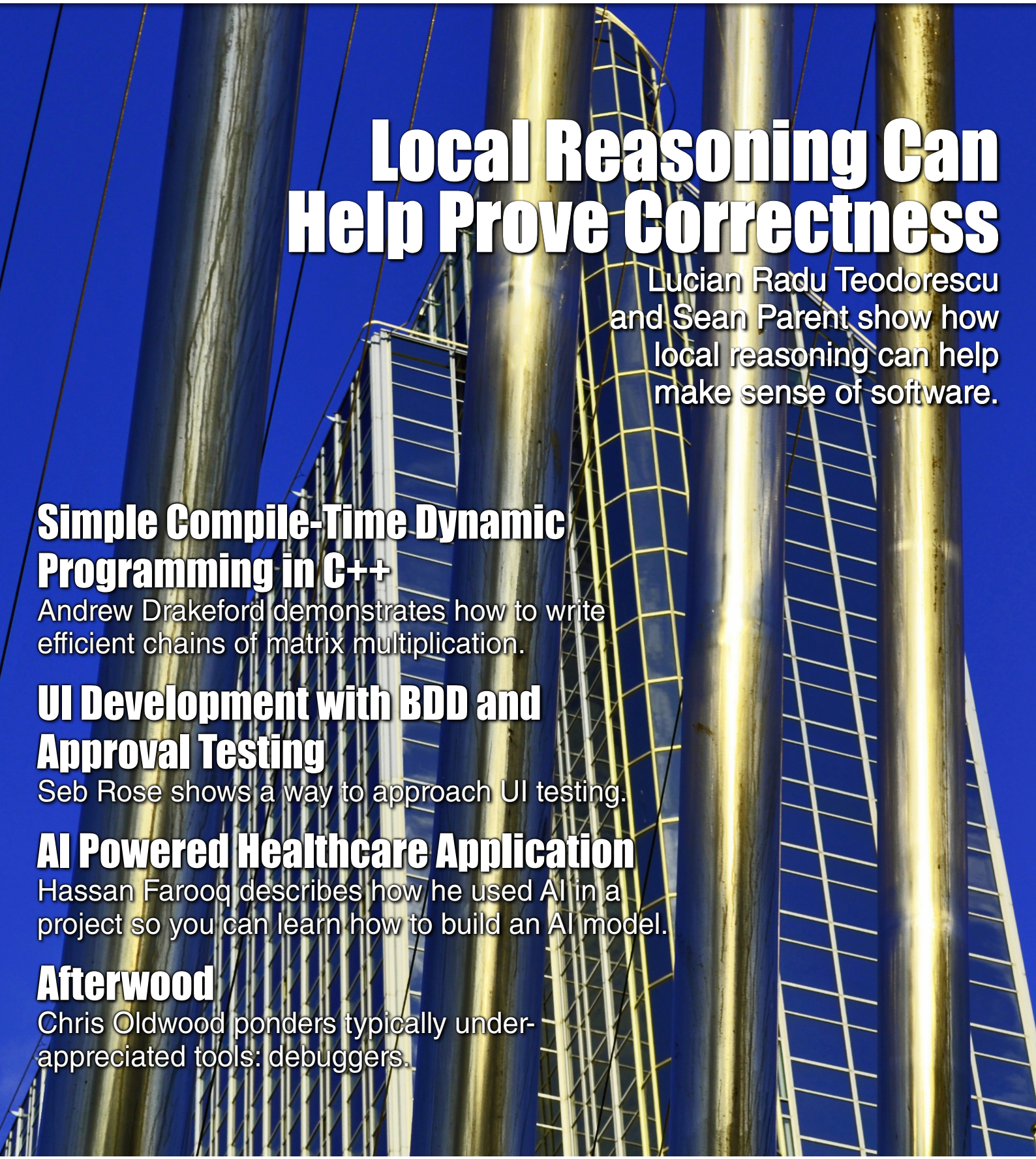
overload 188

AUGUST 2025 £4.50



Local Reasoning Can Help Prove Correctness

Lucian Radu Teodorescu and Sean Parent show how local reasoning can help make sense of software.

Simple Compile-Time Dynamic Programming in C++

Andrew Drakeford demonstrates how to write efficient chains of matrix multiplication.

UI Development with BDD and Approval Testing

Seb Rose shows a way to approach UI testing.

AI Powered Healthcare Application

Hassan Farooq describes how he used AI in a project so you can learn how to build an AI model.

Afterwood

Chris Oldwood ponders typically under-appreciated tools: debuggers.

A magazine of ACCU

ISSN: 1354-3172

Join ACCU

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

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

Join now!
Visit the website

professionalism in programming

www.accu.org

August 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 Kevlin Henney.

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

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

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

Copy deadlines
All articles intended for publication in *Overload* 189 should be submitted by 1st September 2025 and those for *Overload* 190 by 1st November 2025.

What's THIS for?

Sometimes we ignore details to get a task done. Frances Buontempo considers how much more there is to programming than just code.

Overload reserves two pages for an editorial, and I somehow manage to fill them, even though I can never think of a suitable topic. What are the two pages for? An opinion piece? Something topical? Maybe I'm just being a coward and avoiding controversial subjects. Or maybe it is really hard to write to order on a regular basis. I have got away with this for a long time, though. I become editor in 2012. My first attempt at an editorial was entitled 'Allow Me To Introduce Myself' [Buontempo12]. I mentioned then it was hard to think of a topic, or what to do with the two pages. Nothings has changed.

I'm sure we all have examples of being given something we couldn't figure out how to use, though maybe not two pages for an editorial. I was given a stationery tray when I started a previous job, including pens, notepads and a bottle of Tipp-ex. I'm not sure the Tipp-ex would have been that useful for correcting incorrect code. I also suspect some readers may never have seen a bottle of 'correction fluid'. Arcana slips into various places in IT. The save icon is still often a floppy disc. Some forms still have a space for a fax number. Technology changes are very fast paced, so 'old' tech might mean something from a few years or decades ago. How many SCART cables do you have stored in a cupboard? In other disciplines, change is slower – so the almost immediate obsolescence doesn't take hold in the same way. Historical reasons are still permeate, but in a different way. A plumber no longer works with lead, though the name has stuck: plumbum means lead, and a plumbarius works with lead [etymonline]. I'm sure you can think of other examples.

Historical artefacts are one thing, but often there are more things lurking than you first realise, no matter what you are trying to do. When you try to write code, you often find you need to learn extra concepts and idioms. Just copying code from the internet or using GenAI might get you somewhere, but at some point you need to engage your brain and think about what you are doing. Programming jobs themselves can come with unexpected aspects, like FORTRAN lurking in the stack call, or finding yourself on-call. Software itself often has surprising elements which can be hard to figure out. You will either need to learn how to drive an IDE or an editor and a shell or prompt. You might accidentally learn how to type a bit. You might end up arguing about whether mechanical keyboards are the best, or which editor to use. You might have to learn bug tracking software, or produce a Gantt chart for your final project. (Do people still make Gantt charts – yes they do!). You might have to learn how to operator a high tech telephone, or discover how to press + on a numeric keypad. None of us signed up for this!

Software brings other surprises too. Moving from working on a project on your own in a small group as a student to finding your way around a large software system in your first job can be overwhelming. Code tends to accrete, erm, 'unnecessary' functions and the like over time. There are several ways to start finding your way around an unfamiliar code base. If you can't immediately see what a function or class is for, try deleting it and see if you get compiler errors. That's easier with C++ (at the moment) – I tried this on a C# code base, and various parts were only used via reflection, so that was confusing. Michael Feathers wrote about scratch refactoring in his *Working Effectively with Legacy Code* book [Feathers04]. You deliberately refactor code, then throw away your changes, allowing you to discover how the parts hold together. Rolling your sleeves up and tinkering is often a good way to think systems through and learn. You are unlikely to figure out how something works by listening to someone else tell you about it. I have had several contracts where a dev lead took new hires to a room and told them about the code, pressing short-cut keys for 'go to definition' or 'go to usage' in an IDE. Most of us can manage to press a key ourselves. The best onboarding came from the tech lead sitting with me and us refactoring a small function. Doing something together can be so much more informative that talking about it.

Each programming language tends to end up too big to hold fully in your head. There are many features in C++ I haven't used often. I was asked when I had used `dynamic_cast` at an interview once. My honest answer, at the time, was I hadn't. The interviewer was surprised, but talking it through we decided you could often end up with a cleaner design if you avoid dynamic casts. In a different interview I was asked if I had used custom allocators to speed up code, and I hadn't. I still haven't used them often, but have seen them speed up code. I'd need to look up the details to jog my memory if I wanted to use them again. Having an awareness of something means you can just about answer the question, "What's this for?" However, that is not the same as being able to use the feature easily.

Switching between programing languages can be confusing too. You might have a mental model of how one thing works, and make assumptions about similarities when you find something similar in another language. C++ classes have an implicit `this` pointer to refer to the current instance. What's `this` for? Many times you don't need to use it directly, but implementing copy or move assignments will force your hand. Python, in contrast has an explicit parameter for instance methods, usually called `self` by convention. Each instance method takes a first parameter referring to the current object. The `self` isn't a keyword, but most people use this. The subtle difference between keyword or not and implicit versus explicit isn't too confusing, but might catch you out for a moment if you switch between the languages a few times. C++23 introduced *deduce this*, which is a newer C++ feature I have never used [Brand22]. Sy Brand refers to these as "explicit object parameters": you add a function starting with a `this` parameter, which gives you a way to write one function for const/non-const, and rlavues/lvalues, rather than

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.

needing four overloads. Maybe I shouldn't get started on JavaScript's `this` binding quirks. Strict mode makes `this` switch from some global object to undefined [Stackoverflow]. Also arrow functions (think lambdas) don't have a `this` themselves but might find one in lexical scope [Mozilla]. Naming between programming languages differs too: list or array or vector? It depends.

Aside from language quirks, trying to build code can be a challenge. If you're lucky, there's a CI/CD pipeline with a script that just works. However, that's often not the case. And you might need to wait for a tool licence, or fill in forms to install a package. There's more. C++ has several rival package managers, and modules promise ways to speed up builds. Write up an article about either if you are using them successfully: some people are, but I'm just watching from the sidelines for now.

Some people exclusively use an IDE and don't know the details of how a build works. Which is fine, to a point. I recently read a reddit post [Reddit] from someone saying they are a hobby programmer and know how to drive Visual Studio but don't know where to start on other platforms. I started my programming career with embedded coding, so learnt various build tools from the start. I personally like to find out how to build something from a prompt, just to get a feel for what's happening. I should probably learn CMake properly one day, but I am just watching that from the sidelines too. IDEs can help you be more productive, but it's useful to understand a bit of the details. And sometimes upgrades move your IDE's button, which is very irritating. Or sprout new buttons – leaving me wondering what the new buttons are for. I notice my Visual Studio upgrade is offering me GitHub Copilot. I shall avoid this for now.

Different programming languages and varying build systems may leave you wondering what various parts are for. Things can get even more complicated if you also have feature flags. You have probably heard of the Knight Capital Group feature flag fiasco [Wikipedia-1]. Reusing an old feature flag without fully understanding what it did, or where it might be loitering in old code that hadn't been upgraded, led to an expensive round of buying stock by mistake. The moral of that story might be remove feature flags as soon as possible. Such flags are one type of global state that makes code hard to reason about. Feature flags aren't the only way to break code. Locales can cause confusion too. If your build server has different setting to your machine, you might see flaky tests – ones that pass with one locale but not another. Global state can also lead to race conditions in parallel code and further flaky tests that sometimes pass, but only sometimes, on the same machine. If you find a flaky test, first ask if you need it, and if you do try figure out why it sometimes fails. If you leave such a test in a codebase, it will get ignored. Try to nudge it to something reliable. I traced some confusing behavior to a Singleton once, another form of global state. By adding a `reset` method, calls to tests weren't affected by the run order. You can sometimes find a small change that makes your life easier.

