someone. However, the point is reframing feeling stuck as having something challenging to think about is helpful. It's much better than saying "Keep calm and carry on." If you're digging a hole, don't carry on. Stop, think and consider sharing the problem with someone. It might help. Maybe write a blog or article about a problem you are stuck on. That might help you get your thoughts straight and reveal what you do actually know. Doubt yourself from time to time. But allow yourself some confidence too. You're doing great.

References

[BMJ] 'Statements of probability and confidence intervals' at https://www.bmj.com/about-bmj/resources-readers/publications/statistics-square-one/4-statements-probability-and-confiden

[Buontempo21] Frances Buontempo, 'It's not normal' in Overload 166, December 2021, available at https://accu.org/journals/overload/29/166/buontempo/

[Buontempo25a] Frances Buontempo, 'All the information is on the Task', *Overload* 185, Feb 2025, available at https://accu.org/journals/overload/33/185/overload185.pdf

[Buontempo25b] Frances Buontempo, 'An introduction to reinforcement learning: Snake your way out of a paper bag' (abstract), available at https://accuconference.org/2025/session/an-introduction-to-reinforcement-learning-snake-your-way-out-of-a-paper-bag

[C++] 'Constant expressions' in *Working Draft: Programming Languages – C++*, available at https://eel.is/c++draft/expr.const#5

[Drakeford25] Andrew Drakeford, 'Date Oriented Design' (abstract), available at https://cpponline.uk/session/2025/data-oriented-design/

[GCC] GCC bug report: https://gcc.gnu.org/PR105844

[Henney13] Kevlin Henney 'The Uncertainty Principle', in *Overload* 115, June 2013, https://accu.org/journals/overload/21/115/henney_1854

[Love24] Steve Love, 'Beyond Example Based Testing in .NET (dotnet), *nor(DEV):con* 2024, available at: https://www.youtube.com/watch?v=RgDGsZXPK3Y

[Rose23] Seb Rose 'User Stories and BDD – Part 2, Discovery' in *Overload* 178, December 2023 available at https://accu.org/journals/overload/31/178/rose/

[stackoverflow] 'Undefined behavior allowed in constexpr -- compiler bug?' at https://stackoverflow.com/questions/72494618/undefined-behavior-allowed-in-constexpr-compiler-bug

[Tóth24] Šimon Tóth 'Daily bit(e) of C++ | Constexpr vs Undefined Behaviour' available at https://medium.com/@simontoth/daily-bit-e-of-c-constexpr-vs-undefined-behaviour-39330bd42906

[Wikipedia-1] 'How to solve it': https://en.wikipedia.org/wiki/How_to_Solve_It

[Wikipedia-2] Chinese proverb: https://en.wikipedia.org/wiki/A_journey_of_a_thousand_miles_begins_with_a_single_step

[Wikipedia-3] 'Imposter syndrome': https://en.wikipedia.org/wiki/Impostor_syndrome

[Wikipedia-4] 'Dunning-Druger effect': https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

Writing Senders

Senders/receivers can be used to introduce concurrency. Lucian Radu Teodorescu describes how to implement senders.

In the December issue of *Overload* [Teodorescu24], we provided a gentle introduction to senders/receivers, arguing that it is easy to write programs with senders/receivers. Then, in the February issue [Teodorescu25a], we had an article that walked the reader through some examples showing how senders/receivers can be used to introduce concurrency in an application. Both of these articles focused on the end users of senders/receivers. This article focuses on the implementer's side: what does it take to implement senders?

After a section explaining some details about the execution model of senders/receivers, we have three examples in which we build three different senders, in increasing order of complexity. The examples are purposely kept as simple as possible. We didn't bother much about using **`std::move`** when we should, we didn't consider **`noexcept`** functions in depth, we reduced the amount of metaprogramming we needed to do, we didn't showcase the extra complications needed to implement the pipeable notation, and we didn't delve into advanced topics like environments and cancellation. This is meant to be an introductory article for library implementers who are writing senders.

All the code is available on GitHub [Teodorescu25b]. While last time we used the *stdexec* library [stdexec], this time we are going to use the *execution* library that is part of the Beman project [Beman]. This shows that there are multiple valid implementations of the senders/receivers framework, and there is a relatively large implementation experience.

Receivers and operation states

If people are just using frameworks based on **`std::execution`**, they mainly need to care about senders and schedulers. These are user-facing concepts. However, if people want to implement sender-ready abstractions, they also need to consider receivers and operation states – these are implementer-side concepts. As this article mainly focuses on the implementation of sender abstractions, we need to discuss these two concepts in more detail.

A receiver is defined in P2300 as "a callback that supports more than one channel" [P2300R10]. The proposal defines a concept for a receiver, unsurprisingly called **`receiver`**. To model this concept, a type needs to meet the following conditions:

- It must be movable and copyable.
- It must have an inner type alias named **`receiver_concept`** that is equal to **`receiver_t`** (or a derived type).
- **`std::execution::get_env()`** must be callable on an object of this type (to retrieve the environment of the receiver).

A receiver is the object that receives the sender's completion signal, i.e., one of **`set_value()`**, **`set_error()`**, or **`set_stopped()`**. As explained in the December 2024 issue [Teodorescu24], a sender may have different value completion types and different error completion types. For example, the same sender might sometimes complete with **`set_value(int, int)`**, sometimes with **`set_value(double)`**, sometimes with **`set_error(std::exception_ptr)`**, sometimes with **`set_error(std::error_code)`**, and sometimes with **`set_stopped()`**. This implies that a receiver must also be able to accept multiple types of completion signals.

The need for completion signatures is not directly visible in the **`receiver`** concept. There is another concept that the P2300 proposal defines, which includes the completion signatures for a receiver: **`receiver_of<Completions>`**. A type models this concept if it also models the **`receiver`** concept and provides functions to handle the completions indicated by **`Completions`**. More details on how these completions look will be covered in the example sections.

We say that a sender *can be connected* to a receiver if the receiver accepts at least the completion signals advertised by the sender. Formally, we can connect a sender **`s`** to a receiver **`r`** if **`std::execution::connect(s, r)`** is well-formed and returns an object of a type that fulfils the requirements of an **`operation_state`** concept. For a type to match this concept, the following requirements must be met:

- It must have an inner type alias of type **`operation_state_t`** (or a type derived from it) that is named **`operation_state_concept`**.
- **`std::execution::start()`** must be callable on a reference of this type.

If a sender *describes* an asynchronous task, an operation state object *encapsulates* the actual work, including the receiver's role in the entire process. Executing **`start()`** for an operation state triggers the asynchronous operation. The lifetime of the asynchronous operation corresponds to the duration of the **`start()`** execution.

There are a few conditions that must apply to an operation state, in addition to the requirements encoded in the corresponding concept:

- The object must not be destroyed during the lifetime of the asynchronous operation.
- The object must not be copied or moved after it has been created by connecting a sender.

These requirements guarantee that implementations can safely use pointers to operation states during the asynchronous operation's lifetime, as the objects remain valid.

We've just provided technical details on what it means to be a receiver and an operation state, but we have not yet given such details on what it means to be a sender. Previous articles didn't cover these details either, as they are not important for end users. A sender is a type that models (at least) the **`sender`** concept. A type models this concept if:

- It is movable and copyable.

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

The important part to focus on is the definition of just_int_sender, the actual sender type

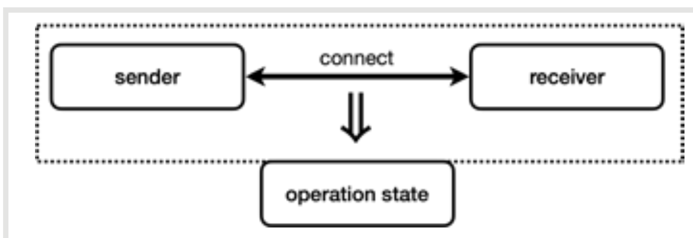

Figure 1

- Either it has an inner type alias named **sender_concept** of type **sender_t** (or derived), or it is an awaitable (in a special way, compatible with senders).
- **std::execution::get_env()** can be called on an object of this type (to retrieve the environment parameters of the sender).

In addition to the **sender** concept, the proposal also defines the **sender_in** concept (to check whether a sender can create an asynchronous operation in a given environment) and **sender_to** (to check whether the sender can be connected with a given receiver type).

The relationship between senders, receivers, and operation states is depicted in Figure 1.

A just example

Let's try to implement a very basic sender. Probably the simplest sender in P2300 is the one created by **just()**. We will attempt to create a simplified version of this. More specifically, our sender will always complete with an **int** value.

Listing 1 shows the main implementation of our sender.

