overload 191

FEBRUARY 2026 £4.50

Implementing vector<T>

Quasar Chunawala explores implementing your own version of std::vector.

Effective Behavior Driven Development

Seb Rose condenses 15 years of BDD learning into two pages of practices, benefits and challenges.

Letter to the Editor

Silas S. Brown writes in response to the article 'Why I don't use AI', published in Overload 190.

Restrict Mutability of State

Kevlin Henney reminds us that when it is not necessary to change, it is necessary not to change.

Afterwood

Chris oldwood reminds us that spending time mulling things over can also be productive.

A magazine of ACCU

ISSN: 1354-3172

February 2026
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

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

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Alison Peck. Near the Levi Ski Resort, Finnish Lapland.

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* 192 should be submitted by 1st March 2026 and those for *Overload* 193 by 1st May 2026.

Everything is Under Control

You can't always plan every detail in advance. Frances Buontempo tries to step back and find out how to respond to change.

When my routine gets disrupted, it takes a while to get back into the swing of things. For example, I went to *Meeting C++* [MeetingC++] in Berlin in November last year and gave a keynote. Being invited to give a talk is a great honour (thank you, Jens), but trying to get back to various tasks when I got home was challenging. Fortunately, our last edition had a guest editor, Quasar Chunawala, who wrote an introduction to coroutines for us [Chunawala25]. I was off the hook! Of course, I haven't written an editorial this time either, as ever. The new year crept up on me and the break put me off. Furthermore, I've been finishing up my new book, an introduction to C++ for O'Reilly [Buontempo26]. I feel like I can't read or write anything, ever again now. I might get over it. We'll see.

A regular routine helps me feel like I have everything under control. It's just a feeling and though it helps me, it might not work for you. Furthermore, life throws curved balls from time to time. Disruptions can give you an opportunity to stop and think. Running on your rails frequently means you are being somewhat mindless. We all need to take a moment to reflect once in a while, and maybe dream about new possibilities. And sometimes, we need to stop for a bit. We are not machines. Stopping and taking stock might help you realise a better approach. Nonetheless, a routine can provide motivation. You do something because it's the next thing to do.

Sometimes sticking with a habit isn't a good idea. When I write code, I occasionally follow my nose and end up with a tangled mess. I bet you have done similar: you need code to do something that's similar to something you've done before so maybe don't stop and think first. Trying to find good examples for my introductory C++ book has been a challenge, too. I was tempted to write short snippets in various places which I hadn't even compiled. I did stop myself, you'll be pleased to hear. Sometimes sketching out a high level plan, whether functions or classes or data flow, makes you stop and think. Some basic principles keep everything under control: version control, tests, maybe even code reviews and some code sanitizers. Certainly checking that code compiles! How do you keep everything under control? Write us an article and tell us your approach ☺

No matter how hard we try, problems sneak in. That might be bugs, or design problems. Even if you think you've come out with a clean architecture, future feature requests can scupper your plans. That said, there are some high-level heuristics that can diminish the chance of bad things happening. C++ has many examples of these, such as 'almost always auto' [Sutter13]. Of course, they are heuristics, and there are always exceptions. Well, maybe not always: embedded code often doesn't use exceptions. That's another story for another time. Every programming language tends to have heuristics or guidelines. However, things change over time, so you need to keep up to date. Maybe that's why you're reading this magazine? Hopefully you learn something each time you do.

Let's consider a heuristic. Years ago, the received wisdom was to pull a call to a container's `size` outside a loop for efficiency. The theory was a call inside a loop would happen each time around the loop. The following might therefore fail a code review:

```
for (size_t index = 0; i< stuff.size(); ++i)
```

Again, we could start another discussion about old-skool `for` loops versus range-based approaches, but won't. Calling `size` in the loop used to be inefficient, but compilers usually optimise this call out of the loop now. I watched Matt Godbolt's 'Advent of Compiler Optimisation' on YouTube [Godbolt25a]. Matt called out various ways in which a compiler might generate code that's different to what you've typed. Hopefully, the idea isn't a surprise to you. What is surprising is the various ways in which optimisations are achieved. The final (actual) episode summed the numbers from 0 to x, for some number x,using a `for` loop. Various compilers removed the loop and found a quicker way, using $x(x\text{-}1)/2$. You may recognize the formula as Gauss summation, but Matt has details on his blog if you'd rather read than listen [Godbolt25b]. You might have known some tips for making code quicker, but you need to keep your knowledge up to date. Things change. This means you might not have everything under control. Seeing a compiler completely remove a loop and use the closed-form formula for a sum is amazing. Of course, you can control the level of optimisations. Less aggressive levels are less likely to completely remove your hand-crafted loops. All of the other advent of code optimisations are worth watching too. I bet you learn something.