The whole ecosystem of programming involves a lot more than just the code. Coding guidelines vary between companies or even teams, so you might need to be flexible if, like me, you often contract. Build systems vary wildly too. Even if you are settled in one role for the long term, or have full control over your setup because you are a hobby programmer, tools change and the languages evolve. I suspect as programmers, we are frequently faced with change in ways that some other professions or hobbies are not. Maybe this keeps our brains more plastic: I'm no brain surgeon or psychologist, but neuroplasticity seems to be important for cognitive function [Wikipedia-2]. Adapt and survive, so the saying goes. Trying to keep up to date is a challenge. C++, and any programming language, will always contains features and aspects you don't fully understand yet. Feel free to ask "What's this for?" The accu-general email list is a friendly place to ask questions. You might even inspire someone to write an article for us. You will always have things you don't fully understand. That's OK. You could delegate to someone else, pair with them, or ask AI. But, whatever you do, stay curious and keep on learning.

References

[Brand22] Sy Brand, 'C++23's Deducing this: what it is, why it is, how to use it', June 2022, https://devblogs.microsoft.com/cppblog/cpp23-deducing-this/

[Buontempo12] Fran Buontempo, 'Let Me Introduce Myself' in *Overload*, 20(110):2-3, August 2012, https://accu.org/journals/overload/20/110/buontempo_1904/

[etymonline] Plumber: https://www.etymonline.com/word/plumber

[Feathers04] Michael Feathers, *Working Effectively with Legacy Code*, ISBN 10: 0131177052 / ISBN 13: 9780131177055 Published by Pearson, 2004

[Mozilla] Arrow functions: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[Reddit] 'I use Visual Studio to write C++ and nothing else. I have no idea what command lines, CMake, or any of that stuff is - where can I find information on how to move forward?': https://www.reddit.com/r/cpp_questions/comments/1lov0in/i_use_visual_studio_to_write_c_and_nothing_else_i/

[Stackoverflow] 'Default binding of the this keyword in strict mode', available at http://stackoverflow.com/questions/49023201/default-binding-of-the-this-keyword-in-strict-mode

[Wikipedia-1] Knight Capital Group: https://en.wikipedia.org/wiki/Knight_Capital_Group

[Wikipedia-2] Neuroplasticity: https://en.wikipedia.org/wiki/Neuroplasticity

Local Reasoning Can Help Prove Correctness

Making sense of software is challenging. Lucian Radu Teodorescu and Sean Parent show how local reasoning can help.

Programs are becoming increasingly complex, while our mental capacities remain constant. We are well past the point where a single person can understand the entirety of an average-sized software system. In this context, it is worth revisiting what Dijkstra called "mental aids" [Dahl72].

One of the most valuable techniques that support our ability to reason about software is *local reasoning*. While the term itself does not appear explicitly in Dijkstra's writings, it is almost as though he had the concept in mind when contributing to *Structured Programming* [Dahl72]. Hoare also explored the idea of reasoning about the correctness of programs by analysing small code blocks [Hoare69], which laid the conceptual groundwork for local reasoning. However, the term *local reasoning* first appeared in the literature in 2001, in the article 'Local Reasoning about Programs that Alter Data Structures' [O'Hearn01].

Local reasoning refers to the ability to analyse and verify a defined unit of code in isolation – without needing to understand all the contexts in which it is used, or the details of the code it depends on. By "unit of code", we typically mean functions or classes (though the concept can be extended to other organisational constructs, such as namespaces). These units must have well-defined APIs that separate the code into client-side usage (e.g., calling a function or instantiating a class) and implementation-side logic (e.g., the internal details of the function or class). Furthermore, the implementation-side logic can itself be subdivided into the focal unit of code and the dependencies it relies upon.

By applying local reasoning, one can more easily understand code, as it separates the unit being analysed from its clients and its dependencies. Having code segments that can be reasoned about independently provides an excellent mental aid, allowing us to incrementally fit code into our limited cognitive resources.

The aim of this article is to show that local reasoning can also support *proving* the correctness of code. This is significant because, unlike safety, correctness does not compose. Let us explain. If two components, A and B, are *safe*, then composing them preserves safety. However, if A and B are *correct*, we cannot guarantee that combining them will result in a correct program. For example, suppose A produces measurements in imperial units, and B consumes values assuming they are in metric units. Individually, A and B may be correct, but used together, they yield incorrect behaviour. This exact type of mismatch contributed to the crash of NASA's Mars Climate Orbiter [Wikipedia].

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

Sean Parent is a senior principal scientist and software architect in Adobe's Software Technology Lab. Sean joined Adobe in 1993, working on Photoshop, and is one of the creators of Photoshop Web, Photoshop Mobile, Lightroom Mobile, and Lightroom Web. In 2009, Sean spent a year at Google working on Chrome OS before returning to Adobe. From 1988 to 1993, Sean worked at Apple, where he was part of the system software team that developed the technologies enabling Apple's successful transition to PowerPC.

Properties

Let us begin with the notion of *correctness*. Following Leslie Lamport [Lamport77], we define correctness as the combination of **liveness** and **safety** properties that describe the behaviour of a program or specification. Liveness properties assert that something desirable eventually happens; safety properties assert that something undesirable never happens. For example, in a program that sorts a list of words, a liveness property might be that the output is a permutation of the input sequence, while a safety property might be that the program does not exhibit undefined behaviour.

To use the distinction introduced by Fred Brooks in his well-known 'No Silver Bullet' article [Brooks95] – a distinction inspired by Aristotelian terminology – we can divide properties into *essential* and *accidental*. In the sorting example, the requirement that the output be a permutation of the input is an essential property: we cannot imagine the program being correct without it. In contrast, accidental properties include the ordering of equivalent elements in a sorted sequence, the number of CPU instructions executed for a given input, the heat generated on a specific machine, or the amount of memory used.

Naturally, when we discuss correctness, we are concerned only with the essential properties; accidental ones can typically be ignored. For the sorting example, two different algorithms may vary in their accidental properties, but they share the same essential properties.

Two programs (or specifications) are considered *equivalent* whenever they share the same set of essential properties.

When dealing with an entire program specification, the set of essential properties can be interpreted as the program's **requirements**. These requirements come in two forms: **explicit** and **implicit**. Explicit requirements are those that are clearly stated or documented. Implicit requirements are not usually written down and are often overlooked, yet we still recognise them as essential when asked. For instance, "the program shall not deadlock" is a typical implicit requirement. Other examples include: "the program shall not exceed the system's memory", "the program shall complete in under one second when sorting 1,000 words", or "the output file format shall match the input file format".

A program is typically composed of multiple parts, ideally structured in some form of hierarchy [Dahl72]. In our simple example, there may be one part that reads the input file, another that writes the output, one that performs the sorting, and a smaller component that compares two words (possibly alongside others). The global properties of the full program may not be directly attributable to individual components. However, just as we decompose programs into smaller parts, we assume there exists a corresponding method to decompose properties into smaller ones applicable to each part. For example, from the program's specification, we could derive what it means for two words to be considered equal or for one to be 'less than' another – key requirements for the comparison component.

To summarise, we define correctness as the set of *essential properties* that apply to both programs and specifications, at both the level of the entire system and its constituent parts. The essential properties of the full specification correspond directly to the program's requirements.

Abstractions

Functions and classes are abstractions. To use Dijkstra's words, an abstraction is something for which we can describe "what it does" while completely disregarding "how it works" [Dahl72]. As mentioned earlier, abstractions define an API, and this API separates the client-side logic from the implementation-side logic. The implementation logic can itself be further divided into the core implementation and the logic it depends upon.

The API of an abstraction typically consists of:

- **Type information** (e.g., function declaration, class definition)
- **Contract specifications**: preconditions and postconditions
- **Semantic properties** (often documented as comments, or occasionally expressed in the name of the abstraction)
- **Implicit global assumptions** (e.g., non-aliasing of function parameters, object lifetime guarantees, etc.)

We refer to the *abstraction API properties* as the set of essential properties derived from the abstraction's API.

We say that an abstraction is *well defined* if its abstraction API properties are equivalent to the essential properties that can be derived from its implementation – considering both the core logic and its dependencies. While it is possible to have correct programs even if some abstractions are not well defined, we still strive to ensure all abstractions are well defined, as this simplifies reasoning about the program.

Listing 1 provides a straightforward example of a well-defined abstraction. In contrast, Listing 2 illustrates two examples of abstractions that are *not* well defined:

- `my_sort()` fails to perform sorting for a vector with exactly two elements,
- `my_sort2()` exhibits undefined behaviour if the input vector contains fewer than two elements.

Programs are typically organised hierarchically – more precisely, as *directed acyclic graphs* (DAGs). The implementation of one abstraction may rely on other abstractions. If we take functions as primary examples of such abstractions, we often encounter functions that call other functions.

For a given abstraction, we refer to the abstractions it directly uses in its implementation as its *child abstractions*. In other words, the child abstractions are those for which the initial abstraction is a direct client. Using this notion, we can navigate the hierarchical structure of programs more effectively.

We define the *local view* of an abstraction as the program or specification obtained by replacing, within its implementation, all child abstractions with program fragments or specifications that are **equivalent** to their APIs. For example, if a function `my_sort()` calls `std::sort()`, the local view of the `my_sort()` abstraction is the program obtained by replacing the call to `std::sort()` with an abstract representation conforming to the API of `std::sort()`, discarding its internal implementation details.

By using local views, we effectively ignore the implementations of child abstractions – i.e. the dependencies of an abstraction's implementation – and focus solely on the core implementation of the abstraction, assuming that all dependencies behave as specified.

```
//! Adds '2' to 'x'.
//Precondition: x + 2 < INT_MAX
int add_two(int x) {
  return x + 2;
}
```

Listing 1

```
//! Sorts inplace `v`.
void my_sort(vector<int>& v) {
  if (v.size() > 2)
    std::sort(v.begin(), v.end());
}

//! Sorts inplace `v`.
void my_sort2(vector<int>& v) {
  std::sort(v.begin(), v.end());
  if (v[0] > v[1]) std::terminate();
}
```

Listing 2

The notion of equivalence in the previous definition is essential. The theorem we will introduce later relies critically on this equivalence. When examining the local view of an abstraction, we are effectively assuming that all of its child abstractions are *perfectly implemented*. For example, when reasoning about a function, we assume that all the functions it calls behave according to their specified contracts – so we do not need to inspect their implementations directly.

It is worth noting that there may be multiple valid substitutions for each child abstraction when constructing the local view. However, since all these substitutions are equivalent with respect to *essential properties*, the resulting local views are also equivalent.

Now that we have defined what a *local view* is, we can define the *local properties* of an abstraction as the set consisting of:

- Properties derived from the type information of the abstraction
- Properties derived from the preconditions of the abstraction
- Implicit global assumptions
- Essential properties of the abstraction's local view, assuming the above as preconditions

When comparing the local properties of an abstraction with its API properties, two key differences arise. First, local properties do *not* include the postconditions or the semantic properties defined by the API. Second, they *do* include all properties that can be inferred from the local view and its preconditions.

Whereas API properties allow us to decouple an abstraction's implementation from its clients, *local properties* decouple the abstraction's local view from the dependencies of its implementation. The local properties capture all assertions that can be made by applying Hoare logic [Hoare69] to the abstraction's local view.

We say that an abstraction is *locally well defined* if its local properties form a **superset** of its API properties. In other words, all the postconditions and semantic properties defined by the abstraction's API can be derived by analysing its immediate implementation.