First, in all our examples, we include the **execution.hpp** header from the Beman libraries. This allows us to utilise and extend the **std::execution** framework. Additionally, we use **ex** as a shorthand for the **beman::execution** namespace, which is the implementation of what C++26 will provide in the **std::execution** namespace.

Similar to the algorithms in P2300, we provide an algorithm that can create our sender; the implementation is straightforward. The important part to focus on is the definition of **just_int_sender**, the actual sender type. Probably the most important aspect of this type, which is not directly visible, is that it models the **ex::sender** concept.

We define an inner type **sender_concept** that aliases **ex::sender_t** to explicitly indicate to the senders/receivers framework that this is intended to be a sender type. This is simply how the framework is designed to work.

Secondly, we define a **completion_signatures** inner type that specifies how the sender is expected to complete. In our case, we indicate that the sender can only complete with a **set_value(int)** signal. We will see more complex completion signature definitions later, but the reader can observe that the framework makes it straightforward to declare a sender's completion signatures.

```
#include <beman/execution/execution.hpp>

namespace ex = beman::execution;

struct just_int_sender {
  // The data of the sender.
  int value_to_send_;

  // This is a sender type.
  using sender_concept = ex::sender_t;

  // This sender always completes with an 'int'
  // value.
  using completion_signatures =
    ex::completion_signatures
    <ex::set_value_t(int)>;

  // No environment to provide.
  ex::empty_env get_env() const noexcept {
    return {};
  }
  // Connect to the given receiver, and produce
  // an operation state.
  template <ex::receiver Receiver>
  auto connect(Receiver receiver) noexcept {
    return detail::just_int_op{value_to_send_,
      receiver};
  }
};
auto just_int(int x) {
  return just_int_sender{x}; }
```

Listing 1

A sender needs to have an environment, but in our case, there is no environment to provide. So we simply provide an empty environment, which is literally defined as:

```
struct empty_env {};
```

The implementation of the operation state type is given in Listing 2.

```
namespace detail {
template <ex::receiver Receiver>
struct just_int_op {
  int value_to_send_;
  Receiver receiver_;

  // This is an operation-state type.
  using operation_state_concept =
    ex::operation_state_t;

  // The actual work of the operation state.
  void start() noexcept {
    // No actual work, just send the value
    // to the receiver.
    ex::set_value(std::move(receiver_),
      value_to_send_);
  }
};
}
```

Listing 2

an operation state needs to store the receiver object connected to the sender so that it knows which object should receive the completion signal

Following the same pattern used for senders, an operation state must declare an inner type named `operation_state_concept`, which must be `ex::operation_state_t` (or a type derived from it). Besides this, the only other required operation for an operation state is `start()`. This is called to actually execute the asynchronous operation described by the sender, while also sending the completion signal to the receiver.

The reader should note that an operation state needs to store the receiver object connected to the sender so that it knows which object should receive the completion signal. The actual completion signal invocation is expressed as:

```
ex::set_value(std::move(receiver_),
  value_to_send_);
```

The `set_value` call is marked as `noexcept`, meaning the user does not need to check for exceptions (or potentially invoke `set_error`); this simplifies the entire process.

The reader should take a few moments to observe the interaction between the sender, the receiver, and the resulting operation state, and how the entire flow works. There are no advanced concepts or concurrency concerns in this example. Pretty easy, right?

With this sender, the user might write something like

```
ex::sender auto work = just_int(13)
| ex::then([] (int x) { printf(
"Hello, world! Here is a received value: %d\n",
x); });
```

to declare a sender that, when executed, will print a message containing value `13`.

And then, another example

Now, let's discuss a slightly more complex example. This time, we want to implement a `then` sender – a simpler version of the sender with the same name in the P2300 proposal [P2300R10].

The main code for this sender is presented in Listing 3.

```
template <ex::sender Previous,
  std::invocable<int> Fun>
struct then_sender {
  Previous previous_;
  Fun f_;

  using sender_concept = ex::sender_t;
  ex::empty_env get_env() const noexcept {
    return {}; }

  using completion_signatures =
    ex::completion_signatures<
      ex::set_value_t(int),
      ex::set_error_t(std::exception_ptr),
      ex::set_stopped_t()>;

  template <ex::receiver Receiver>
  auto connect(Receiver receiver) noexcept {
    return ex::connect(previous_,
      detail::then_receiver{f_, receiver});
  }
};

template <ex::sender Previous,
  std::invocable<int> Fun>
then_sender<Previous, Fun> then(Previous prev,
    Fun f) {
  return {prev, f};
}
```

Listing 3

Here, there are three key differences compared to the previous example: using a preceding sender, defining slightly more complex completion signatures, and implementing `connect` in a different way.

`then` is a sender adapter. That is, it takes a *previous* sender and chains some extra work on top of it to create a new sender. More specifically, in this case, it executes the given invocable after the previous sender completes. The invocable receives the value produced by the previous sender. In our case, the previous sender must complete with an `int` value if it completes successfully (as we will see shortly).

In P2300, all sender adapters have two equivalent forms: one that takes the previous sender as an argument and one that is pipeable. For example, `ex::then(ex::just(), f)` and `ex::just() | ex::then(f)` both generate the same sender value. To achieve the pipeable form, we need a different overload for the `then` function – one that returns an expression template that can be combined via `operator |` with another sender. For simplicity, we leave out the implementation of the pipeable sender. Implementing such a form can be a good exercise for the reader.

Unlike our previous example, we now advertise multiple completion signatures: a successful completion with an `int` value, an error completion with an exception, and a stopped completion. The `completion_signatures` helper from the senders/receivers framework easily accommodates specifying multiple completion signatures.

In this example, we simply assume that the value completion signature needs to be `int`. However, more generic implementations might deduce this from the result of the given invocable. Furthermore, such generic implementations would likely ensure that the value completion signals of the previous sender match the type of the invocable and would also conditionally add support for error and stopped completions. This easily turns into a metaprogramming exercise. For simplicity, we leave these details out.

Finally, we need to explain the implementation of the `connect` method. In this example, we take a different approach to implementing it and returning an appropriate operation state. Instead of defining the actual resulting operation state, we define the receiver that needs to be connected to the previous sender and place our logic inside that receiver. This intermediate receiver establishes the connection between the previous sender and the receiver connected to our sender.

implementing sender adaptors is not particularly complicated – most of the complexity arises from making the implementation generic

The code for this receiver is shown in Listing 4. The implementation is relatively straightforward. First, we declare that this is a receiver type by using the `receiver_concept` inner type; this follows the same pattern as for senders and operation states. Then, in addition to storing the necessary data, we implement the methods that will receive the completion signals from the previous sender – in our case: `set_value()`, `set_error()`, and `set_stopped()`. All these methods must be marked as `noexcept`.

When connecting the previous sender to this receiver, the framework will check that all the advertised completion signals of the sender have a corresponding method in the receiver and that the types match. This means that if the previous sender successfully completes with a value that is not of type `int`, it cannot be connected to our sender.

The `set_error()` and `set_stopped()` cases are straightforward: we simply forward the signal to the next receiver. The main logic is handled in the `set_value()` method. Here, after receiving the value from the previous sender, we call the given invocable and pass the result to the next receiver. Since the call to `f_` can throw, we need to catch any exceptions and pass them as errors to the next receiver– this is a common pattern in implementing senders.

And that's it. As this example demonstrates, implementing sender adaptors is not particularly complicated. Most of the complexity arises from making the implementation generic: detecting the completion signals of the previous sender, ensuring they match the signature of the invocable, generating appropriate completion signatures, handling r-value objects, and so on.

A serializer example

In this section, we have shown an example of implementing a sender that addresses some concurrency concerns. We aim to implement a *serializer context* and a corresponding sender that ensures only one work item can be executed at a given time within the context. The context is similar to `std::mutex`, and the sender is similar to `std::lock_guard`, which can be obtained from the mutex.

```
namespace detail {
template <ex::receiver Receiver, typename Fun>
struct then_receiver {
  Fun f_;
  Receiver receiver_;

  using receiver_concept = ex::receiver_t;

  void set_value(int value) noexcept {
    try {
      ex::set_value(std::move(receiver_),
        f_(value));
    } catch (...) {
      ex::set_error(std::move(receiver_),
        std::current_exception());
    }
  }
  void set_error(std::exception_ptr e) noexcept {
    ex::set_error(std::move(receiver_), e); }
  void set_stopped() noexcept {
    ex::set_stopped(std::move(receiver_)); }
};
}
```

Listing 4

```
ex::sender auto branch1 = ex::schedule(sched)
  | let_value([]{ return work1; });
ex::sender auto branch2 = ex::schedule(sched)
  | let_value([]{ return work2; });
ex::sender auto branch3 = ex::schedule(sched)
  | let_value([]{ return work3; });
ex::sync_wait(ex::when_all(branch1, branch2,
   branch3));
```

Listing 5

Having such a facility would allow users to migrate more easily to senders/receivers without significantly altering their business logic. In terms of resource usage, the serializer will typically be more efficient than a mutex: it won't block any threads and will always ensure the optimal execution of work items. This is not a new idea (see, for example, [Intel] and [Kohlhoff23]), and my hope is that it will be standardised in the next release cycle (C++29).