Trying to keep everything under control goes beyond the code itself, as anyone who has had to support a production system knows. Logging can be very useful, but sometimes doesn't provide enough information. Chris Oldwood wrote an article called 'Terse Exception Messages' a while ago [Oldwood15]. He started with a line from a log:

```
ERROR: Failed to calibrate currency pair!
```

If the message also said which currencies were involved, Chris' life would have been easier. Thinking through whether the information logged is useful or not is important. One system I worked on had very chatty logging. Full sentences and many extraneous details. One of the team spent time reformulating the logging, to make the lines easier to search with regex. Many of the lines in the log became shorter, which were slightly quicker to write out so sped up the process a tiny bit. An added bonus. More importantly, being able to find relevant lines in one search with a script made our lives much easier. Sometimes a bit of thought and planning can give you more control. Just saying.

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 called *Genetic Algortithms and Machine Learning for Programmers*, and one to help you catch up with C++ called *Learn C++ by Example*. 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.

It can be hard to find a good balance between logging everything or just a few salient messages. Recently, the news in the UK mentioned an increase in theft of bicycles from train stations. "So what?" I hear you cry. Well, many stations have bike racks, and these are often covered by CCTV. Unfortunately, this means there are many hours of footage which takes time to watch, so the British Transport Police said they can't spend all their time watching the footage because then they won't be able to patrol railway stations. It's likely that a bike theft would be on the footage but there's too much to look through, like the chatty logs I mentioned. The BBC News reported this with the attention grabbing headline: Bike thefts at stations 'decriminalised' [BBC25]. Now, I wonder if you could get a machine learning system to spot the moment when a bicycle is removed from a spot – it can't be that hard. That would at least narrow down the time period of footage that needed investigating. You 'just' need to say which square contains a bike and which doesn't and clock the timestamps. You could go the extra step of trying to use AI to recognize the face of the person taking the bike, but that's another story – certainly one that is causing controversy in the UK at the moment. If anyone is studying computer vision, I may have thought of a research project for you: 'Spot the bike'.

Too much information can be overwhelming. A long time ago, I wrote about drowning in emails (and more besides) [Buontempo12]. My emails are out of control again. No matter how many lists I unsubscribe from, I seem to end up on more and more every week. Setting up filters helps, giving me some control over what I see in my inbox, but it doesn't entirely solve the problem. Perhaps I need a machine learning system to filter and summarise my emails. And reply as well… except now I suspect I am talking about various attempts at 'AI' personal assistants. I don't really want GenAI to write a summary for me. One option would be abandoning email altogether, but that might cause other problems. I guess the trick is prioritising what needs dealing with and what can wait, in conjunction with finding a way to remember the things that are waiting. When you are faced with a wall of noise/information, you need to find a good way to search through it, or narrow down what you are trying to figure out. The same goes with internet searches and even reading books. It's OK to skip from the table of contents to the index and go straight to the pertinent parts. You don't need to read all the things.

OK, so you can use machines to automate tasks, including searching your logs or emails to find information. However, just because an approach is trendy or possible, doesn't make it a good idea. It's still challenging to stop GenAI 'hallucinating' (making stuff up). You can try more specific prompts, tell the LLM to own up if it doesn't know or to provide a chain of reasoning, explaining the steps. This will never give you complete control. Many suspect we can never stop GenAI making stuff up [O'Brien23], and to my mind that is how LLMs are designed: precisely to make things up. In case you missed it, we had an article from Andy Balaam last time explaining why he doesn't use (Gen)AI [Balaam25]. He suggested the results are sometimes *dangerously* incorrect. We have a follow up letter to the editor in this edition. I am pleased to see people discussing GenAI and questioning it. Any new tech can be useful, but that does not mean you have to fully embrace it and use it for everything.