In this definition, we allow the set of local properties to be *larger* than the set of API properties, rather than requiring them to be equivalent. This choice is intentional. Applying Hoare logic to the abstraction's implementation may yield additional properties, not all of which are relevant at the abstraction level. For instance, consider a sorting routine that always handles more than 10 elements. Its API may specify: "the routine shall not terminate if the input has more than 10 elements". However, reasoning about its implementation may yield the stronger statement: "the routine will never terminate". The latter includes the former.

Listing 3 (next page) presents two examples of functions that are *not* locally well defined.

- The function `times_two()` is not locally well defined because its local view implies that the function returns the input plus `2`, while its semantic intent (as suggested by its name) is to multiply the input by `2`.
- In the second case, `count_primes_below()` is not locally well defined because reasoning over its local view does not yield the required property. The problem lies in its dependency

```
// precondition: abs(x) < 1000
int times_two(x) { return x + 2; }

bool test(int n);

// precondition: 2 < n < 2^16.
int count_primes_below(int n) {
  int result = 0;
  for (int i=1; i<n; i++) {
    if (test(i)) result++;
  }
  return result;
}
```

Listing 3

on a function named `test()`, whose API is underspecified. There is nothing stating that `test()` must return true when the input is a prime number. Without that assumption, we must treat `test()` as returning arbitrary values. Consequently, the body of `count_primes_below()` cannot be shown to produce the property that it counts the number of primes below the given input – a property implied by the function's name.

There are also cases where a function is (provably) *well defined*, but not *locally* well defined. Listing 4 illustrates such a situation. The function `add_three()` behaves correctly: it adds `3` to its input. However, it does so by relying on a helper function `add_one()`, whose API does not match its actual implementation. For `add_three()` to be locally well defined, we must either correct the API of `add_one()` or adjust the implementation of both functions to be consistent with their declared behaviour.

Local reasoning

We say that an abstraction has *local reasoning* if it is *locally well defined*. In other words, we can reason about the correctness of the abstraction by focusing solely on its core implementation – without inspecting the implementations it depends on.

Lemma. If an abstraction A has local reasoning and all its child abstractions are well defined, then A is also well defined.

We provide a schematic proof by contradiction.

Assume that there exists an essential property P which belongs to the API properties of A, but does *not* hold for the implementation of A. From the perspective of Hoare logic, there must exist an instruction Q in the implementation of A which is expected to contribute (directly or indirectly) to ensuring P, but fails to do so. That is, a fault at Q explains the discrepancy between the specification and the implementation.

Now, Q must either belong to the core implementation of A, or to its dependent logic. If Q is in the dependent logic, then the failure lies in one of the child abstractions. But by assumption, all child abstractions are well defined – so this case is impossible.

Therefore, Q must be in the core implementation of A. This implies that P cannot be among the local properties of A, since local properties are derived by applying Hoare logic to the core implementation (including Q). But if A has local reasoning, then its local properties are equivalent to its API properties, and thus P should not appear in the API properties either. This is a contradiction.

Hence, if A has local reasoning and all of its child abstractions are well defined, then A is also well defined.

To express the next corollary more conveniently, let us denote by *tr(A)* the *transitive closure* of the child abstractions of A, including A itself.

```
// precondition: abs(x) < 1000
int add_one(int x) { return x + 2; }
// precondition: abs(x) < 1000
int add_three(int x) { return 1 + add_one(x); }
```

Listing 4

```
// Postcondition: returns 2
int f();
// Postcondition: returns 2
int g();

int f() { return g(); }
int g() { return f(); }
```

Listing 5

Corollary. For a given abstraction A, if all abstractions in *tr(A)* have local reasoning, and *tr(A)* forms a DAG (directed acyclic graph), then A is well defined.

Since the abstractions in *tr(A)* form a DAG, we can arrange them in a sequence such that, for any abstraction A_0, all its child abstractions appear earlier in the sequence (i.e. to the left of A_0). By traversing this sequence from left to right, we can iteratively prove that each abstraction in the sequence is well defined.

For any element A_0 in the sequence, its child abstractions appear earlier and have therefore already been shown to be well defined. Given that A_0 has local reasoning and all its child abstractions are well defined, we can apply the lemma to deduce that A_0 is also well defined.

Thus, by proceeding through the sequence from left to right, we prove that all elements are well defined abstractions. Since abstraction A appears as the final element in the sequence (as per our construction), we conclude that A is well defined.

Hence, the corollary holds.

The reader should note that the corollary does not apply if the abstractions do not form a DAG. Listing 5 presents an example of two mutually dependent functions that are not well defined. According to their API properties, each function is expected to return the value 2, but in practice, they call each other endlessly, resulting in non-termination.

Now, let us turn our attention to the entire program. We assume the program can be represented by a *top-level abstraction*. In terms of functions, this typically corresponds to the `main()` function. We also assume that the abstractions within the program can be organised hierarchically as a DAG.

Theorem. If all abstractions in a program support local reasoning, and the program is composed of hierarchically organised abstractions (i.e. forms a DAG), then the entire program is correct.

The proof follows directly from the corollary above. Given the existence of a top-level abstraction for the program, and the assumption that the abstraction hierarchy forms a DAG, we can apply the corollary to conclude that the top-level abstraction is well defined. That is, its API properties match the essential properties derivable from its implementation.

By our definition of correctness – as the equivalence between API-level properties (requirements) and implementation-level properties – we conclude that the program is correct.

Thus, we arrive at the desired result.

Discussions

Local reasoning is good

The average programmer spends significantly more time reading code than writing it – by a factor of more than 10×, according to Robert C. Martin [Martin09]. *Local reasoning* is one of the 'mental aids' (to borrow Dijkstra's terminology) that helps us read code more effectively, and more broadly, to reason about it.

Much of this article has focused on showing how local reasoning supports *proving* that code is correct. Turning the problem around, local reasoning can also be invaluable for *identifying* issues. It is far easier to spot problems when one can analyse a single function or class at a time, checking whether it conforms to its specification, without constantly examining

its dependent functions. In a way that parallels our earlier corollary, the process of issue detection becomes *linear* rather than *exponential*.

Understanding code also benefits greatly from local reasoning. Rather than attempting to fit an entire program into one's mental model, a developer can focus on understanding smaller pieces of code, and how those pieces compose into larger abstractions. To draw an analogy: imagine trying to make sense of a Wikipedia article. Like local reasoning, the article should include sufficient context to be readable in isolation. If understanding it required recursively drilling down into every linked page, reading the article would become a Sisyphean task – well beyond our cognitive capacity.

Local reasoning also offers a secondary benefit in tooling: most static analysis tools benefit heavily from it. Code that is easier for humans to understand and reason about is also easier for machines to analyse effectively.

Organizing programs hierarchically

One of the widely applied design principles in software engineering is the *Acyclic Dependencies Principle*, which states that "the dependency graph of packages or components should have no cycles" [Martin97]. Cyclic dependencies in an application typically lead to tight coupling, hinder reuse, and cause *domino effects* – where a small change in one module propagates through others in unintended ways.

This article offers an additional perspective on why cyclic dependencies are problematic. Cyclic dependencies can make it significantly harder to reason about correctness when applying local reasoning. Although this may not be the primary reason to avoid cyclic dependencies, it is a consideration worth pondering.

Reasonable software and global reasoning

A well-known maxim in our industry is that *we should write code for humans, not for machines*. In other words, we should strive to produce *reasonable software*. We define reasonable software as software that can be easily reasoned about by people, and as software that is decent and fair – meaning that it avoids unexpected surprises.

Local reasoning is perhaps the most effective way to achieve reasonable software. At the same time, it can be viewed as just one aspect of what reasonable software entails. In practice, we also need some form of *global reasoning*: that is, we expect certain properties to hold across multiple abstractions of a program – albeit with occasional exceptions.

A good example of such a global property is the *applicability of local reasoning itself*. Local reasoning is of limited value unless it is applied consistently: if one function supports local reasoning but calls other functions that do not adhere to their contracts, reasoning about correctness quickly becomes ineffective.

Other examples of global properties that contribute to reasonable software include:

- Abstractions are organised hierarchically (no cycles)
- The program is well structured
- The *Law of Exclusivity* [McCall17] is applied consistently
- The program exhibits no undefined behaviour
- Consistent conventions for naming and documentation are followed
- Implicit assumptions across the program are clear and respected

From requirements to abstraction properties

One weakness of the approach presented in this article is the assumption that we can readily translate complete requirements – both explicit and implicit – into properties associated with the top-level abstraction (e.g., the `main()` function), and then systematically distribute those properties to child abstractions. In practice, this rarely occurs in a disciplined or complete manner.

Explicit requirements are typically the *known knowns* of a project. Implicit requirements are often the *unknown knowns*. Moreover, as Kevlin Henney has frequently pointed out, most software projects are also plagued by *unknown unknowns* [Henney21]. The unfortunate reality is that we seldom know the full set of requirements. As a result, it becomes difficult to ensure that all essential properties are properly assigned across abstractions. This makes the top-down application of the approach described here hard to carry out rigorously.

However, this is only partially bad news. In practice, having good abstractions – that is, abstractions with clearly defined and trustworthy contracts – greatly aids in achieving overall correctness.

In contrast to the top-down approach of distributing requirements across abstractions, starting from solid abstraction contracts enables a complementary bottom-up strategy. By applying local reasoning and examining each abstraction individually, we can incrementally infer the properties of the entire program. And even if we do not know all the requirements in advance, we can still verify that the inferred properties align with our expectations of the program's behaviour.

There is another important way in which the bottom-up strategy proves valuable: by applying local reasoning to individual components, we can improve their correctness in isolation. This, in turn, enhances our ability to reason about the system as a whole. Dave Abrahams and Sean Parent refer to this process as "building islands of correctness". It can be applied not only to newly developed software but also to existing systems.

Improving local reasoning

There is relatively little literature dedicated to local reasoning. The original article that introduced the term [O'Hearn01] does not offer practical guidance on how to create or improve local reasoning in a program. More recently, the works of Dave Abrahams, Dimi Racordon, and Sean Parent have explored local reasoning as a useful, applicable technique in real-world software [Racordon22a, Abrahams22a, Abrahams22b, Abrahams22c, Racordon22b, Parent23, Parent24].

Local reasoning can be severely impeded by reference semantics in the presence of mutation. If a function holds a reference to a memory location, it becomes difficult to reason locally about that function if other functions can also access and mutate the same memory. For example, the code in Listing 6 may appear correct to the untrained eye – but a call such as `add_twice(x, x)` produces unintuitive results. If `x == 2`, then `x` becomes `8` instead of `6`, because the value of `y` changes between the first and second additions. This is counter-intuitive: nothing in the body of `add_twice` suggests that such behaviour is even possible.

While the example in Listing 6 is intentionally constructed to demonstrate this point, real-world code often contains many such opportunities for *spooky action at a distance*. This can happen any time a function takes parameters by reference, uses global variables, or accesses shared ownership (e.g. via shared pointers). In all of these cases, unexpected side effects may occur, making local reasoning much harder.

The most effective mitigation is to enforce the *law of exclusivity* [McCall17], which states that if a piece of code holds a mutable reference to an object, no other references to that object may exist concurrently. Conversely, multiple references may coexist as long as they are all read-only. Languages such as Rust, Swift, and Hylo enforce this rule at the compiler level. In languages like C++, the law of exclusivity can be approximated by adopting *value semantics* instead of *reference semantics*.