Let's first look at a usage example. The code in Listing 5, without such a serializer, will likely execute `work1`, `work2`, and `work3` concurrently (`work1`, `work2`, and `work3` are senders). Listing 6 shows how the code can be modified using our serializer to ensure that `work1`, `work2`, and `work3` cannot be executed concurrently; at most, one of them will be executed at a given time.

Listing 7 (on next page) shows a simple implementation of the `serializer_context` class. It has two public methods: one that is called when new work needs to be executed within the context, and one that notifies the context when the work is done.

When `on_serializer` wants to start some work, it may be that the context is already in the process of executing something. This means that the work needs to be delayed. Thus, we need to *store* the work for later. The way we do this is by encapsulating the work inside a `std::function<void()>` object.

When enqueuing work, we have two scenarios: one where there is no ongoing work, and one where work is already being executed on the serializer. In the first case, we execute the work immediately, and in the second, we store the work for later execution in a vector. Of course, we

```
serializer_context ctx;
ex::sender auto branch1 =
  on_serializer(ex::schedule(sched), ctx, work1);
ex::sender auto branch2 =
  on_serializer(ex::schedule(sched), ctx, work2);
ex::sender auto branch3 =
  on_serializer(ex::schedule(sched), ctx, work3);
ex::sync_wait(ex::when_all(branch1, branch2,
  branch3));
```

Listing 6

The implication of this strategy is that one scheduler may queue up a significant amount of work to be executed, which is not always desirable

```
struct serializer_context {
  using continuation_t = std::function<void()>;
  // Called when new work needs to be enqueued
  void enqueue(continuation_t cont) {
    {
      std::lock_guard<std::mutex>
        lock{bottleneck_};
      if (busy_) {
        // If we are busy, we need to enqueue
        // the continuation
        to_run_.push_back(std::move(cont));
        return;
      }
      // We are free; mark ourselves as busy,
      // and execute continuation inplace
      busy_ = true;
    }
    cont();
  }
  // Called when the work completes
  void on_done() {
    continuation_t cont;
    {
      std::lock_guard<std::mutex>
        lock{bottleneck_};
      assert(busy_);
      if (to_run_.empty()) {
        // Nothing to run next, we are done
        busy_ = false;
        return;
      }
      // We have more work to do; extract the
      // first continuation
      cont = std::move(to_run_.front());
      to_run_.erase(to_run_.begin());
    }
    if (cont) {
      cont();
    }
  }
private:
  bool busy_{false};
  std::vector<continuation_t> to_run_;
  std::mutex bottleneck_;
};
```

Listing 7

need to synchronise access to the vector and to the flag that indicates whether execution is currently in progress.

When a chunk of work has finished executing on the serializer, we again have two cases: one where there is no pending work on the serializer, and one where there is pending work. In the first case, we should simply exit, marking the serializer as not busy. In the second case, we need to start executing the next work item.

The implication of this strategy is that one scheduler may queue up a significant amount of work to be executed, which is not always desirable. To address this issue, one might add a scheduler to the context and always

```
template <ex::sender Previous, ex::sender Work>
struct on_serializer_sender {
  Previous previous_;
  serializer_context& context_;
  Work work_;

  using sender_concept = ex::sender_t;
  using completion_signatures =
    ex::completion_signatures<
      ex::set_value_t(),
      ex::set_error_t(std::exception_ptr),
      ex::set_stopped_t()>;
  ex::empty_env get_env() const noexcept {
    return {}; }

  template <ex::receiver Receiver>
  auto connect(Receiver receiver) noexcept {
    return ex::connect(previous_,
      detail::on_serializer_receiver{context_,
      work_, receiver});
  }
};
template <ex::sender Previous, ex::sender Work>
on_serializer_sender<Previous,
    Work> on_serializer(Previous prev,
    serializer_context& ctx, Work work) {
  return {prev, ctx, work};
}
```

Listing 8

execute pending work on this scheduler. We leave this as an exercise for the reader.

Moving on, the code that implements the actual sender is presented in Listing 8, and its intermediate receiver in Listing 9. The sender code doesn't contain anything particularly noteworthy. It simply follows the same pattern we've seen before: `sender_concept`, `completion_signatures`, `get_env()`, and `connect`.

As in the previous example, the key logic happens in the `set_value()` method of the intermediate receiver. Here, we enqueue a lambda into the serializer context that must perform three actions: execute the given work, notify the context when the work is done, and signal the final receiver with a completion signal.

For the first two actions, we create a sender that will execute them; we name this sender `work_and_done`. This is a straightforward composition of the original work and an `ex::then` with a lambda that calls `on_done` on the context. Then, to cover the third action, we connect this sender to the final receiver object, storing the resulting operation state in the variable `op`. We then call `start` on this operation state to actually execute everything. To summarise, this will first execute the `work_and_done` sender, which will first execute the given work, then call `on_done` on the context, and finally trigger a completion signal to the final receiver.

The `start()` call will last as long as this operation needs to execute. This means that the lifetime of `op` is slightly longer than this operation.

the requirement we have for operation states ... forbids copying and moving operation state objects while the asynchronous operation is running

```
namespace detail {
template <ex::receiver Receiver, ex::sender Work>
struct on_serializer_receiver {
  serializer_context& context_;
  Work work_;
  Receiver receiver_;

  using receiver_concept = ex::receiver_t;

  void set_value() noexcept {
    context_.enqueue([this] {
      ex::sender auto work_and_done =
        work_ | ex::then([this] {
          context_.on_done(); });
      auto op = ex::connect(work_and_done,
        std::move(receiver_));
      ex::start(op);
    });
  }
  void set_error(std::exception_ptr e) noexcept {
    ex::set_error(std::move(receiver_), e); }
  void set_stopped() noexcept {
    ex::set_stopped(std::move(receiver_)); }
};
}
```

Listing 9

This is a common pattern when implementing code that manually runs operation states. It is also well aligned with the requirement we have for operation states, which forbids copying and moving operation state objects while the asynchronous operation is running (see above).

Depending on the workload of the context, this lambda might be executed immediately or deferred to a later execution. In both cases, we uphold the guarantees of the serializer and the expectations of senders.

Conclusions

In this article, we presented three examples to demonstrate that writing senders is not very complicated and can be done relatively easily by regular C++ engineers. There are a few concepts that need to be understood (we covered the basics in the first part of the article), but once one is familiar with those concepts, writing new senders is not difficult. Of course, senders that deal with complex concurrency concerns are harder to write, as they require extra care on the concurrency side, but this is inherent to the problem being solved.

Through these examples, we also reinforced the idea presented in the previous two articles ([Teodorescu24], [Teodorescu25a]) that the senders/receivers framework has great composability features.

If the first two articles in this series conveyed the message that senders are easy to use but probably harder to implement, this article aims to show that senders are also easy to implement. However, there is a caveat: while it is easy to implement regular senders, implementing fully generic senders can be more challenging. That said, most users will not need to implement fully generic senders. Thus, for most programmers, writing senders should be a straightforward task.

Senders/receivers are not complicated. People just need to spend some time getting acquainted with how they work. They are often compared to the introduction of iterators for generic programming: they may not be the first tool a novice programmer reaches for, but after some practice, their value is inestimable. I truly believe that the same description applies to senders/receivers. ■

References

[Beman] Dietmar Kühl and other Beman project contributors, execution, available at: https://github.com/bemanproject/execution

[Intel] Intel, 'Local Serializer' in *Intel® oneAPI Threading Building Blocks Developer Guide and API Reference*, available at: https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-12/local-serializer.html

[Kohlhoff23] Christopher M. Kohlhoff, 'Strands: Use Threads Without Explicit Locking', *boost C++ Libraries*, available at: https://www.boost.org/doc/libs/1_87_0/doc/html/boost_asio/overview/core/strands.html

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

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

[Teodorescu24] Lucian Radu Teodorescu, 'Senders/receivers: An Introduction', *Overload* 184, December 2024, available at: https://accu.org/journals/overload/32/184/teodorescu/

[Teodorescu25a] Lucian Radu Teodorescu, 'Using Senders/Receivers', *Overload* 185, February 2025, available at: https://accu.org/journals/overload/33/185/teodorescu/

[Teodorescu25b] Lucian Radu Teodorescu, overload186_sr_examples (code for the article) available at: https://github.com/lucteo/overload186_sr_examples

C++26: Erroneous Behaviour

C++'s undefined behaviour impacts safety. Sandor Dargo explains how and why uninitialised reads will become erroneous behaviour in C++26, rather than being undefined behaviour.