Another aspect of GenAI people find disconcerting is that the same prompt can give different replies. This is by design. If you can turn the temperature parameter down to 0, you'll get the same response each time. [Noble], unless the model changes under your feet. If you've ever worked with models using random numbers, this will be familiar. Probabilistic models use random numbers deliberately, allowing you to explore expected outcomes or averages. In theory you can record the seed used for the pseudo-random numbers and repeat the outcome. Of course the same seed on different compilers might give a different number sequence [Reddit].

They say the best laid plans of mice and men often go astray. That doesn't mean it's not worth trying to form a plan. You might need to be prepared to follow a back–up plan or trying something else. Don't get carried away trying to plan for every eventuality. You'll end up catastrophizing and won't get anything done. And, if something does go wrong it will probably be something you didn't even think of. I was giving a talk at the ACCU conference once and a power surge fried my laptop live on stage. I did have the talk itself in bitbucket, but the UI had recently changed and I struggled to find the right workspace on a friend's laptop. Everyone in the room tried to help, and we got there in the end. However, I couldn't run my demos. Subsequently, I've tried to embed mp4s into my slides, but they often won't play. I just need to learn to 'roll with the punches': like boxing, a side swipe can potentially knock you out. If instead, you accept things don't always go to plan, you might get better at coming out with a plan B just when you need it. Try to take some control, but not 100%. Let surprises happen too. They aren't always bad. And you might learn something. As the agile manifesto says, we have come to value 'Responding to change over following a plan' [Agile].

References

[Agile] 'Manifesto for Agile Software Development', available at http://agilemanifesto.org/

[Balaam25] Andy Balaam 'Why I don't use AI' in Overload 33(190):4–5, December 2025, available to members only at https://accu.org/journals/overload/33/190/balaam/

[BBC25] Tom Edwards, 'Bike thefts at stations 'decriminalised'', posted 2 October 2025 at https://www.bbc.co.uk/news/articles/c8jm3wxvlkjo

[Buontempo12] Frances Buontempo 'Too Much Information' in *Overload*, 20(111):2–3, October 2012, available to members only at https://accu.org/journals/overload/20/111/buontempo_1885/

[Buontempo26] Frances Buontempo (2026) *Introducing C++*, O'Reilly Media, Inc., https://www.oreilly.com/library/view/introducing-c/9781098178130/

[Chunawala25] Quasar Chunawala 'A Guest Editorial' in *Overload* 190, December 2025, available to members only at https://accu.org/journals/overload/33/190/chunawala-buontempo/

[Godbolt25a] Matt Godbolt 'Advent of Compiler Optimisations 2025', available at https://www.youtube.com/playlist?list=PL2HVqYf7If8cY4wLk7JUQ2f0JXY_xMQm2

[Godbolt25b] Matt Godbolt 'When compilers surprise you', *Matt Godbolt's blog*, posted December 2025, available at https://xania.org/202512/24-cunning-clang

[MeetingC++] *Meeting C++* website: https://meetingcpp.com/

[Noble] Joshua Noble 'What is LLM Temperature?' (date unknown), IBM, available at https://www.ibm.com/think/topics/llm-temperature

[O'Brien23] Matt O'Brien and The Associated Press 'Tech experts are starting to doubt that ChatGPT and A.I. 'hallucintations' will ever go away: 'This isn't fixable'' *Fortune*, posted 1 August 2023, available at https://fortune.com/2023/08/01/can-ai-chatgpt-hallucinations-be-fixed-experts-doubt-altman-openai/

[Oldwood15] Chris Oldwoood 'Terse Exception Messages' in Overload 23(127):15–17, June 2015. Available to members only at https://accu.org/journals/overload/23/127/oldwood_2110/

[Reddit] 'Inconsistency in C++ random', discussion held approximately 11 years ago on https://www.reddit.com/r/cpp/comments/30w7cs/inconsistency_in_c_random/

[Sutter13] Herb Sutter, 'GotW #94 Solution: AAA Style (Almost Always Auto)' on *Sutter's Mill*, posted 12 Aug 2013, available at https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

Effective Behavior Driven Development

Most software development aims to provide solutions to business problems. Seb Rose has condensed fifteen years of BDD learning into two pages of practices, benefits, and challenges.