By following the law of exclusivity and using value semantics, we significantly improve the applicability of local reasoning.

```
// Adds twice the value of 'y' to 'x'.
void add_twice(int& x, const int& y) {
  x += y;
  x += y;
}
```

Listing 6

Another useful technique for supporting local reasoning is the use of *whole-part relationships* [Stepanov09, Parent15]. Two objects (commonly referred to as *parent* and *child*) are said to be in a whole-part relationship if the following properties hold:

- **Connectedness**: one can reach the child from the parent.
- **Non-circularity**: an object cannot be its own parent (i.e. an object cannot be part of itself).
- **Logical disjointness**: modifying one object does not affect any other distinct object (although shared children are permitted).
- **Ownership**: it is possible to copy an object, modify the copy, and destroy it without affecting the original object.

The last two properties are particularly important for enabling local reasoning. If a function operates on a value of such an object, then modifications to other objects – including copies – will not affect the original.

Containers in the C++ Standard Template Library (STL) exemplify this relationship: they maintain whole-part semantics with respect to the values they contain. When we build objects through *simple composition* (i.e. without pointers or references), and all sub-objects respect the whole-part relationship, the composed object also preserves this property.

Designing data structures around the whole-part relationship is an effective way to enhance local reasoning.

The third tip we offer for supporting local reasoning is to *craft good abstractions* and assign *clear, well-defined contracts* to them. As the saying goes, *the devil is in the details* – so ensuring that an abstraction's contract accurately captures its intended semantics is essential for enabling local reasoning. For a useful introduction to this topic, the reader is encouraged to watch the 'Contracts in C++' talk by Sean Parent and Dave Abrahams, presented at *CppCon 2023*.

As for the art of creating good abstractions, there is a wealth of literature on the principles of sound software design. We will just leave the reader with a relevant quote from Edsger W. Dijkstra [Disjkstra72]:

> The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Conclusions

Local reasoning is a powerful tool. Since the 1970s, when structured programming first emerged, the core idea of local reasoning has served as a vital mental aid – helping programmers make sense of software, both when reading existing code and writing new code. This benefit alone secures local reasoning a place among the most valuable techniques in software development. But there is more to it. As this article has explored, local reasoning also plays a crucial role in enabling formal reasoning about correctness.

Achieving correct software is a long-term effort, but local reasoning offers a path toward steady, composable progress. By crafting strong contracts, avoiding cyclic dependencies, enforcing the law of exclusivity, and favouring value semantics, we can build systems in which reasoning is tractable – one function, class, or component at a time. This approach applies equally well to greenfield development and to the modernisation of legacy code.

In recent years, the software industry – especially the C++ community – has increasingly focused on *safety*. Yet, as important as safety is, *correctness* is the harder goal. As Alexander Stepanov once said:

> Understanding why software fails is important, but the real challenge is understanding why software works.

Local reasoning gives us that ability.

If there is one takeaway from this article, it is that local reasoning is not a luxury – it is a necessity for sustainable software. While language and tooling support for enforcing local reasoning remains limited, we as engineers can take deliberate steps to design and structure our systems with this principle in mind. In doing so, we move closer to building software that is not only powerful, but understandable – and ultimately, trustworthy. ■

References

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

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

[Abrahams22c] Dave Abrahams, 'Values: Safety, Regularity, Independence, and the Future of Programming', *CppCon 2022,* https://www.youtube.com/watch?v=QthAU-t3PQ4.

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

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

[Disjkstra72] Edsger W. Dijkstra, 'The Humble Programmer', *ACM Turing Lecture 1972*, https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html.

[Henney21] Kevlin Henney, 'Beyond the Known Knowns', 2021, https://www.youtube.com/watch?v=eNeOzOoipQs.

[Hoare69] Hoare, C. A. R., 'An axiomatic basis for computer programming' in *Communications of the ACM*, 12(10), 1969, http://sunnyday.mit.edu/16.355/Hoare-CACM-69.pdf.

[Lamport77] Leslie Lamport, 'Proving the correctness of multiprocess programs' in *IEEE transactions on software engineering 2*, 1977, https://www.microsoft.com/en-us/research/publication/2016/12/Proving-the-Correctness-of-Multiprocess-Programs.pdf.

[Martin97] Robert C. Martin, 'Granularity: The Acyclic Dependencies Principle (ADP)', https://web.archive.org/web/20151130032005/http://www.objectmentor.com/resources/articles/granularity.pdf, 1997.

[Martin09] Robert C. Martin, *Clean Code: A handbook of agile software craftmanship*, Prentice Hall, 2009.

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

[O'Hearn01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang, 'Local Reasoning about Programs that Alter Data Structures', *Lecture Notes in Computer Science*, Volume 2142, Springer, 2001, http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/localreasoning.pdf.

[Parent15] Sean Parent, 'Better Code: Data Structures', *CppCon 2015*, https://www.youtube.com/watch?v=sWgDk-o-6ZE.

[Parent23] Sean Parent, Dave Abrahams, 'Contracts in C++', *CppCon 2023*, https://www.youtube.com/watch?v=OWsepDEh5lQ.

[Parent24] Sean Parent, 'Local Reasoning in C++', *NDC TechTown 2024*, https://www.youtube.com/watch?v=bhizxAXQlWc.

[Racordon22a] Dimi Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams and Brennan Saeta, 'Implementation Strategies for Mutable Value Semantics' https://www.jot.fm/issues/issue_2022_02/article2.pdf.

[Racordon22b] Dim Racordon, 'Val Wants To Be Your Friend: The design of a safe, fast, and simple programming language', *CppCon 2022*, https://www.youtube.com/watch?v=ELeZAKCN4tY&list=WL.

[Stepanov09] Alexander A. Stepanov, Paul McJones, *Elements of programming*. Addison-Wesley Professional, 2009.

[Wikipedia] Mars Climate Orbiter: https://en.wikipedia.org/wiki/Mars_Climate_Orbiter.

Simple Compile-Time Dynamic Programming in Modern C++

Compile time code can be very efficient. Andrew Drakeford demonstrates how to write efficient chains of matrix multiplication.

Modern C++ enables us to solve mathematical optimisation problems at compile time. With the expanded constexpr capabilities [Fertig21, Turner18, Turner19, Wu24], we can now write clear and efficient optimisation logic that runs during compilation. Fixed-size containers such as `std::array` fit naturally into these routines. Even standard algorithms, such as `std::sort` and `std::lower_bound`, are now constexpr, enabling more straightforward code and more powerful compile-time computations. Additionally, compile-time optimisation generates constant results, which enables the compiler to create even more efficient code. We will use the matrix chain multiplication problem as our worked example.

Matrix chain multiplication

Matrix chain multiplication is a classic *dynamic programming* problem [Corman22, Das19, Mount]. It aims to determine the most efficient method for multiplying a sequence of matrices. Since matrix multiplication is associative, the order of grouping does not affect the result. However, the number of scalar multiplications involved can vary depending on the grouping.

Consider the three matrices $\mathbf{A_1}$ (10×100), $\mathbf{A_2}$ (100×5), and $\mathbf{A_3}$ (5×50), multiplied in a chain, $\mathbf{A_1 \times A_2 \times A_3}$.

There are two ways to multiply them:

1. **Grouping as $(\mathbf{A_1 \times A_2}) \times \mathbf{A_3}$** first computes a 10×5 matrix, then multiplies that with $\mathbf{A_3}$. This results in 5,000 operations for the first multiplication, and another 2,500 for the second – a total of 7,500 scalar multiplications.
2. **Grouping as $\mathbf{A_1 \times (A_2 \times A_3)}$** first multiplies $\mathbf{A_2}$ and $\mathbf{A_3}$, then $\mathbf{A_1}$. This results in 25,000 operations for the first step and 50,000 for the second – a total of 75,000, which is clearly worse.

Setting up the dynamic programming problem

We define the cost of computing the matrix chain as the total number of scalar multiplications. To solve the dynamic programming problem, we need to find the grouping which minimises the cost. Let us consider the matrix chain shown in Figure 1.

If we split the matrix chain in half, we obtain the two sub-chains as shown in Figure 2.

On each side of the split, we multiply the matrices together (shown in blue: the uppers, smaller boxes) to produce the two intermediate matrices (shown in red, as 'result'). To complete the chain multiplication, we multiply the intermediate results together. The total cost for the chain is:

Total cost = left chain cost + right chain cost + split point cost

The split point cost is the cost of multiplying the intermediate (red, 'result') matrices together to complete the chain. By pre-calculating the minimum cost for both left and right chains, we can quickly determine the lowest cost for the entire chain by evaluating each possible split and selecting the least expensive option.

Figure 1

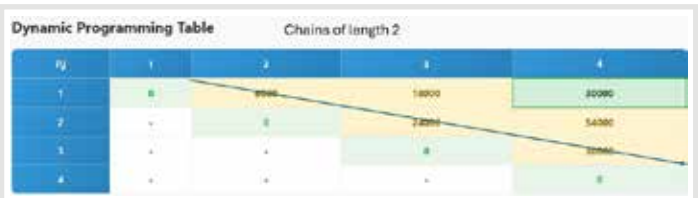
Figure 2

Solving it with dynamic programming: the algorithm

We approach this by using bottom-up dynamic programming, i.e., solving the smallest subproblems first. Consequently, we work over chains of increasing length, finding their optimal split position and minimum cost. For each chain, we calculate the cost at each possible split point and then identify the optimal split point. We store the optimal costs of each chain (subsequence) in a square matrix, where the row and column indices define the start and end points of the chain, respectively. This enables us to retrieve the optimal costs for both left and right chains, avoiding recalculation. We use an additional array, the split matrix, to store the optimal split position for each chain. Figure 3 shows the dynamic programming matrix.

The code

Listing 1 (next page) shows the function `matrix_chain_dp`. It takes a fixed-size `std::array` containing the matrix dimensions of the chain and returns a square array of the optimal split points for all sub-chains. The function has two local variables, `dp` and `split`, which are square



The line shows the chains of length 2.

Figure 3

Andrew Drakeford holds a PhD in physics and has dedicated the past few decades to developing high-performance financial libraries and applications using C++. He has a keen interest in quantitative finance, machine learning, vectorisation, and high-performance computing (HPC). Additionally, he is a member of the UK C++ panel and can be reached at andreedrakeford@hotmail.com.

compile-time optimisation generates constant results, which enables the compiler to create even more efficient code

```
// Function to compute MCM using Bottom-Up DP
// (constexpr)
template <std::size_t N>
constexpr std::array<std::array<int, N>, N>
  matrix_chain_dp(const std::array<int, N + 1>&
  dims)
{
  std::array<std::array<int, N>, N> dp{};
  std::array<std::array<int, N>, N> split{};

  // Initialize dp table for chains of length
  // 1(have zero cost) since no multiplication
  for (std::size_t i = 0; i < N; ++i)
  {
    dp[i][i] = 0;
  }
  // Bottom-Up DP computation
  for (std::size_t len = 2; len <= N; ++len)
  // iterate over chain length
  { // Chain length
    for (std::size_t i = 0; i < N - len + 1; ++i)
    // iterate over chain start position
    {
      std::size_t j = i + len - 1;
      dp[i][j] = std::numeric_limits<int>::max();
      for (std::size_t k = i; k < j; ++k)
      // iterate over split position
      {
        // optimal cost  = cost of left chain
        // + cost of right chain + cost of split
        int cost = dp[i][k] + dp[k + 1][j]
          + dims[i] * dims[k + 1] * dims[j + 1];
        if (cost < dp[i][j])
        {
          dp[i][j] = cost; // store lowest cost
          split[i][j] = k; //store optimal split
        }
      }
    }
  }
  return split;
}
```

Listing 1

arrays used to store the optimal cost and split point for each sub-chain considered.

The function solves the dynamic programming problem in a bottom-up manner. The code iterates through chain length, position, and split point in nested loops. At each split, it evaluates the cost function and stores the minimum cost and its split point in the **dp** and **split** matrices, respectively. The function then returns the **split matrix**.