If you pick a random talk at a C++ conference these days, there is a fair chance that the speaker will mention safety at least a couple of times. It's probably fine like that. The committee and the community must think about improving both the safety situation and the reputation of C++.

If you follow what's going on in this space, you are probably aware that people have different perspectives on safety. I think almost everybody finds it important, but they would solve the problem in their own way.

A big source of issues is certain manifestations of undefined behaviour. It affects both the safety and the stability of software. I remember that a few years ago when I was working on some services which had to support a 10× growth, one of the important points was to eliminate undefined behaviour as much as possible. One main point for us was to remove uninitialized variables which often lead to crashing services.

Thanks to P2795R5 by Thomas Köppe, uninitialized reads won't be undefined behaviour anymore – starting from C++26. Instead, they will get a new behaviour called 'erroneous behaviour'.

The great advantage of erroneous behaviour is that it will work just by recompiling existing code. It will diagnose where you forgot to initialize variables. You don't have to systematically go through your code and let's say declare everything as auto to make sure that every variable has an initialized value. Which you probably wouldn't do anyway.

But what is this new behaviour that on C++ Reference is even listed on the page of undefined behaviour? [CppRef-1] It's well-defined, yet incorrect behaviour that compilers are **recommended** to diagnose. *Is recommended enough?!* Well, with the growing focus on safety, you can rest assured that an implementation that wouldn't diagnose erroneous behaviour would be soon out of the game.

Some compilers can already identify uninitialized reads – what nowadays falls under undefined behaviour. For example, clang and gcc with `-ftrivial-auto-var-init=zero` have already offered default initialization of variables with automatic storage duration. This means that the technique to identify these variables is already there. The only thing that makes this approach not practical is that you will not know which variables you failed to initialize.

Instead of default initialization, with erroneous behaviour, an uninitialized object will be initialized to an implementation-specific value. Reading such a value is a conceptual error that is recommended and encouraged to be diagnosed by the compiler. That might happen through warnings, run-time errors, etc.

Sandor Dargo is 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

```
void foo() {
  int d;  // d has an erroneous value
  bar(d); // that's erroneous behaviour!
}
```

So, looking at the above example, ideally `int d;` should be already diagnosed at compile-time as a warning. If it's ignored, at some point, `bar(d);` will have an effect during program execution, but it should be well-defined, unlike undefined behaviour where anything can happen.

It's worth noting that undefined behaviour and having erroneous values is not possible in constant expressions. In other words, `constexpr` protects from it.

Initializing an object to anything has a cost. What if you really want to avoid it and initialize the object later? Will you be able to still do it without getting the diagnostics? Sure! You just have to be deliberate about that. You cannot just leave values uninitialized by accident, you must mark them with C++26's new attribute, `[[indeterminiate]]`.

We must notice in the example, that `d` doesn't have an erroneous value anymore. Now its value is simply indeterminate [CppRef-2]. On the other hand, if we later use that variable still without initialization, it's undefined behaviour!

Above, we've only talked about variables with automatic storage duration. That's not the only way to have uninitialized variables. Moreover, probably it's not even the main way, think about dynamic storage duration, think about pointers! Also, if any member is left uninitialized, the parent object's value will be considered either indeterminate or erroneous. See Listing 1.

Not only variables variables but function parameters can also be marked `[[indeterminate]]`. See Listing 2 (next page).

At the point of writing (January 2025), no compiler provides support for erroneous behaviour.

```
struct S {
  S() {}
  int num;
  std::string text;
};

int main() {
  [[indeterminate]] S s1; // indeterminate value
  std::cout << s1.num << '\n'
        // this is UB as s1.num is indeterminate

  S s2;
  std::cout << s2.num << '\n'
        // this is still UB, s2.num is an
        // erroneous value
}
```

Listing 1

Soon, compilers will be recommended to diagnose every occurrence of reads of uninitialized variables and function parameters

```
struct S {
  S() {}
  int num;
  std::string text;
};

void foo(S s1 [[indeterminate]], S s2)
{
  bar(s1.num); // undefined behavior
  bar(s2.num); // erroneous behavior
}
```

Listing 2

Conclusion

C++26 introduces erroneous behaviour in order to give well-defined, but incorrect behaviour for reading uninitialized values. Soon, compilers will be recommended to diagnose every occurrence of reads of uninitialized variables and function parameters.

Also, if something is not initialized at a given moment on purpose, you can mark it with the `[[indeterminate]]` attribute following the don't pay for what you don't need principle.

This new behaviour is a nice step forward in terms of C++'s safety. ■

References

[CppRef-1] 'Undefined behavior' on cppreference.com, available at https://en.cppreference.com/w/cpp/language/ub

[CppRef-2] 'C++ attribute: indeterminate' on cppreference.com, available at https://en.cppreference.com/w/cpp/language/attributes/indeterminate

This article was previously published on Sandor Dargo's Blog on 4 February 2025, and is available at https://www.sandordargo.com/blog/2025/02/05/cpp26-erroneous-behaviour

Join ACCU

visit

www.accu.org

for details

accu

CVu and *Overload* are written by programmers for programmers.

We need **your** contributions:

- What are you doing right now?
- What technology are you using?
- What have you learned recently, or explained to someone else?
- What techniques and idioms are you using?
- What worked, and what didn't?

Not written for publication before? Don't worry, our editorial team is here to help.

Even if your ideas aren't fully formed, get in touch for advice and guidance:

- cvu@accu.org
- overload@accu.org

constexpr Functions: Optimization vs Guarantee

Constexpr has been around for a while now, but many don't fully understand its subtleties. Andreas Fertig explores its use and when a constexpr expression might not be evaluated at compile time.

The feature of constant evaluation is nothing new in 2023. You have constexpr available since C++11. Yet, in many of my classes, I see that people still struggle with `constexpr` functions. Let me shed some light on them.

What you get is not what you see

One thing, which is a feature, is that `constexpr` functions can be evaluated at compile-time, but they can run at run-time as well. That evaluation at compile-time requires all values known at compile-time is reasonable. But I often see that the assumption is once all values for a `constexpr` function are known at compile-time, the function will be evaluated at compile-time.

I can say that I find this assumption reasonable, and discovering the truth isn't easy. Let's consider an example (Listing 1).

```
constexpr auto Fun(int v)
{
  return 42 / v; ❶
}
int main()
{
  const auto f = Fun(6); ❷

  return f; ❸
}
```

Listing 1

The `constexpr` function `Fun` divides 42 by a value provided by the parameter `v` ❶. In ❷, I call `Fun` with the value `6` and assign the result to the variable `f`.

Last, in ❸, I return the value of `f` to prevent the compiler optimizes this program away. If you use Compiler Explorer to look at the resulting assembly, GCC with `-O1` brings this down to:

```
main:
        mov     eax, 7
        ret
```

As you can see, the compiler has evaluated the result of 42 / 6, which, of course, is 7. Aside from the final number, there is also no trace at all of the function `Fun`.

Now, this is what, in my experience, makes people believe that `Fun` was evaluated at compile-time thanks to `constexpr`. Yet this view is incorrect. You are looking at compiler optimization, something different from `constexpr` functions.

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

```
auto Fun(int v)
{
  return 42 / v; ❶
}
int main()
{
  const auto f = Fun(6); ❷

  return f; ❸
}
```

Listing 2

Let's remove the constexpr from the function first (Listing 2).

The resulting assembly, again GCC and `-O1` is the following:

```
Fun(int):
        mov     eax, 42
        mov     edx, 0
        idiv    edi
        ret
main:
        mov     eax, 7
        ret
```

Okay, that looks more like proof that `constexpr` helped before. You now can see the function `Fun`, but the result is still known in `main`. Why is that?

The reason is that `constexpr` implies inline! Try for yourself, make `Fun inline`, and you will see exactly the same assembly output as when the function was `constexpr`.

Because of the implicit `inline`, the compiler understands that `Fun` never escapes the current translation unit. By knowing that there is no reason to keep the definition around. Then, `Fun` itself is reasonably simple to the compiler, and the parameter is known at compile-time. An invitation for the optimizer, which it happily accepts.