It's tempting to think of software development as a purely technical task—writing code to instruct a machine. In reality, software is the product of an ongoing socio-technical conversation among users, business stakeholders, and the delivery team. That conversation evolves over time, and the challenge is to capture it in a form that is precise, testable, and useful as a guide for building the system. This is the central goal of Behavior-Driven Development, or BDD.

BDD originated in the mid-2000s as a refinement of Test-Driven Development (TDD), created to make technical practices more accessible and closely aligned with business needs. Today, it is supported by a mature ecosystem of tools and years of hard-earned lessons from practitioners. From the initial discovery of requirements to the use of automated tests that ensure applications adapt gracefully to changing needs, BDD offers a collaborative, sustainable way to bridge the gap between intent and implementation.

Why is software delivery so hard?

The purpose of most software development is to deliver solutions to business problems. Despite well over 50 years industry experience, organizations still regularly experience significant challenges specifying, delivering and maintaining the software systems on which they rely.

You might be the victim of these challenges if you have struggled to find answers for some of the following questions:

- What does the customer actually need?
- How should we capture these needs in an unambiguous way?
- How do we make sure that these are understood by everyone involved in specification and delivery?
- How can we demonstrate that the functionality of what we deliver meets the customer's needs and is adequately reliable?
- How can we ensure that the software will be able to adapt over time?

These questions span the entire software development process, from requirement analysis to maintainability, but in many cases the underlying issues are rooted in the following three key problems.

- Incomplete requirements – the requirements do not properly convey the information about the problem to be solved and the expected behavior of the solution to the team.
- Unreliable documentation – the maintenance of the system is hampered by the absence of documentation that reliably links the stakeholder needs to the functionality that the team delivers.
- Slow feedback – delay, unnecessary rework, or context-switching is required because of the lack of fast, reliable, meaningful feedback about system behavior.

An overview of BDD

Behavior Driven Development (BDD) is an agile approach to software development that closes the gap between business people and technical people. BDD emphasizes the collaboration needed to create and maintain linkage between requirements, documentation, tests and the system being developed. It is made up of three practices, shown in Figure 1, which illustrates the order that the BDD practices should be applied to iteratively 'drive out' each small increment of functionality in your application.

1. User stories are lightweight descriptions of a piece of functionality that will be of value to some user of the system. They are not requirements but are created in response to an understanding of the needs of the stakeholders.
2. Discovery is a structured, collaborative activity that uses examples to discover the detailed requirements of a user story. This practice helps uncover the ambiguities and misunderstandings that traditionally derail software projects.
3. Formulation is a creative process that turns the examples produced during discovery into business-readable scenarios. The subsequent review of the scenarios delivers the confidence that the team really has understood what the stakeholders are asking for.
4. Automation is where code is written that turns the scenarios into tests. Not only do the tests guide the implementation of the system, but they also transform the scenarios into living documentation. Every time the system is built, the tests give us feedback that the scenarios still accurately describe its behavior.
5. Working software is our ultimate result that BDD contributes to. There are many activities that take place after each scenario is automated, before the functionality described by the scenario can be delivered to users. These activities are not directly related to BDD.

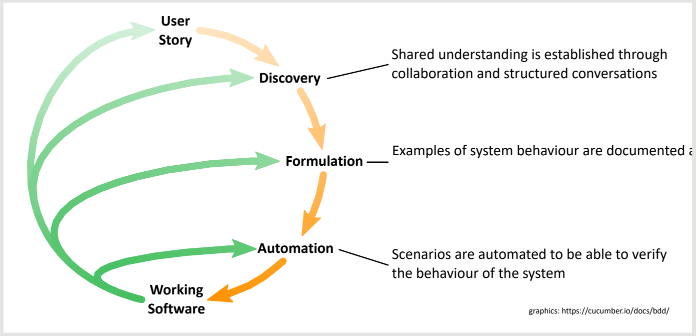

User Story
Discovery
Shared understanding is established through collaboration and structured conversations
Formulation
Examples of system behaviour are documented a
Automation
Scenarios are automated to be able to verify the behaviour of the system
Working Software
graphics: https://cucumber.io/docs/bdd/

Figure 1

Seb Rose has been a consultant, coach, designer, analyst and developer for over 40 years. Lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

User stories

User stories are a widely used agile concept. Each story initially captures an element of application functionality that might be valuable to the customer but defers detailed requirements gathering until the story has been prioritized for development.