Listing 2 shows an example of running the optimisation at compile time. The function `main` establishes an array, `dims`, which is `constexpr`. We initialise it declaratively, with the matrix chain dimensions. It is passed into the constexpr function `matrix_chain_dp`, which initialises the constexpr variable `split` with the optimisation results.

```
int main()
{
  // Given Matrix Chain: 6 Matrices (DIMENSIONS)
  constexpr std::array<int, 7> dims = {
    10, 100, 5, 50, 10, 100, 5 };
  constexpr std::size_t N = dims.size() - 1;

  // Compute split table at compile time
  constexpr auto split
    = matrix_chain_dp<N>(dims);
  // Print optimal parenthesization
  std::cout << "Optimal Parenthesization: ";
  print_optimal_parenthesization(split, 0,
    N - 2);
  std::cout << "\n";
  return 0;
}
```

Listing 2

The result, `split`, is an array of arrays, representing a square matrix, which contains the value of a chain's optimal split point. The rows and columns index the start and end points of a sub-chain, respectively.

To extract the optimal chain sequence, we recursively unpack the **split matrix**, starting from the top-right cell of the first row, which yields the split point for the entire chain (from 0 to N-1). This point divides the chain into left and right sub-chains, whose optimal splits can also be found in the **split matrix**.

The function `print_optimal_parenthesization` retrieves the optimal evaluation order and prints it out to give the result shown below:

```
Optimal Parenthesization: ((M1 x M2) x ((M3 x M4) x M5))
```

This is our first Godbolt example, which is available at https://godbolt.org/z/9a9TP6aos

However, to use something like this in real code, we need to execute the matrix chain calculation using the results given by the split matrix. For this, we use the split matrix to build a tree structure of expression templates for the matrix multiplications. Our second Godbolt example illustrates this https://godbolt.org/z/b3x3ao14K

This example creates a chain of twelve matrices. It performs both a naïve (in declaration order) matrix chain evaluation and an optimised multiplication, recording the times taken to calculate the results.

Figure 4 (next page) compares the run time of a matrix chain multiplication for naïve and optimal cases.

Conclusion

Modern C++ provides the necessary features to develop concise code that can solve optimisation problems at compile time. The values resulting from the optimisation process are constant, which enables the compiler to generate even more efficient executables. The optimisation results

Modern C++ provides the necessary features to develop concise code that can solve optimisation problems at compile time

could even define a structure, such as an expression tree, for the optimal execution of a custom domain-specific language.

It is not necessary to limit our optimisation approach to dynamic programming; alternative techniques, such as graph algorithms or linear programming, may also be employed. The range of potential optimisation problems that we could tackle at compile time is vast; we need to make sensible choices about when this sort of mundane wizardry is appropriate. ■

References

[Corman22] Thomas Corman, Charles Leiserson, Ronald Rivest and Clifford Stein. (2022). *Introduction to Algorithms*. 4th Edition. MIT Press. ISBN13: 978-0262046305

[Das19] Avik Das (2019). Dynamic programming deep-dive: Chain Matrix Multiplication. medium.com. Retrieved from https://medium.com/@avik.das/dynamic-programming-deep-dive-chain-matrix-multiplication-a3b8e207b201

[Fertig21] Andreas Fertig (2021) *Programming with C++20 Concepts, Coroutines, Ranges, and more* Fertig Publications.

[Mount] Dave Mount (n.d.). 'Dynamic Programming: Chain Matrix Multiplication' Retrieved July 2025, from https://www.cs.umd.edu/class/spring2025/cmsc451-0101/Lects/lect10-dp-mat-mult.pdf

[Turner18] Jason Turner (2018) 'Practical constexpr', *Meeting C++* Retrieved from https://isocpp.org/blog/2018/02/practical-constexpr-jason-turner

[Turner19] Jason Turner (2019) 'Applied constexpr: Doing More Work At Compile Time', *cppConn2019*. Retrieved from https://cppcon.org/class-2019-constexpr/

Wu Yongwei (2024) 'C++ Compile-Time Programming', *Overload* 183, available at https://accu.org/journals/overload/32/183/wu/


```
Created 12 matrices with random values.
M0: 190x10
M1: 10x1600
M2: 1600x140
M3: 140x10
M4: 10x150
M5: 150x2000
M6: 2000x12
M7: 12x180
M8: 180x10
M9: 10x1600
M10: 1600x140
M11: 140x190
Matrix ((((((((((M0xM1)xM2)xM3)xM4)xM5)xM6)xM7)xM8)xM9)xM10)xM11) (190x190):
Performing naive multiplication...
Naive multiplication took 172ms

Performing optimized multiplication...
Optimized multiplication took 9ms
Matrix (M0x(((((M1xM2)xM3)x(((M4xM5)xM6)x(M7xM8)))x(M9xM10))xM11)) (190x190):
Results match: Yes
Final matrix dimensions: 190x190
```

Figure 4

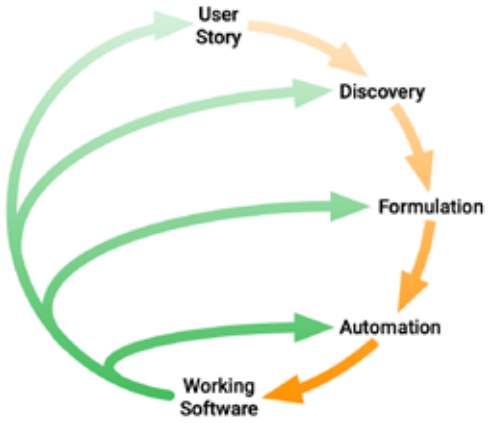
UI Development with BDD and Approval Testing

Testing with confidence. Seb Rose shows a way to approach UI testing.

This article explores the challenges of applying a Behaviour-Driven Development (BDD) approach to UI development. In addition to giving a high-level introduction to BDD, I'll describe a technique called Approval Testing that complements traditional assertion-based testing to give developers clearer visibility of the correctness of their implementation.

What is BDD?

BDD is an agile development approach in which three practices are applied to each story/backlog item: Discovery, Formulation, and Automation. Much has been written about BDD and there are many good introductory articles available, but here I'd like to stress that these practices must be applied in the correct order. Discovery, then Formulation, then Automation. [Rose24]



In the context of BDD, Automation means the writing of test code before the code under test has been written. Just like Test-Driven Development (TDD), the failing automated test drives the development of the production code. There are two implications of this approach:

- Every test should be seen to fail when it is written and seen to pass when the correct production code is implemented. Never trust a test that you haven't seen fail.
- The automation code must be written either by the people who will write the production code or by someone who is collaborating very closely with them.

Should automation be end-to-end?

There's a common misconception that all BDD scenarios will be automated end to end, exercising the entire application stack. Since each BDD scenario is only intended to specify a single business behaviour, using an end-to-end approach for each one would be incredibly wasteful:

- Runtime – end-to-end tests take longer to run than tests that exercise specific areas of the code.
- Noise – the more of the code each test exercises, the more likely it is that many tests will all hit the same code paths. So, an error in that code path will cause all the tests that use it to fail, even if that part of the code has nothing to do with the business behaviour the scenario was created to illustrate. In the face of multiple failing scenarios, it's hard to diagnose which behaviour has deviated from specification.

The 'Test Automation Pyramid' [Rose20] is a common metaphor that suggests most tests should not be end to end. Applying this approach to BDD automation means that we should consider the most appropriate automation to ensure that a specific behaviour has been implemented as specified.

How should the UI be tested?

BDD scenarios that describe how the UI should behave are usually written using tools such as Selenium. These can be slow and brittle because the UI is often tightly coupled with the rest of the application. However, conceptually, the UI is a component that interacts with other application components. It should therefore be possible to test the UI in isolation, near the bottom of the Test Automation Pyramid, by providing test doubles for the application components that it depends on.

Many applications are architected in such a way that the UI can only be exercised as part of an end-to-end test. Whenever possible, a more flexible architecture (that allows the UI to be exercised with the rest of the application 'stubbed out') should be preferred.

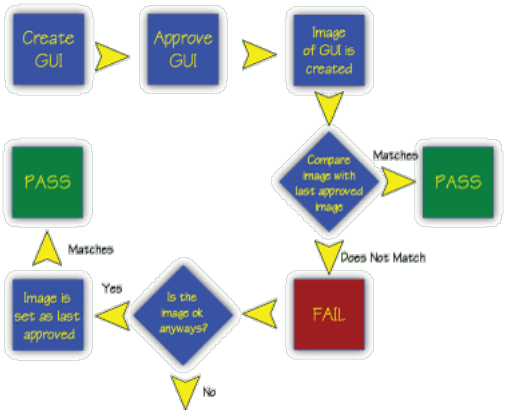
Is that all?

Even with a flexible architecture and automation that conforms to the Test Automation Pyramid, there are challenges. Most test frameworks come with simple assertion libraries that verify if text or numerical values are set as expected. If you need to validate all the fields in a report, you will need an assertion for each of them. This approach leads to verbose automation code that is time consuming to write and difficult to maintain. Additionally, as soon as one assertion fails, the whole test fails without checking any subsequent assertions.

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. Co-author of the BDD Books series Discovery and Formulation (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

For many years, a technique called Approval Testing has been used in these situations, and several tools have been developed to help teams incorporate approval testing into their software development processes. The mechanics of how the tools work vary, but their approach is the same:

1. The first time the test is run, the output is checked manually. If correct, this output is stored as the 'approved' output. If not, fix the software and repeat until the output produced is correct.
2. On subsequent test runs, the tool will compare the output produced to the 'approved' output previously recorded. If they are found to be the same, then the approval test has passed. If not, the approval test has failed. [Falco08]



Naturally, it's not quite as simple as that. For example, if the complex output that we're comparing includes timestamps, these will likely be different each time the test is run. Therefore, approval testing tools typically include mechanisms to specify parts of the output that should not be compared. These exclusions will be specified when the approval test is first created and stored alongside the test.

Does approval testing work for UIs?

Simply specifying areas of the output that should not be compared is insufficient if we're trying to automatically verify the correctness of a visual component. Perhaps a text field is now too close to the border of a control or one visual element is overlaying/obscuring another one.

In these situations, machine learning (ML) and artificial intelligence (AI) can deliver huge benefits. Our tests can leverage these technologies to identify these hard-to-spot issues to a precision that the human eye cannot. But they take time – and slow feedback from a build is the enemy of automated testing.

Instead, AI/ML powered visual tests should be run in a separate stage in the build pipeline, after the faster automated checks have already passed. This ensures that developers get the fast feedback they require while also delivering confidence that the UI is free of visual defects.

If the visual tests pass, then all is well. If there's a failure during the visual tests, then manual investigation is required – because not all failures indicate a defect in the code.

When is a failure not a fail?

We normally think of a test as having a binary outcome. It either passes or fails. Life in software development is rarely that simple. To ensure that the software we ship satisfies our customers' needs, we want to minimize false positives. So, when a test passes, we need to be confident that the behaviour being verified is implemented correctly.

When a test fails, it doesn't necessarily mean that the behaviour has been implemented incorrectly. There are three expected situations that cause a test to fail:

1. Incorrect implementation: this could be caused by a misunderstanding of the specification or an error in the implementation.
2. Incorrect specification: the test is performing the wrong check(s) or the check(s) are being carried out incorrectly.
3. BDD/TDD: the test has been written before the behaviour it's designed to check has been implemented.

When any sort of failure happens in a build, investigation is required. If you find that Situation 1 or 2 has occurred, fix the defect (either in the implementation or the specification) and run the build again.

Situation 3) is a signal to the development team that the work is incomplete. Seeing an automated test fail is an important part of all BDD/TDD workflows. Usually, we would like the failure to be seen in the developers' environment and made to pass before being pushed to CI. However, some workflows may see the tests committed and pushed before the behaviour being verified is implemented.