You can alter the code even more, and the optimizer will still be able to produce the same result. Have a look at the changes I made to the original code in Listing 3.

```
inline auto Fun(int v) ❶
{
  return 42 / v;
}

int main()
{
  int  val{6};        ❷
  auto f = Fun(val); ❸

  return f;
}
```

Listing 3

Remember that I started by stating that distinguishing optimization from the guarantee is difficult?

Fun is now inline as ❶ shows. The input to **Fun** is now a non-**const** variable **var** ❷, and the result of the call to **Fun** in ❸ is stored in a non-**const** variable. All just run-time code. Except that the compiler can still see that the input to **Fun** is always 6. With this knowledge the compiler gets its friend the optimizer onboard and the result is the same as with the initial code that looked way more constant then this version.

What you see here is still an optimization. Yes, if you are interested in a small binary footprint, you will be happy. But, **constexpr** can give you more! You can get guarantees from **constexpr**. Let's explore that.

Ways to enforce constant evaluation

The current code does not force the compiler to evaluate **Fun** at compile-time in a manner that could cause compile-time evaluation to fail. The evaluation could silently fail for integral data types declared **const**, which isn't allowed with **constexpr**. Essentially, you must force the compiler into a compile context for the evaluation. You have roughly four options for doing so:

- assign the result of **Fun** to a **constexpr** variable;
- use **Fun** as a non-type template argument;
- use **Fun** as the size of an array;
- use **Fun** within another **constexpr** function that is forced into constant evaluation by one of the three options before.

In Listing 4, you find the four cases in code.

Enforcing constant evaluation

So far, I have done neither of the four variants, time to change this. Let me make the variable **f constexpr** (Listing 5).

Once you look at the resulting assembly, you see ... no change compared to the initial example. Remember that I started by stating that distinguishing optimization from the guarantee is difficult?

My example now comes with the guarantee that **Fun** is evaluated at compile-time. However, since there is no difference between the former version in the resulting assembly, what is my point?

```
constexpr auto Other(int v)
{
  return Fun(v);
}

int main()
{
  constexpr auto f{Fun(6)};
  int            data[Fun(6)]{}; // Please prefer
                        // the std::array solution
  std::array<int, Fun(6)> data2{};
  constexpr auto          ff{Other(6)};
}
```

Listing 4

```
constexpr auto Fun(int v)
{
  return 42 / v; ❶
}

int main()
{
  constexpr auto f = Fun(6); ❷

  return f; ❸
}
```

Listing 5

Well, time to start talking about the guarantee.

What if, and please don't be shocked, I replace **6** with **0** in my call to **Fun**? Urg, yes, that will result in a division by zero. Who, aside from Chuck Norris, can divide by zero? At least, I can't, and neither can any of the compilers I use.

But the initial example, despite the fact that **Fun** is **constexpr**, compiles just fine. Well, this little warning about the division by zero aside. Ah, yes, and the result is, well, potentially the result to expect if one of us could divide by zero.

The guarantee

Make the variable **f** in ❷ **constexpr**, or choose another way to force the compiler into constant evaluation. The result? If you make the change, your compile will fail, and the compiler tells you the obvious: a division by zero does not produce a constant value. This is what **constexpr** functions bring you: an evaluation free of undefined behavior!

Putting **constexpr** on a function only gives you a small part of **constexpr**. Only by using a **constexpr** function in a context requiring constant evaluation will you get the full benefits out of it, no undefined behavior.

I hope this article helps you better understand what **constexpr** can offer and how to distinguish the guarantee from a compiler's optimization. ■

References

The code listings are available on godbolt:

- Listing 1: https://godbolt.org/z/chG8oe3TG
- Listing 2: https://godbolt.org/z/85W5Mdv4T
- Listing 4 (with the extra needed for it to compile): https://godbolt.org/z/ddrGrYr3M
- Listing 5: https://godbolt.org/z/4Y5nraeYG

This article was published on Andreas Fertig's blog on 6 June 2023, and is available at: https://andreasfertig.com/blog/2023/06/constexpr-functions-optimization-vs-guarantee/

UML Statecharts Formal Verification

Formal verification can be applied to UML statecharts. Aurelian Melinte demonstrates how to model statecharts in Promela.

How difficult is it to model an UML statechart in Promela [Spin] for formal verification? There seems to be very little information on how to do it.

Some notes about the Promela code

At its core, Promela is a modelling language, not a programming one. Even though it looks like C, the differences are significant. As such, you might find it lacks the constructs you would normally expect as a programmer and the `goto`, `define` and `inline` (with a very specific semantics) are ruling the field.

There are a few keywords that are not what you usually expect:

- `process`: a process is Promela's idea of modelling a parallel execution task. It is not an operating system process. A process executes a special `proctype` function. Processes exchange messages via channels (`chan` artefacts) and can reference global data.
- `state`: Promela turns both the model you describe with the language, and the linear temporal language formula (LTL) describing the desired behaviour of the model in time, into Büchi automatons [Wikipedia] – these automatons' states are the ones that Promela will report about. To avoid confusion with the states of the statechart, I will explicitly name these as UML-states. Plain 'states' are Promela/Büchi ones.
- `timeout`: Promela has no notion of measurable time (though you can force one in). `timeout` is a variable that is set true when the verification has not yet run to completion yet the Büchi automaton does not know which state to transition to next and the verification has to stop. It could be a genuine timeout, a dead lock or something else, depending on what the model is about. Thus, you can check for `timeout` conditions in the model and act on accordingly.

It is important to note that statements are generally blocking statements: the process is stopped at a statement until the statement become executable. Some statements are unconditionally executable (think `printf` or `assert` and plain variable assignments) but most of them become executable only when meeting the appropriate condition (e.g. `var == true` becomes executable only when `var` is set to `true`; until this happens, the process trying to execute that statement is awaiting on the condition).

The unusual looking `if` construct is more akin to a C switch statement. It is a selection statement awaiting for statements to become executable. An `::` option sequence starts executing when its first statement (the 'guard') becomes executable. And if more than one option becomes executable, the verification will explore all these paths 'non-deterministically'. If none is executable, the selection blocks until at least one option becomes executable.

The curious `end` labels you see in the models are there to inform the verification engine that it is normal for that process to end the verification in that particular state. Otherwise, the verification will report it as an error.

Side note: do use the iSpin GUI coming with spin; or some other GUI. The insights offered into the model are really worth it.

Statechart modelling notes

There are (at least) two things to pay special attention to when modelling:

- run to completion (RTC): an event entering the statechart should be fully processed before a next event is considered for processing. This is key, as having multiple events propagating quasi-simultaneously through the statechart will end up with the wrong events processed by the wrong UML-states. The side effects of an event propagating through will start being visible as the event processing is progressing and but this is good: the statechart, supposedly, lives in a multi-threaded environment and you want to check for race conditions that the actions on enter/exit/transitions are asynchronously effecting changes on shared resources.
- transition execution order: the canonical order when transitioning from UML-state A to UML-state B is: execute on-exit actions of A then execute the actions associated with the transition itself then execute the on-entry actions of B. Some statechart implementations [QP] might have a different execution order (e.g. transition then on-exit A then on-exit B). You may want to model for the specifics of the statechart engine that the model will be implemented in.

A finite state machine (FSM)

Modelling a plain FSM should be a rather straight-forward task: one process for the whole statechart should suffice. Figure 1 (next page) is one easy FSM to model: a double-switch light: the light is on only when both a wall switch and an on-lamp switch are turned on. Listing 1 (also next page) shows the relevant parts of the model [Milente-1]. `goto` aside, it is quite readable and a rather direct translation of the UML chart.

The statechart gets its events from a model-global `_stateMachineChannel` channel. This channel needs only a one-message capacity as the statechart processes events one at a time. There is a `_stateMachineReady` flag to let the events flow into the statechart once it has reached the proper initial state. And there is an `_isLightOn` ghost variable cum-lightbulb for verification purposes.

To verify it, Promela needs a 'closed environment': that is, we need to add code to exercise the statechart – a family of test scenarios. We also need to tell Promela what the expected behavior is for the given test scenarios: an LTL formula to verify. These are in Listing 2.