Discovery

During discovery, the team participates in requirement workshops at which they create examples to explore and illustrate the expected behavior of the story. Focusing on examples makes the intention of the acceptance criteria and business rules clear – each should be illustrated by one or more examples. This is important because acceptance criteria and business rules are often subject to misunderstandings.

Formulation

The examples generated during discovery are the bridge between the requirements and the software. For this bridge to be useful for both business and technical people, the examples must be captured in a form that is accessible and meaningful to both. Formulation is the creative process that turns each example produced during discovery into a business-readable *scenario* that is understandable by all stakeholders, yet also precise enough to specify the software that needs to be written.

Automation

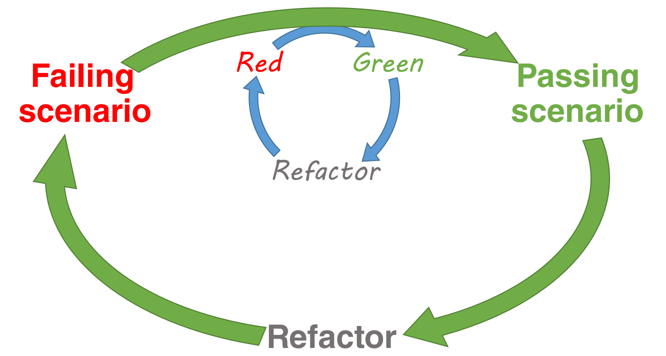
Once an example has been formulated as a scenario, it can be read by automation tools that understand the format produced during formulation. The team can now write automation code that these tools will call in response to each line in the scenario.

The team starts by writing any new automation code needed by the scenario they are currently working on. When they run the new scenario, it should initially fail (red) – because the implementation code that it specifies has not been written yet. Then they iteratively implement the application code (possibly using TDD) until the scenario finally passes (green). After cleaning up the codebase (refactoring), they move on to the next scenario. See Figure 2.

Automated scenarios also address many of the issues around unreliable documentation. Since the automation will be run during every build, the team will be notified whenever the system does not behave in the way a scenario expects it to. These failures have several possible causes:

- The functionality has not been implemented yet: implement the functionality.
- There is a defect in the code: fix the defect.
- The specification is incorrect or out of date: correct the scenario.

In this way, automated scenarios preserve a direct connection between the specification and the system implementation. This type of documentation is called *Living Documentation*.



The BDD cycle is the outer loop – write a failing scenario, get the scenario to pass, then refactor. We enter the inner, TDD cycle to implement the code needed to move from a failing scenario to a passing scenario.

Figure 2

Testing

BDD itself does not define how testing should be performed. Instead, it provides a set of practical guidelines that facilitate the agile testing process. The basic concept of agile testing is to move the responsibilities of testing from finding and reporting application issues to ensuring that these issues are never added to the codebase in the first place. When following a BDD approach, the responsibility for code quality is shouldered by the whole delivery team, not just dedicated testers.

Bridge

The examples uncovered during discovery are an executable connection between the behavior required by the stakeholders and the system implemented by the delivery team for the lifetime of the product:

- They provide the stakeholders with confidence that the team has implemented their requirements in a verifiable way.
- They provide the delivery team with confidence that they have correctly understood the stakeholders' requirements.
- They provide the organization with confidence that side effects and regressions will be prevented as the system evolves.
- They provide the organization with confidence that there is reliable, persistent documentation of how the system behaves, ensuring its ongoing maintainability as the business and team changes over time.

BDD helps to maintain this connection and so acts as a bridge between the stakeholders and the delivery team, between the organization and the product, and between the past and the future.

Conclusion

BDD is a collection of practices that, on their own, seem mostly sensible and beneficial. Each can be described quite succinctly and don't appear to be particularly demanding. However simple they sound, adopting them requires change, and no change is easy.