AI/ML powered visual approval testing

There are several popular, free, open source approval testing tools available ([TextTest], [ApprovalTests]). Their support for visual comparison is limited (absent in the case of TextTest), but there are techniques that, used in conjunction, may be sufficient for your needs (see the Printer section in this article by Emily Bache [Bache19] for example).

With the increasing availability of AI/ML techniques, a number of visual testing tools are now available that incorporate AI functionality to facilitate the validation of complex graphical applications. Applitools is possibly the most popular commercial offering [Applitools], but there are many others available with competing functionality and pricing.

Conclusion

The development team need regular, fast feedback to give them visibility of important aspects of their software's quality. They need confidence that they're developing the right thing in the right way.

BDD and TDD are techniques that give developers that confidence (among other benefits). Currently, most organisations that adopt these approaches use popular assertion-based tools to verify that the code being developed satisfies the specifications. This focus on assertion-based testing is unsuitable for some of the subtle and complex issues that occur when developing today's applications.

Approval testing in all its flavours can help bridge the gap between automated assertion checking and time-consuming manual testing. Existing approval testing libraries are excellent when dealing with complicated textual outputs and simple graphical components. Visual testing tools are emerging that leverage AI/ML to bring approval testing for modern UIs within reach of reliable automated testing.

References

[Applitools] Applitools: https://approvaltests.com/

[ApprovalTests] Approval Tests: https://approvaltests.com/

[Bache19] Emily Bache 'Approval Testing', published at https://approvaltests.com/on 23 July 2019.

[Falco08] Llewellyn Faloc 'Approval Tests (a picture worth a 1000 tests)' posted at https://llewellynfalco.blogspot.com/2008/10/approval-tests.html on 13 October 2008.

[Rose20] Seb Rose 'Eviscerating the Test Automation Pyramid', available at https://cucumber.io/blog/bdd/eviscerating-the-test-automation-pyramid/, posted 7 February 2020.

[Rose24] Seb Rose 'Behaviour-Driven Development', available at https://cucumber.io/docs/bdd, last updated November 2024.

[TextTest] TextTest: https://www.texttest.org/

This article was first publishedon Seb Rose's blog on 8 February 2023: https://cucumber.io/blog/bdd/bdd-approval-testing-and-visualtest/. It has been reviewed and updated for *Overload*.

Trip report: C++ On Sea 2025

Another year, another trip report from C++ On Sea! Sándor Dargó shares what he learned.

First, a heartfelt thank-you to the organizers for inviting me to speak, and an equally big thank-you to my wife for taking care of the kids while I spent three days in Folkestone – plus a few more in London to visit the Spotify office and catch up with some bandmates.

If you have the chance, try to arrive early or stay around Folkestone for an extra day. It's a lovely town and it's worth exploring it. The conference program is very busy even in the evenings, so don't count on the after hours. This year, I arrived an half a day in advance and I had a wonderful hike from Folkestone to Dover. It was totally worth it.

In this article, I'll share:

- Thoughts on the conference experience.
- Highlights from talks and ideas that resonated with me.
- Personal impressions, including reflections on my own sessions – both the main talk and the lightning talk.

My favourite talks

Before diving into individual sessions, I want to highlight the overall mood at the conference – full of enthusiasm and excitement. C++ On Sea 2025 took place just after the WG21 meeting in Sofia, Bulgaria, where several game-changing proposals were discussed and adopted. Herb Sutter's keynote focused entirely on a few features of C++26, and Timur Doumler's talk spotlighted an exciting upcoming feature: contracts.

With that context in mind, here are my three favourite talks, presented in the order they were scheduled.

Three cool things in C++ (26) by Herb Sutter

Herb is part of that rare breed of presenters. He is extremely knowledgeable, his presentation style is entertaining and if this wouldn't be enough, he is also always enthusiastic.

As I mentioned earlier, *C++ On Sea 2025* took place right after a particularly productive committee meeting, which added even more excitement to the atmosphere – both for Herb and for the audience.

Herb's keynote focused on three major features coming in C++26:

- Erroneous behaviour
- Reflection
- `std::execution`

Thanks to the new erroneous behavior, uninitialized variables will no longer result in undefined behaviour by default. You can still opt in to keep variables uninitialized, but now, as Herb put it, "*the sharp knife is in a drawer by default*". (If you're curious to dive deeper, I have written more about erroneous behaviour on my blog [Dargo-1].)

Now, reflections – this one is huge. We can't possibly cover it in just a few paragraphs. But according to Herb and pretty much everyone I've talked to, it's set to be a real game changer for C++. We don't yet fully grasp all the possibilities it will unlock.

In a nutshell, reflections will provide a standardized, generalized API to a language-level abstract syntax tree (AST). In C++26, we'll be able to reflect on types, functions, and parameter lists. The rest will follow in C++29. What's especially exciting is that, unlike in many other languages, C++ reflection will be entirely compile-time, meaning there's *no runtime overhead*.

This article isn't the right place to go into too much detail, but reflection is expected to massively simplify things like creating language bindings. Just as I did for concepts, I plan to dedicate an entire blog series to reflections – both to learn it myself and to share my insights with you.

Software engineering completeness pyramid: knowing when you are done and why it matters by Peter Muldoon

Peter is another highly energetic speaker, and his talk focused on the hardcore discipline of software engineering – without touching a single line of code. He opened with a question we all hear from our managers:

> Are you done yet?

But how can we *really* know? And, once we're 'done', how can we be sure the software actually delivers value?

Peter argued that software only brings value when it is *available, usable, and reliable – all at the same time*. Large and slow releases rarely meet that standard; small, incremental changes that satisfy all three criteria do far better.

To clarify what it really takes to deliver valuable software, he introduced the 'Software Engineering Completeness Pyramid' (see Figure 1, next page), a four-level hierarchy reminiscent of Maslow's. As with Maslow, you usually need to satisfy each level before you can meaningfully think about the next one.

Where do *you* operate?

- Are you simply shipping features and squashing bugs?
- Have you moved up a level to caring about codebase health – an essential step toward senior-engineer territory?
- Do you think like a systems engineer, considering how each change fits the broader architecture and influences system stability?

Sándor Dargó 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

In a nutshell, reflections will provide a standardized, generalized API to a language-level abstract syntax tree (AST)

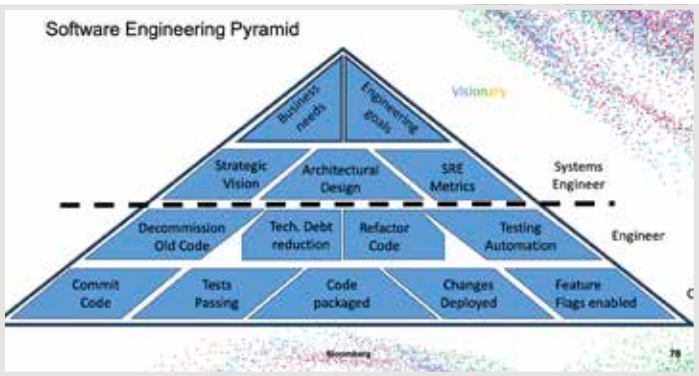


Figure 1

- Or have you reached the summit, weighing every decision against business goals and market positioning?

That final tier may seem distant, but we should all keep the business context in mind. After all, there's no point in delivering features the business – and its users – don't actually need.

Why software engineering interviews are broken – and how to actually make them better (Kristen Shaker)

I was deeply moved by Kristen's talk. More on that at the end.

Most of us would likely agree that the software engineering interview process is fundamentally flawed. We're asked to solve LeetCode-style problems, analyze runtime and memory complexities – the infamous Big O – not because they're part of our daily work, but because that's how the industry has standardized hiring.

As Kristen explained, these kinds of interview questions are bad for both developers and companies. Candidates often need to spend months preparing just to stand a chance. A full interview process can take up to 8 hours – just for a single position! Yet if we don't change jobs frequently, we risk falling behind in compensation.

It's a bad deal for companies too. They don't necessarily end up hiring the engineers who would perform best on the job. And because the interview process is so exhausting, many people stay in roles they're unhappy with – quiet quitting instead of seeking new opportunities. This system also tends to favor hiring the same kinds of people over and over, while diverse skill sets would lead to stronger, more balanced teams.

So what can be done instead?

Kristen proposes that we move away from LeetCode-style questions and ask better ones – questions that signal whether someone will actually succeed in the role. Questions that have multiple valid answers. That allow candidates to demonstrate different skill sets. That start simple, but can go deep. That value real-world experience.

Examples of such questions include:

- What's your favorite feature of C++ (or another language)?
- Review this piece of code.
- "Yap" about a past project you worked on.

In my view, we do fairly well at Spotify – but that's clearly not the industry average.

Why was I so touched by this talk?

When I saw Kristen was speaking, I immediately remembered her lightning talk at *C++ On Sea 2022* [Shaker22] about querying the Clang AST. Being curious, I googled her name and found a real estate agent with the same name. She looked familiar, but I thought, "That can't be her."

It is her. She was so fed up with the software industry – especially the interview process – that she left engineering and became a real estate agent instead.

Good luck, Kristen.

My favourite ideas

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

The embedded world needs more C++ (Marcell Juhasz)

Marcell Juhasz gave a talk with the title 'Balancing Efficiency and Flexibility: Cost of Abstractions in Embedded Systems' [Juhasz25]. He essentially took an embedded project written in good-old C and started to add layers of abstractions in C++, making the code more readable, testable and maintainable. Goals that I deeply care about.

But Marcell not only improved the code, he also took measurements after each step. Mostly about binary size as that's what mattered to him the most. If he found any increase, he checked where it comes from and tried to get rid off the increase while keeping the benefits of the new layer of abstraction.

The outcome?

One cannot justify using plain C because of worse performance and bigger binaries. When applied cautiously, modern C++ features are perfect for the embedded world.

Compile-time debugging (Mateusz Pusz)

Mateusz Pusz gave a talk on features that help us write great C++ libraries – both existing ones and those coming with C++26 or later. While the talk was informative and full of useful insights, I want to focus on one specific feature that really stood out to me and that I'd love to explore further: *compile-time debugging*, which will be part of C++29.

Debugging `constexpr` – let alone `consteval` – functions can be quite a challenge. Traditional debugging tools are mostly useless in this domain, making issues hard to trace and fix.

This is where P2758 [P2758R5] comes into play. It introduces new ways to emit messages at compile time – not just plain output via

Three days packed with inspiring talks about various topics, including not just C++26, but embedded, testing, engineering interviews and many more.

`std::constexpr_print_str`, but also compile-time *warnings* using `std::constexpr_warning_str` and even compile-time *errors* via `std::constexpr_error_str`.

These additions go far beyond simple '`printf`-style' debugging at compile time. They allow library authors to:

- Communicate clearly what's going wrong (or right) at compile time.
- Surface warnings proactively before they become runtime issues.
- Provide error messages that are both specific and user-friendly.

I believe these features have the potential to *significantly* improve the developer experience in C++, making compile-time diagnostics clearer and more actionable than ever before. If used well, they could help us build libraries with error messages that are both meaningful and educational – something C++ has long needed.

Difficult test? Think about your design! (Björn Fahller)

Björn gave an excellent talk on software testing. It was both insightful and educational, covering a big variety of testing strategies – from different types of tests and their purposes, to comparisons of various unit testing frameworks.

But there's one key takeaway I want to highlight from his presentation:

> If you find that something is extremely difficult to test, and you just can't figure out how to approach it – don't keep banging your head against the wall.

Instead, **pause and reflect on your API design**.

If testing a component is overly complicated, the problem might not lie in your testing skills or the framework you're using – it could be a sign that your design needs improvement. Clean, testable APIs usually indicate a well-thought-out architecture. On the other hand, if you're struggling to test something, it may be tightly coupled, doing too much, or hiding behavior behind obscure layers of abstraction

My talks

Finally, let me share my contributions to the conference.