The verification is only as useful as the closed environment is elaborate enough to match the reality that the statechart will be subjected to in its real implementation. And only as useful as the LTL formula expresses the

Aurelian Melinte Aurelian acquired his programming addiction in late 90s as a freshly-minted hardware engineer and is not looking for a cure. He spends most of his spare time reading and exercising. Feel free to contact him at ame01@gmx.net

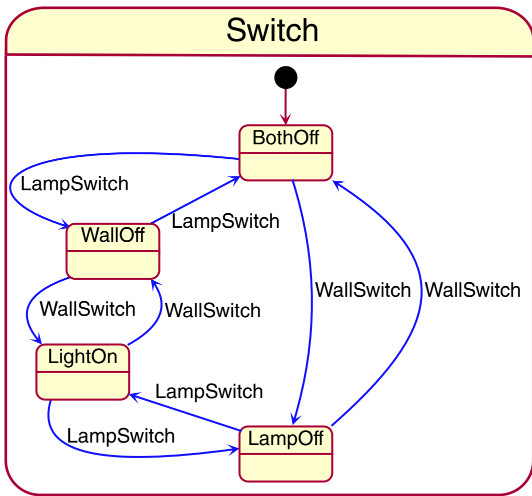

Figure 1

expected behavior. One other process will be environment injecting the events into the statechart. The `TestEnvironment` flips one switch then another to get the light on then flips both again to get back to the initial state. It should start injecting events only when the `provided` clause holds true, which happens when the statechart reached its initial state.

```
#define idx_unknown -1
#define idx_state_BothOff 0
#define idx_state_LampOff 1
#define idx_state_LightOn 2
#define idx_state_WallOff 3

mtype = { event_LampSwitch, event_WallSwitch }

typedef event {mtype evId};

chan _stateMachineChannel = [1] of {event};
bool _stateMachineReady = false;
bool _isLightOn = false;

inline send_event(evt)
{
  local event evtSend;
  evtSend.evId = evt;
  _stateMachineChannel!evtSend;
}

proctype Switch()
{
  xr _stateMachineChannel;
  local event evtRecv;
  local short currentState = idx_unknown;

/* initial state idx_state_BothOff[*/
entry_BothOff: //0
  /* execute on-entry BothOff actions if any */
  currentState = idx_state_BothOff;
  _stateMachineReady = true;
end_Switch:
body_BothOff:
  if
  :: (evtRecv.evId == event_LampSwitch) ->
    /* execute on exit state BothOff then
       transition actions*/
    goto entry_WallOff;

  :: (evtRecv.evId == event_WallSwitch) ->
    /* on exit BothOff &
       transition actions here */
    goto entry_LampOff;
  fi
/*]state idx_state_BothOff*/

/* state idx_state_LampOff[*/
...
/*]state idx_state_LampOff*/
```

Listing 1

```
/* state idx_state_LightOn[*/
entry_LightOn: //2
  _isLightOn = true;
  currentState = idx_state_LightOn;

body_LightOn:
  if
  :: (evtRecv.evId == event_LampSwitch) ->
    _isLightOn = false;
    goto entry_LampOff;

  :: (evtRecv.evId == event_WallSwitch) ->
    _isLightOn = false;
    goto entry_WallOff;
  fi
/*]state idx_state_LightOn*/

/* state idx_state_WallOff[*/
...
/*]state idx_state_WallOff*/
} // Switch
```

Listing 1 (cont'd)

Hence `TestEnvironment` offers four possible execution paths – four scenarios – to verify. In all four cases, the LTL should hold true. The LTL can be expressed with plain-English operators or more cryptic ones – both possibilities are shown here.

A hierarchical state machine

Now onto the main dish: how to model HSMs. There are at least two approaches.

Flattening

One approach could be flattening. Theoretically, any HSM can be 'flattened' into an equivalent FSM by hoisting the transitions table from a substate into the composite states owning it. Start with the lowest leaf states – flattening will eliminate these much like all four UML-states are expressed into the one FSM `Switch` above. Repeat until you reach the

```
proctype TestEnvironment() provided
  (_stateMachineReady)
{
  assert(Switch:currentState ==
    idx_state_BothOff);
  if
  :: true ->
    send_event(event_WallSwitch);
    send_event(event_LampSwitch);
  :: true ->
    send_event(event_LampSwitch);
    send_event(event_WallSwitch);
  fi
  (_isLightOn == true);
  assert(Switch:currentState ==
    idx_state_LightOn);

  if
  :: true ->
     send_event(event_LampSwitch);
     send_event(event_WallSwitch);
  :: true ->
     send_event(event_WallSwitch);
     send_event(event_LampSwitch);
  fi
  (_isLightOn == false);
  (Switch:currentState == idx_state_BothOff);
} // TestEnvironment

ltl { always/*[]*/ eventually/*<>*/
    (              (Switch:currentState ==
      idx_state_BothOff && _isLightOn == false)
    implies/*->*/ (Switch:currentState ==
      idx_state_LightOn && _isLightOn == true)
    implies/*->*/ (Switch:currentState ==
      idx_state_BothOff && _isLightOn == false)
)}
```

Listing 2

```
mtype = { event_ExitState, event_EnterState,
          event_LampSwitch, event_WallSwitch, }
typedef event {mtype evId; short toState};

chan  _stateMachineChannel = [1] of {event};
chan  _stateMachineInternalChannel = [3] of
        {event};
short _currentState = idx_unknown;
bool  _stateMachineReady = false;
bool  _isLightOn = false;

inline send_internal_event(evt, toState)
  // sorted order
{
  _stateMachineInternalChannel!evt(toState);
}
inline send_event(evt, toState)
{
  empty(_stateMachineInternalChannel);
  _stateMachineChannel!evt(toState);
}
```

Listing 3

top. If you have to do it manually (I know of no tool to do it for Promela), I doubt it can work with sizeable statecharts, given the resulting `goto` ping-pong code. Good luck keeping the Promela model true to its UML spec.

A process per UML-state

Another approach is to use one process per UML-state and a maze of channels to move events between these. Despite the growing complexity of the model, the translation of the UML spec is rather mechanical, and it keeps the transition tables local to each process-cum-UML-state. It can scale, at least as a mental effort. We can think of the double-switch as if being an HSM and model it accordingly: the switch itself can be looked at as a composite UML-state with four substates. See the relevant snippets of this model [Melinte-2] in Listing 3 (supporting code), Listing 4 (a sample state implementation out of four) and Listing 5 (switch HSM) for the model; and Listing 6 (next page) for the closed environment.

```
proctype StateLightOn(chan superChannel;
                      chan eventProcessedChan)
{
  local event evtRecv;

entry_LightOn:
  _currentState = idx_state_LightOn; //2
  _isLightOn = true;

body_LightOn:
  superChannel?evtRecv;
  // Send event to substates/regions for
  // processing (none here). If substates did not
  // processed the event: attempt to process the
  // event per our transition table below

  atomic {
    if
      :: (evtRecv.evId == event_ExitState &&
         evtRecv.toState == idx_state_LightOn) ->
      eventProcessedChan!true;
      goto exit_LightOn;

      :: (evtRecv.evId == event_LampSwitch) ->
      send_internal_event(event_EnterState,
                          idx_state_LampOff);
      eventProcessedChan!true;
      goto exit_LightOn;

      :: (evtRecv.evId == event_WallSwitch) ->
      send_internal_event(event_EnterState,
                          idx_state_WallOff);
      eventProcessedChan!true;
      goto exit_LightOn;

      :: else -> assert(false);
      skip;
    fi

    goto body_LightOn;
  } // atomic
exit_LightOn:
  _isLightOn = false;
}
```

Listing 4

```
proctype Switch(chan superChannel)
{
  local event evtRecv;
  chan substateChannel = [1] of {event};

  chan eventProcessedChannel = [0] of {bool}
  bool eventProcessed = false;

entry_Switch:
  send_internal_event(event_EnterState,
    idx_state_BothOff); // enter initial state

body_Switch:
  if
  :: nempty(_stateMachineInternalChannel) ->
     _stateMachineInternalChannel?evtRecv
  :: empty(_stateMachineInternalChannel)  ->
     end_Switch: superChannel?evtRecv;
  fi

  atomic {
    if
      :: (evtRecv.evId == event_EnterState &&
         evtRecv.toState == idx_state_BothOff) ->
      run StateBothOff(substateChannel,
                       eventProcessedChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_EnterState &&
         evtRecv.toState == idx_state_LampOff) ->
      run StateLampOff(substateChannel,
                       eventProcessedChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_EnterState &&
         evtRecv.toState == idx_state_LightOn) ->
      run StateLightOn(substateChannel,
                       eventProcessedChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_EnterState &&
         evtRecv.toState == idx_state_WallOff) ->
      run StateWallOff(substateChannel,
                       eventProcessedChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_ExitState) ->
      substateChannel!evtRecv;
      eventProcessedChannel?eventProcessed;
      goto body_Switch;

      :: else -> skip; // send to substates
                       // for processing
      fi
      // Send event to substates/regions for
      // processing
      substateChannel!evtRecv.evId(_
currentState);
      eventProcessedChannel?eventProcessed;
      if
      :: (eventProcessed == true) ->
         goto body_Switch;
      :: else -> skip; // to the transition table
                       // next
      fi
      // Attempt to process the event per our
      // transition table which is empty
      assert(false);
      goto body_Switch; // next event?
    } // atomic
exit_Switch:
} // Switch
```

Listing 5

A substate awaits events from its superstate via a dedicated **superChannel**. If the event is about entering one of its own substates, it spawns the appropriate process. If not, it first passes the event down to its own currently executing sub-substate and awaits the result of. The sub-substate reports back whether it processed or passed on the event via an **eventProcessedChannel** rendez-vous (a zero-messages capacity) channel. If the event was passed on, the substate checks its own transition table and acts on the event or passes it accordingly. Finally, it informs its superstate of the processing result in the same vein it was informed of the processing status by the sub-substate.

There are a few changes to take notice of:

- The **atomic** wrapping a bunch of statements are there to reduce the number of verification states (it works: for this particular model the reduction is about two thirds less Büchi states)
- There is now a **_stateMachineInternalChannel** This is needed to accommodate the RTC requirement: this channel will accumulate **event_EnterState** and **event_ExitState** as the HSM moves from one UML-state A to UML-state B. Manually determine the least common ancestor (LCA) UML-state of A and B and queue exit events then enter events into the internal channel as you traverse from A to B. The state machine will first process internal events (if any) then await for an external event. The capacity of the **stateMachineInternalChannel** is twice of the longest LCA path plus one for the transition itself.
- The zero-sized **eventProcessedChannel** channel that is local to each UML-state that is a composite state is a rendez-vous channel for the substates to report back to their superstate how an event that reached into that substate was processed (or passed on). The superstate will block on it until the substate is done processing, thus ensuring RTC.
- **_currentState** is now a global. **TestEnvironment** and the LTL have been changed accordingly. We avoid remote references to processes' internal variables for reasons that will get explained below.

And here is a slightly modified test scenario and LTL formula in Listing 6. Of note, the **provided** clause has been replaced with a plain statement: **(_stateMachineReady == true);**. The change is explained below.

As stated above, **TestEnvironment** and the LTL are now referencing the global **_currentState**.

```
proctype TestEnvironment()
{
   (_stateMachineReady == true);
  assert(_currentState == idx_state_BothOff);

  // unchanged code
  ...

  (_isLightOn == true);
  assert(_currentState == idx_state_LightOn);

  // unchanged code
  ...

  (_isLightOn == false);
  (_currentState == idx_state_BothOff);
} // TestEnvironment


ltl {[] <>
   (  (_currentState ==
       idx_state_BothOff && _isLightOn == false)
  -> (_currentState ==
       idx_state_LightOn && _isLightOn == true)
  -> (_currentState ==
       idx_state_BothOff && _isLightOn == false)
)}
```

Listing 6

Complexity and runtime notes

The FSM model has less than 300 states and the verification is done in a few jiffies. The HSM has less than 4000 Büchi states and the verification runs its course still as fast. The models are too small to infer anything significant but, while the statechart model is rather fixed in size, the environment counterpart can be made really complex. For instance we can add a loop to inject 10 times or 100 times more events in the statechart:

HSM Model Complexity	Performance
Base case	4000 states; runtime less than 0.01 second
10x	52k states in 0.03 seconds
100x	270k in 0.13 seconds

On my low-end machine, the verification can churn through 4.5 million states in two minutes. That seems like a huge margin allowing for very complex models to be verified in a reasonable time.

Unless the model needs fairness. Fairness requires that ready-to-run processes will not be starved: eventually such processes will get their turn. This model does not need fairness to be functional but it might happen you need it if the statechart does more than flip on/off a lightbulb boolean. Nevertheless, let's turn fairness on and have a panic moment:

HSM Model Complexity	Performance
Base case	625k states; runtime 0.3 sec
10x	4.7 million states in 50 seconds
100x	Verification might not complete in this lifetime

That makes an enormous difference. There still is a chance that you will be able to hold your breath without passing out while the verification completes: to let the verification use its partial-order reduction (POR) algorithm. POR is active by default but it can be disabled by some artefacts used in the model:

- the **provided** clause. This is a too-strong synchronization mechanism as it can suspend a process when the clause turns false. A simple statement with its built-in execution semantics is enough here to let events flow from **TestEnvironment**.
- remote variable references: **_currentState** is now a global that LTL can use instead of reaching directly into processes' internal variables. It is ugly to expose internals but ugly will save the day.
- other constructs not discussed here such as **_last**, **enabled**.
- finally, the rendez-vous channel mechanism had to be replaced. Again, the synchronization offered by rendez-vous is too strong for what we need (it changes states in two processes in one verification step). Listing 7 and Listing 8 (both on next page) are code snippets of the new model [Melinte-3] that replaces that channel with an **_eventProcessed** global. More internal details exposed.

And look at the difference in performance with POR:

HSM Model Complexity	Performance
10x	130k states in 0.1 sec
100x	655k states in 0.13 sec

More modelling

What about orthogonal regions? Regions can be modelled as processes akin to the other UML-state processes.

History states: one way to model these is to add a **_deferredEventsInternalChannel** channel to the model for transitions in/out of the history states. Events accumulated in this channel should be processed first, before the **_stateMachineInternalChannel** and the **_stateMachineChannel** channel.

Choice and junctions: see [Damjan17] for ideas.

```
short _eventProcessed = idx_unknown;
proctype StateBothOff(chan superChannel)
{
  local event evtRecv;

entry_BothOff:
  _currentState = idx_state_BothOff; //0
  _stateMachineReady = true;

body_BothOff:
end_BothOff: // valid verification's end
  superChannel?evtRecv;

  //As before

  atomic {
    if
      :: (evtRecv.evId == event_ExitState &&
        evtRecv.toState == idx_state_BothOff) ->
      _eventProcessed = idx_processed_Processed;
      goto exit_BothOff;

      :: (evtRecv.evId == event_LampSwitch) ->
      /* execute transition actions then on exit
         state BothOff; not UML-compliant */
      send_internal_event(event_EnterState,
                          idx_state_WallOff);
      _eventProcessed = idx_processed_Processed;
      goto exit_BothOff;

      :: (evtRecv.evId == event_WallSwitch) ->
      send_internal_event(event_EnterState,
        idx_state_LampOff);
      _eventProcessed = idx_processed_Processed;
      goto exit_BothOff;

      :: else -> assert(false);
      _eventProcessed =
        idx_processed_NotProcessed;
      skip;
    fi

    goto body_BothOff;
    } // atomic
exit_BothOff:
    /* execute on exit BothOff actions */
}
```

Listing 7

TLA+

The equivalent FSM model has identical logic flow barring the differences in syntax.

An equivalent HSM model will need significant adjustments because TLA cannot dynamically ‘spawn’ processes-cum-UML-states. These will have to be all created at-start and de/activated by events. But more importantly: there is no algorithm in TLA that I know of that is equivalent to POR and the verification engine is Java code: slower by an order of magnitude or two from the get-go. Complexity could kill it. ■

```
proctype Switch(chan superChannel)
{
  local event evtRecv;
  chan substateChannel = [1] of {event};

entry_Switch:
  send_internal_event(event_EnterState,
    idx_state_BothOff); // initial state

body_Switch:
  if
  :: nempty(_stateMachineInternalChannel) ->
    _stateMachineInternalChannel?evtRecv
  :: empty(_stateMachineInternalChannel)  ->
    end_Switch: superChannel?evtRecv;
  fi

  atomic {
    _eventProcessed = idx_unknown;
    if
      :: (evtRecv.evId == event_EnterState &&
        evtRecv.toState == idx_state_BothOff) ->
      run StateBothOff(substateChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_EnterState &&
        evtRecv.toState == idx_state_LampOff) ->
      run StateLampOff(substateChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_EnterState &&
        evtRecv.toState == idx_state_LightOn) ->
      run StateLightOn(substateChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_EnterState &&
        evtRecv.toState == idx_state_WallOff) ->
      run StateWallOff(substateChannel);
      goto body_Switch;

      :: (evtRecv.evId == event_ExitState) ->
      substateChannel!evtRecv;
      goto body_Switch;

      :: else -> skip; // send to substates for
                       // processing
      // goto body_Switch;
    fi

    // send event to substates/regions for
    // processing
    _eventProcessed = idx_unknown;
    substateChannel!evtRecv.evId(_currentState);
    (_eventProcessed != idx_unknown);
    if
    :: (_eventProcessed == idx_processed_
Processed) -> goto body_Switch;
    :: else -> skip; // to the transition table
                     // next
    fi

    // Attempt to process the event per our
    // transition table which is empty
    assert(false);

    goto body_Switch;
  } // atomic

exit_Switch:
} // Switch
```