My time came very quickly this year. I had my slot on the 'C++ Fundamentals' track right after Herb Sutter's keynote about 'Three Cool Things in C++' [Sutter25]. That's both terrifying and calming at the same time!

I was even more surprised – and humbled – to see Jason Turner attending my talk. I had a brief discussion with him the next day, and he mentioned that there was some overlap between our topics and he wanted to refresh his notes on namespaces. What a pleasant and unexpected surprise!

It's no secret that I talked about *namespaces*. What they are, how they work, and what best practices you should follow when using them. I've already covered some of these topics on my blog [Dargo-2], and more may come. Of course, I'll share the video once it becomes available.

While I had my talk on the first morning, a lightning talk awaited me later that evening. I try to grab these opportunities to speak – it's a great way to fight stage fright! I presented a technique for *designing your workweek*, something I've written about on *The Dev Ladder* [Dargo25].

And finally – no clicker issues this time! After 'Meeting C++' last year, I bought a Logitech Spotlight and it was one of my best conference decisions. No glitches, just smooth transitions. Same goes for my presentation overall – though next time, I'll aim to highlight key points in code examples more clearly.

Conclusion

C++ On Sea was a great experience in 2025, as well as any other year. [Dargo-3] Three days packed with inspiring talks about various topics, including not just C++26, but embedded, testing, engineering interviews and many more.

The best we can do is to spread the word – share the videos, tweet your favourite insights, write about what you learned – so that maybe even more people join next year.

I hope to be back to Folkestone in 2026! ■

References

[Dargo-1] Sándor Dargó's blog posts about erroneous behavior: https://www.sandordargo.com/blog/2025/02/05/cpp26-erroneous-behaviour

[Dargo-2] Sándor Dargó's blog posts about namespaces: https://www.sandordargo.com/tags/namespaces/

[Dargo-3] Sándor Dargó's blog posts about previous *C++ on Sea*: conferences: https://www.sandordargo.com/tags/cpponsea/

[Dargo25] Sándor Dargó 'Where Pomodor meets a spreadhseet' (sic), posted 4 July 2025 at https://devladder.substack.com/p/where-pomodoro-meets-a-spreadhseet

[Juhasz25] Marcell Juhasz, abstract of 'Balancing Efficiency and Flexibility: Cost of Abstractions in Embedded Systems', at https://cpponsea.uk/2025/session/balancing-efficiency-and-flexibility-cost-of-abstractions-in-embedded-systems

[P2758R5] Barry Revzin, 'P2758R5: Emitting messages at compile time', published 2025, available at https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2758r5.html

[Shaker22] Kristen Shaker 'Using Clang Query to Isolate AST Elements', a 'lightning talk' delivered at *C++ on Sea 2022*, available at https://www.youtube.com/watch?v=2LOxsfpCCyI

[Sutter25] Herb Sutter, abstract of 'Three Cool Things in C++', at https://cpponsea.uk/2025/session/three-cool-things-in-cpp

This article was published on Sándor Dargó's blog in July 2025 and is available at https://www.sandordargo.com/blog/2025/07/02/cpponsea-trip-report. The blog will be updated with links to videos when these become available.

AI Powered Healthcare Application

Many people are raving about AI. Hassan Farooq describes how he used it in a project so you can learn how to build an AI model.

Problem

In the healthcare sector, there is a challenge in delivering healthcare advice that is accessible and personalised to patient's needs in a timely manner, especially for individuals who live in areas with limited medical services. This can lead people to rely on online sources which may be unreliable. As a result, they may feel anxious because they don't know if the information is correct and if they follow incorrect advice, their health may get worse.

Meanwhile, healthcare professionals are overwhelmed with high workloads and delivering consultations which are not urgent which could be handled by digital tools. This puts additional strain on healthcare systems, shifting critical resources from urgent care needs.

Therefore, there is an increasing need for intelligent systems that provides users with personalised advice. The key challenge lies in developing a solution that integrates generative AI to offer accurate, timely and user-friendly recommendations while adhering to medical standards.

Implementation

This report documents the development of an AI-Powered Healthcare web application as part of my final year project at University of Bradford for the Software Engineering BEng Hons. I had around eight months to complete this project alongside my modules and with limited resources. Also, I utilised only the tools and technologies within my capacity, without any external support.

I developed an AI-Powered Healthcare web application that integrates Generative AI to provide users with personalised medical advice and support.

To interact with the chatbot, users must securely register and log in. I built the authentication system using JWT authentication [JWT] and role-based access control.

When using the chatbot for the first time, users are required to read and accept a disclaimer. This disclaimer outlines the chatbot's purpose and includes consent for the use of personal data. See Figure 1.

The chatbot initially asks for the user's age and gender. If the input is invalid or irrelevant, it handles the response gracefully with appropriate prompts. Once this information is received, the chatbot then asks about the user's health concern. If the input provided is too vague, it will ask the user to provide more detail.

To handle vague inputs, I implemented a vague input detection system. This works by comparing the user's input to a set of predefined vague

Hassan Farooq is is a Software Engineering BEng (Hons) graduate from the University of Bradford. He is passionate about building intelligent systems that solve real-world problems and to drive digital transformation and help businesses innovate. His interests include AI, programming languages, cloud computing, healthcare technology and the ethics of software development. You can contact him at hassanfarooq105964@outlook.com

Figure 1

phrases using the SentenceTransformer model [SBERT]. Both the user input and the vague phrases are converted into vector embeddings and the cosine similarity between them is calculated. If the similarity score exceeds a predefined threshold, the input is considered vague and the chatbot prompts the user to elaborate. See Figure 2.

Once a clear health concern is provided, the chatbot sends this information including the age, gender and health issue to the Mistral AI language model [Mistral]. To improve the accuracy and relevance of the responses, I enhanced the model using Retrieval-Augmented Generation (RAG), which incorporates official NHS health articles which were stored as vector embeddings in the Pinecone database [Pinecone]. RAG was the best technique to apply because it allowed me to use authoritative NHS articles as a factual base, improving trust and accuracy. Implementing it involved web scraping and cleaning NHS pages, converting the content to embeddings using SentenceTransformers, storing them in Pinecone and querying them based on the user's input before sending to the language model.

I applied prompt engineering to the language model to set the context to healthcare and so it considers the user's age and gender when

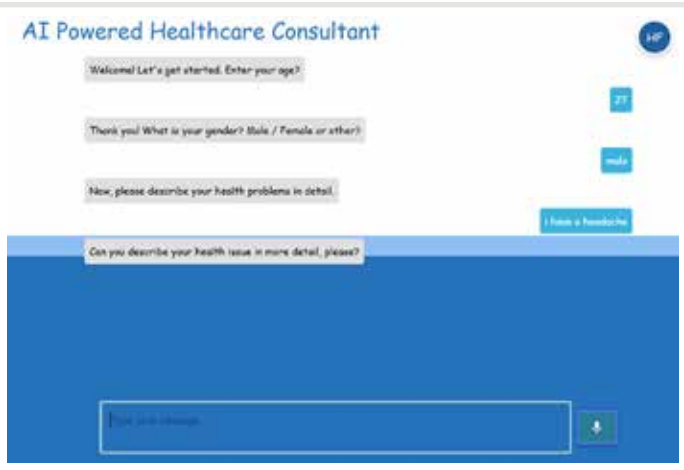


Figure 2

Figure 3

Figure 4

generating health advice. As a result, the model generates responses in a conversational tone while referencing information from the NHS database and it also includes links to the original NHS sources. This promotes user trust, transparency and allows them to explore their health issues further on the NHS website. See Figure 3.

The chatbot features a dynamic input button enabling users to interact via voice recognition or typed text, powered by Google's Speech Recognition API [Google].

Users can also review previous conversations as well as start new chats or delete old ones, giving them full control over their chat history.

Additionally, I implemented a Health Assessment feature on another page. (Figure 4). Here, users answer a series of yes/no health-related questions and at the end, their responses are sent to the language model. The model then provides general health advice and recommendations, allowing users to benefit from the chatbot's support even if they don't have a specific health issue.

Given the sensitive nature of the application, data protection issues were key priorities throughout the development of this application.

- User Consent – Before interacting with the chatbot, users must accept a disclaimer and consent to the use of their data. This step ensures informed participation and transparency.
- Data Minimisation – Only minimal data is collected such as age, gender and the health concern.
- Secure Storage – User credentials like password are hashed using industry-standard hashing algorithms such as bcrypt.
- Chat history is stored in a secured MySQL database where only the user has full control over their previous chats. They can view, delete or start new conversations, this shows privacy of their health interactions.
- Role-based access control ensures that only authorised users like Admins can manage sensitive user data.

Ultimately, the tech stack includes React (front end), Java (back end), Python (for the chatbot and RAG logic), MySQL for data storage and Pinecone for storing NHS data as vector embeddings.

Chatbot testing

Testing was conducted to assess the effectiveness of my chatbots response to real answers. I took 10 random questions from the MedQuad Dataset.

```
PS C:\Users\Hassa\Chatbot> python evaluate_chatbot.py

◆ Chatbot Evaluation Results ◆
                                                Question  My Chatbot Similarity (%)  GPT-4 Similarity (%) Better Response
0  Ive recently had a sinus inffection my doctor ...                      67.570000            67.309998     My Chatbot
1  What are some potential causes of restrictive ...                      73.010002            83.379997          GPT-4
2  I'm having horrible pain in my lower abdominal...                      74.459999            76.699997          GPT-4
3  For 7-8 months I've had ulcers come on my tong...                      64.379997            53.340000     My Chatbot
4  I was recently put on asthma inhalers/ventolin...                      44.910000            46.580002          GPT-4
5  My boyfriend through he just had a panic attac...                      68.120003            68.120003            Tie
6  About six months ago I started to see a tiny l...                      53.029999            56.580000          GPT-4
7  I had blood in my urine for 5 days and then su...                      50.990002            43.009998     My Chatbot
8  When I was younger my mother had stage 4 thyro...                      76.940002            75.449997     My Chatbot
9  My mom suffers from angina, had emergency dial...                      65.250000            77.019997          GPT-4
PS C:\Users\Hassa\Chatbot>
```

Figure 5

MedQuAD is an abbreviation for Medical Question Answering Dataset and it consists of pairs of question and answers which are curated from 12 trusted National Institutes of Health (NIH) websites.

I have made a testing file called `evaluate_chatbot.py` where I send my predefined medical MedQuAD to my chatbot. The response is then collected from my chatbot and compared to GPT 4's (ChatGPT) [GPT] answers to see which is more semantically similar to the expected answer. See Figure 5.

Comparison is done using BERT-based sentence embeddings (all-MiniLM-L6-v2) to calculate cosine similarity. It's basically like an Automated Semantic Evaluation and BERT similarity mainly checks if my chatbot conveys the same idea which is critical in medical and conversational AI tasks.

Results and discussion

Based on the results of the test, ChatGPT which is state of the art, generated long and nuanced answers, whereas my model which is an open source using the free tier, generated responses that were truncated. Nevertheless, the results demonstrate that my chatbot with the support of RAG and prompt engineering is quite competitive with ChatGPT, even with its shorter answers. For a couple of questions, my chatbot nailed it but also felt short for some.

By integrating Retrieval Augmented Generation (RAG) into my chatbot and using NHS articles as the knowledge base, the chatbot retrieves factual information directly from NHS resources before responding to users. This ensures that responses are grounded in reliable data which reduces the risk of the chatbot hallucinating (generating incorrect answers) which is particularly critical in the healthcare domain. For example, if a user's health concern is expressed as, "I have a headache due to dehydration," the model would generate a response that includes relevant NHS articles, such as https://www.nhs.uk/symptoms/headaches/ along with other related resources.

Although the NHS articles are static, the chatbot interacts with them to craft responses in a conversational tone by tailoring the information to suit the user's queries. Additionally, the chatbot can provide relevant links to NHS articles which allows users to explore and read more about their health concerns if they wish. Ultimately, the responses generated by the chatbot are backed by trusted sources, which may enhance user trust, satisfaction and it would allow the chatbot to generate better responses compared to the model itself.

On the other hand, the limitation of the chatbot is that the accuracy of its advice depends heavily on the quality of user input. If users fail to clarify their condition or provide enough details, the chatbot's responses may lack precision. For example, a user inputting difficulty in breathing must indicate whether they have been diagnosed with asthma or how long the pain has been and explain the pain, so the chatbot can tailor its advice accordingly.

Additionally, the Mistral model was selected as a feasible language model due to its availability in a free version, but this comes with limitations because once the free tier has exceeded, payment will be required.

Furthermore, there were 170 NHS articles used in my chatbot which means that my chatbot may not fully have evidence or reliable source for every single healthcare scenario. More number of NHS and other reliable sources would allow the chatbot to cover more healthcare scenarios and maybe improve the results of the chatbot test.

Challenges

My initial plan was to use a language model and fine tune it with datasets related to healthcare. However, most of the models I came across had a large number of parameters, required payment and demanded high memory to run locally. This made it challenging to find a language model that was both suitable and feasible to run on a laptop with 8GB of RAM.

Eventually, I came across the Mistral AI language model which was suitable. My next step was to fine-tune it but I discovered that that this process requires a lot of computational power. Additionally, fine-tuning is more research oriented and the time required for this task would have exceeded the timeframe I had for the project.

One of the biggest challenges I faced was figuring out a way to validate the AI's responses instead of simply relying on a third party language model to generate answers. That's when I discovered Retrieval-Augmented Generation (RAG). After researching it and assessing its suitability, I decided to implement it.

To test the accuracy of the chatbot's advice, I was limited to using 10 questions from the MedQuad dataset. Adding more questions would have exceeded the free tier limit of the language model.

Whilst searching for a reliable dataset, I discovered that RAG allows you to store URLs in its knowledge base after cleaning, extracting the content and converting it into embeddings. Therefore, I chose to use NHS URLs as the source for the RAG knowledge base.

Future work

For future work, more state-of-the-art tools and models such as the latest GPT model or DeepSeek [DeepSeek] could be integrated. This process would be straightforward, involving a simple pick and plug approach to enhance the system's capabilities. The dataset could be expanded to include more NHS articles or other reliable resources could improve the system's coverage of healthcare scenarios.

Additionally, prompt tuning could replace prompt engineering to set the healthcare context of the language model more effectively. By learning the task's prefix rather than relying on handcrafted prompts, this approach has the potential to achieve higher accuracy.

For performance improvement, the model could be running locally on a GPU if feasible, this would allow the response to be a lot faster. In future, expert human evaluation would help assess the medical validity of the chatbot's responses.

Ultimately, my end goal is to have this project deployed on a cloud platform. It will always be open for improvements with additional features in my own time.

If the project were to be a commercial product, medical device approval will be required, GDPR compliance, clinical safety validation and transparency in AI usage. This shows that the process can be complex as it requires consultation with legal, clinical and regulatory experts.

Technology overview

Retrieval-Augmented Generation (RAG): RAG is a generative AI technique which allows you to modify language models with external data.

Prompt Engineering: Prompt engineering is used to set the context and guide language models so they can understand the question and give an appropriate response.

Pinecone: A vector database used for storing embeddings (numerical representations of text) so the system can retrieve the most relevant health information.

Cosine Similarity: A mathematical way used to compare the similarity between two vectors (text embeddings) by calculating the cosine of the angle between them. It helps determine how similar two pieces of text are in terms of meaning regardless of their magnitude.

SentenceTransformer: SentenceTransformer is a library (based on models like BERT) that converts sentences or texts into dense vector embeddings. These embeddings can then be used for tasks like semantic search, clustering, and similarity comparison.

Vector Embeddings: These are numerical representations of text, where semantically similar texts have similar vectors.

State of the art AI health chatbots

There are many different online solutions out there that are designed to allow patients to manage their health, give them quick and useful advice after inputting relevant information which they could gain without booking an appointment.

The Ada Health app [Ada24] was launched in 2011 by a global company founded by medical experts, it has an AI system which aims to make healthcare easier and more effective for users allowing them to manage their health independently. It uses artificial intelligence to help diagnose symptoms [Singh21]. You input your symptoms, and you are required to answer some question from the chatbot. The app then analyses your input and answers to provide possible diagnoses and advice.

The app compares your symptoms with medical dictionaries which it has been trained and based on that, the app is able to generate a personalised report. The app may be able to also assist you with different symptoms such as anxiety, pain, allergies, headache and many more. It's a useful tool for getting quick view of your health problems.

The NHS also has a chatbot known as the Limbic Access chatbot and it's used to streamline the mental health referral process within the NHS [NHS22]. It helps services like Mind Matters Surrey NHS (IAPT) by acting as a digital front door for patients seeking mental health support.

Instead of calling or filling out a long form, you just chat with the bot online. When someone wants help, the chatbot asks them friendly, step-by-step questions about how they're feeling. It collects important details, like symptoms or if they're at risk and sends that information to the NHS team. This helps staff save time because they don't have to ask those questions again. Ultimately, it makes the process of asking for mental health help quicker, easier and less stressful for both patients and staff. ■

References

[Ada24] Ada: https://ada.com/about/

[DeepSeek] DeepSeek: https://www.deepseek.com/

[Google] Speech Recognition: https://pypi.org/project/SpeechRecognition/

[GPT] Generative Pre-trained Transformer: https://openai.com/index/introducing-gpt-4-5/

[JWT] 'Introduction to JSON Web Tokens: https://jwt.io/introduction

[Mistral] Mistral AI: https://mistral.ai

[NHS22] NHS Transformation Directorate (2022), 'Using an AI chatbot to streamline mental health referrals', available at: https://transform.england.nhs.uk/key-tools-and-info/digital-playbooks/workforce-digital-playbook/using-an-ai-chatbot-to-streamline-mental-health-referrals/

[Pinecone] Pinecone Vector Database: https://www.pinecone.io

[SBERT] SentenceTransformers: https://www.sbert.net/

[Singh21] Singh, V., (2021) 'Benzinga: Artificial intelligence doctor app Ada Health closes $90M funding led by Bayer, Samsung', available at https://www.benzinga.com/m-a/21/05/21322232/artificial-intelligence-doctor-app-ada-health-closes-90m-funding-led-by-bayer-samsung

Afterwood

Debuggers have been around for a long time. Chris Oldwood ponders these typically under-appreciated tools.

If debugging is the process of removing software bugs, then programming must be the process of putting them in.
~ Edsger W. Dijkstra

When it comes to most books and technologies I'm typically very late to the party – the rest of the world has usually moved on just as I'm getting interested in it. Somewhat unusually for me, then, I ended up reading the book *Coders at Work* by Peter Seibel within the first year of its publication (2009). A natural consequence of always being late to the party is that most of my books are second-hand, so it was especially unusual for me to own, and be reading a brand-new book. The circumstances that led to this seemingly Black Swan like event were my upcoming birthday, an online bookstore wish-list, and a mother struggling (yet again) to think of a present for her forty-year-old son.

I must confess that at the time I didn't know who most of the programmers were that the author had interviewed – I only recognised four out of the fifteen names, and one of those had only entered my consciousness a couple of months earlier at the StackOverflow Dev Day. As someone who considered themself reasonably well-read this was somewhat of a surprise. Fifteen years later I can still only tell you who nine of the fifteen are, despite reading the entire book. It turns out names are hard in the real-world too – remembering them, that is.

Anyway, what I do remember most about that book was the subject of debugging. Most of the interviewees made very little use of a debugger, if at all, print-style debugging appeared to be their default technique. And yet, based on their biographies and conversations, these people have clearly all worked on non-trivial codebases so I couldn't fathom how they'd manage to live without such an important diagnostic and exploratory tool.

At that point in time, my programming career had focused entirely on lower-level programming languages, such as assembly language, C, and then C++. My first experience with a debugger was with the Pyradev toolset on the Amstrad CPC6128, although they called it a 'monitor' rather than a debugger, but you could still single-step through machine code. From there to Devpac on the Atari ST, CodeView under 16-bit Windows (not forgetting the truly awesome SoftICE when you needed to pull out the big guns), and eventually to Visual Studio over its many incarnations (with CDB/NTSD/WinDbg making guest appearances here and there). In essence, from my mid-teens in the 80's right up to my forties (and eventually on to my fifties), a debugger has been a fundamental part of my programming toolkit, even as I moved into the managed world of C# and .Net.

In contrast, the last five years has seen me working in an in-house language that has no debugger, and so consequently print-style debugging has become my *only* choice. This language is a pure functional language, so memory-safety is not a source of bugs, and people generally write small functions which are easy to invoke on a whim via a REPL. It's a very different world, and yet I still yearn for a debugger, not necessarily to help fix bugs, though it would help reduce the time there, but mostly to help visualise the flows through the 100K+ lines of code.

When my son was very young, he came into the office one day when I was working from home and stood behind my chair and watched my screen. Eventually he asked, "What *are* you doing dad?" I explained that I was "Debugging some code". Later he remarked to my wife that all I did all day was, "Make a yellow line move up and down the screen". (I was using an editor colour scheme where the current statement in the debugger was highlighted with a yellow bar. As an aside, that choice of colour scheme – blue background with white text – also dates back to Pyradev.)

He wasn't wrong. Source code is inherently static, and what a debugger does is to animate it. Suddenly the instruction pointer stops being an abstract hexadecimal value as the debugger translates that into positions within source files. Flow control stops being something you have to mentally picture because the debugger shuffles the code around on screen as you enter and exit functions and cycle around loop constructs. Most mainstream programming languages feature closures, which are simplified by syntactic sugar, but the debugger typically reveals what's going on under the hood. (Maybe motion sickness during debugging could be a proxy for code complexity?)

This idea of using a debugger as a code exploration tool, not just something you reach for in times of despair, is far from new. In *Writing Solid Code*, Steve Maguire proposed you also step through all *new* code in the debugger as a way of testing it. In essence, the debugger allows you to take the same journey as the CPU before it hits production. It's much easier to spot an incorrect flow or off-by-one loop error when you're watching the code execute in slow motion.

My move away from system programming languages has definitely caused my skills with a debugger to atrophy. That's not the only reason though – Test-Driven Development has also given me confidence in my code in a way that allows me to avoid firing up the debugger 'to see it in action', more times than not. Even so, old habits die hard, and being able to fire up the debugger and use a unit test to quickly get into the production code is simply the icing on the cake and yet another way to unearth further test scenarios that weren't obvious from the static viewpoint.

I've never really understood the backlash against using a debugger, as if 'real programmers' *only* need print statements, though maybe Chuck Norris is probably the exception. I suspect Brian Kernighan's famous quote probably hasn't helped: "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

Of course, he wrote that way back in 1979 and therefore likely suffers from the same malaise as *that* optimisation quote from Sir Tony Hoare. Debugging Katas are quite a niche exercise too, which doesn't exactly help their popularity outside crisis management.

Having someone take my toys away from me has undoubtedly been a force for good as it's prompted me to 'think harder' (apparently that's not just for LLMs) and put even more effort into making my code easier to reason about, but I'd still prefer to have a debugger in my back pocket for those Lewis & Clark style expeditions. ■

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

accu

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

accu.org

To connect with like-minded people visit accu.org

accu