Listing 8

References

[Damjan17] Panisara Damjan and Wiwat Vatanawood ‘Translating UML State Machine Diagram into Promela’ in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2017 Vol I*, IMECS 2017, March 15 - 17, 2017, Hong Kong, available at: https://www.iaeng.org/publication/IMECS2017/IMECS2017_pp512-516.pdf

[Melinte-1] Promela SM Models – switch.promela: https://github.com/melintea/upml/blob/main/doc/promela-sm-models/switch.promela

[Melinte-2] Promela SM Models – switch.hsm.promela: https://github.com/melintea/upml/blob/main/doc/promela-sm-models/switch.hsm.promela

[Melinte-3] Promela SM Models – switch.hsm.limites.promela: https://github.com/melintea/upml/blob/main/doc/promela-sm-models/switch.hsm.limits.promela

[QP] Quantum Products: https://www.state-machine.com/products

[Spin] ‘Verifying Multi-threaded Software with Spin’: https://spinroot.com/spin/whatispin.html

[Wikipedia] ‘Büchi automaton’: https://en.wikipedia.org/wiki/B%C3%BCchi_automaton

P271828R2: Adding mullptr to C++

C++ evolves via proposals, which involve a lot of hard work from all concerned. Teedy Deigh attempts to help by sharing her proposal for a new state for pointers, which may not get traction, but might make you smile.

Abstract

This proposal concerns the addition of a new feature to ISO C++. It involves a new keyword and some semantic changes. *[Question for reviewers: Is this abstract enough? Should it be vaguer? Is specifying 'ISO C++' too concrete or should I instead replace it with something like 'an existing programming language'?]*

Rambling

The menagerie of programming languages is an overcrowded dog-eat-dog, cat-eat-mouse, startup-eats-your-lunch farmyard. C++ needs to both differentiate and integrate itself. Leaving aside the obviously popular languages – C, C#, Java, JavaScript, and Python, plus a couple of languages that are too young to drink or vote – C++ also needs to compete with the likes of Haskell, a language that makes up for its lack of mainstream presence by being a Millennial influencer.

In common with many other languages, C++ has exhibited – and, indeed, acted on – its fair share of FP envy, often looking wistfully at the simplicity and feature set of Haskell and other functional programming languages. For example, compile-time computation in C++ follows functional constraints and style more closely than the less pure runtime language, especially in its original template metaprogramming form. TMP started – and, some would contend, continued – as an accident, becoming anything but temporary. Apart from the frustration induced by compiler messages and the maintenance costs arising from code subtlety and obscurity, it has no side effects. The last couple of decades have seen more concepts and, of course, **`concept`**s added to both mitigate and amplify this. Whole books and week-long training courses, for example, are now dedicated to explaining C++'s ever expanding pantheon of **`const`**-related keywords, semantics, and surprises. Creating and meeting this demand keeps the C++ market vibrant and fizzbuzzing!

There are burgeoning opportunities for C++ compile-time programming in future thanks to the increased focus on safety. Most bugs happen as a result of side effects; ergo, eliminating side effects reduces bugs. As runtime bugs are an artefact of the runtime, favouring compile-time over runtime programming seems the logical conclusion. [Aside: Deprecating the whole of runtime C++ is not the subject of this proposal. Rest assured, however, once this proposal has been accepted, I will be working on a paper to drop the runtime language. Such a possibility should, I hope, offer a clear incentive to committee members as to how they should respond to the current proposal. Rather than merely reducing the occurrence of undefined behaviour in the C++ standard and, therefore, its incidence in C++ code, removing the runtime-related parts of the standard will have the benefit of eliminating undefined behaviour in C++, as well as radically simplifying the language and library. I'm a little surprised no one has suggested this before.]

Another example of FP envy is lambdas. These were adopted into C++11, just missing the half century anniversary of their inclusion in Lisp and jumping the footgun on their 80th birthday.

One area of FP that has received at most optional attention in C++ is monads. Haskell, for example, has the **`IO`** monad to mark code with side effects – in C++, this is equivalent to pretty much any C++ – and the **`Maybe`** monad to indicate an optional value – in C++, this is similar to **`std::optional`**, but with the added elegance of Haskell and the full blessing of monadic goodness. *[Question for reviewers: Some readers of previous drafts have asked for clarification of the term 'monad', requesting a deeper explanation of the concept and its role in programming. I thought it would be enough to say that a monad is just a monoid in the category of endofunctors, but have been told that is neither sufficient nor necessary. Suggestions?]*

Meandering

This proposal concerns itself with the issue of pointers and safety. Null has been called a billion-dollar mistake. While this pales in comparison to the **`int`**-busting cost of Crowdstrike's unchecked off-by-one error, the issue still warrants addressing. Similarly, one attempt to reduce undefined behaviour in the language – for example, uninitialised pointers – has been to rebrand and remarket much 'undefined behaviour' as 'erroneous behaviour' and hope people buy it. I believe a less cynical and less critical path can be taken.

This proposal pursues – and pounces on – a different approach. In addition to pointing to valid memory or to null, a pointer can be in the mull state. A mull pointer reflects an uncertainty and lack of commitment that **`Maybe`** suggests, but without the intellectuality and clarity of monads. Squinted at just right, the mull state can be considered the offspring of – or head-on collision between – JavaScript's **`undefined`** and IEEE 754's NaN.

Whether dereferencing a mull pointer works or not is largely a matter of consideration. What, after all, do we mean by 'works'? Who are we to say whether a piece of code is 'correct' or not? Is it not presumptuous for us to judge? There are many opportunities for interpretation here that compiler implementors may wish to ponder. Mulling implies that the program could hang indefinitely. Or perhaps it captures the spirit of the conversation between developers collectively CSI-ing a core dump, thus encouraging software development to be a more social activity. Either way, it is left to the implementor's discretion rather than being left undefined.

As you can see, such a contemplative and reflective approach [Aside: There has to date been no discussion on how – or even whether – this should be integrated with reflection features and proposals.] sidesteps and deftly dodges questions of safety. It also takes the edge off the judgemental negativity of 'erroneous behaviour' and the formal snubbery of 'undefined behaviour'.

Teedy Deigh Like other members of Generation X, C++ isn't showing any signs of going away. Teedy has, therefore, decided to help it be its best self. Bringing about such change typically demands deep knowledge of the language, awareness of the standardisation process, political acumen, sensitivity to other people's opinions and taste in matters of programming and design. Teedy, however, rarely submits to the demands and expectations of others.

Heavy users of floating-point numbers will surely be excited to have a new non-finite state that is beyond compare

It is easy to see, with a little overthinking and overdesign, that introducing the mull state into C++ offers many possibilities for future generalisation. For example, in addition to truth and falsehood, `bool` could have a third mull state, allowing C++ to better model indecision, three-valued logic systems, and political discourse. For Unicode characters and strings the mull state could be mapped to the shrug emoji. Heavy users of floating-point numbers will surely be excited to have a new non-finite state that is beyond compare. However, much as premature generalisation is a favourite pastime of C++ developers, I will resist the temptation to dive into that rabbit warren here. *[Question for reviewers: That said, I do have a fairly fully worked out preliminary draft of all these possibilities.*[Aside: And, indeed, may have got slightly distracted working on that rather than this proposal, so apologies for the delay.] *Please let me know if you would like to see it. Also, as the current paper concerns pointers, there is no reference section.]*

Concrete

The `mullptr` keyword is a constant of the mull state that is implicitly convertible to any pointer type, dumb or smart. Whether dereferencing a mull pointer results in unspecified or implementation-defined behaviour is not defined.

The declared type of `mullptr` is `mull_t`, which can be picked up as an `std` (either namespace or module). No decision has yet been taken as to what header `mull_t` should be defined in, but in keeping with existing practice it will either be something quite obvious (e.g., **<mull_t>** or **<mullptr>**) or somewhat surprising (e.g., **<cstdint>** or **<any>**). It is intended that `mull_t` is to be pronounced *mullet* rather than *mult* or *multi*.

Handwavium

It is customary to eventually include proposed wording changes for the standard. This would be premature at this stage, but readers can be assured that there will be words. [Aside: Especially if I have to go another round with committee members who have suggested this feature is 'frivolous', 'poorly thought out', and 'a waste of valuable committee time'.] ■

For peer support, join
ACCU
From £35 a year.
That's less than £3 a month.
You'd spend more on a cup of coffee.
Visit accu.org for details.

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

To connect with like-minded people visit accu.org

accu