When done well, in whole or part, BDD practices have delivered improved outcomes for thousands of organizations globally. However, the collaborative and technical practices that make up BDD are dependent on the organization's willingness to change not just what work is done, but how the work is done.

BDD is not a free lunch. Discovery will not succeed unless the product owner is truly available to collaborate with the team. Formulation will not succeed unless the business stakeholders are truly available to review the scenarios. Automation will not succeed unless the delivery team is truly willing to value all aspects of code quality. BDD is simple but not easy – you will have to work for your lunch.

BDD is not a silver bullet. There are many problems in product design and delivery. BDD focuses on aspects of software development that range from business requirements through to delivery and operations, but there are many aspects that fall outside the scope of BDD practices. There are problems that BDD cannot solve – you will have to decide if the problems BDD helps solve are the problems you most need to solve.

All change is hard, but change is possible and often necessary. We have tried to emphasize that BDD adoption is challenging, not to put you off, but to give you realistic expectations. The outcomes of successful BDD adoption – less waste, fewer defects, reduced rework, faster feedback, living documentation among them – are worth working towards. We have written a book, *Effective Behavior Driven Development*, to help you to adopt BDD practices. ■

This article is taken from Chapter 1 of the book *Effective Behavior Driven Development* by Gáspár Nagy and Seb Rose, published by Manning and available from https://www.manning.com/books/effective-behavior-driven-development
Gáspár Nagy is the creator of SpecFlow & Reqnroll, brings over 20 years of experience as a coach, trainer, and test automation expert. He currently leads SpecSync, aiding teams in test traceability with Azure DevOps and Jira.

Implementing vector<T>

Finding out to to implement features from the standard library can be a useful learning exercise. Quasar Chunawala explores implementing your own version of std::vector.

If you know std::vector , you know half of C++.
~ Bjarne Stroustrup

The most fundamental STL data-structure is the **`vector`**. In this article, I am going to explore writing a custom implementation of the **`vector`** data-structure. The standard library implementation **`std::vector`** is a work of art, it is extremely efficient at managing memory and has been tested *ad nauseam*. It is much better, in fact, than a homegrown alternative would be.

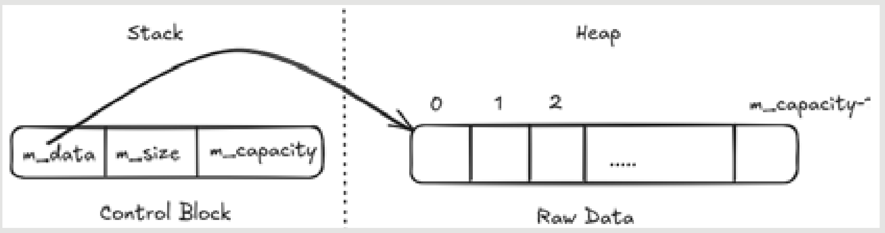


Figure 1

Why then write our own custom **`vector`**?

- Writing a naive implementation is challenging and rewarding. It is a lot of fun!
- Coding up these training implementations, thinking about corner cases, getting your code reviewed, revisiting your design is very effective at understanding the inner workings of STL data-strucures and writing good C++ code.
- Its a good opportunity to learn about low-level memory management algorithms.

We are not aiming for an exhaustive representation or implementation, but we will write test cases for all basic functionalities expected out of a **`vector`**-like data-structure.

Informally, a **`std::vector<T>`** represents a dynamically allocated array that can grow as needed. As with any array, a **`std::vector<T>`** is a sequence of elements of type **`T`** arranged contigously in memory. We will put our homegrown version of **`vector<T>`** under the **`dev`** namespace.

Unit tests for vector<T>

For low-level data-structures such as the **`vector`**, let's write the unit-tests upfront before the implementation. This will help us think through the interface and corner cases. Tests will also serve as documentation of the expected functionality.

The internal representation of a **`vector`**-like type has a book-keeping node (see Figure 1) that consists of:

- A pointer to the raw data (a block of memory that will hold elements of type **`T`**)
- Size of the container(the number of elements in the container)
- Capacity

It's important to distinguish between **`size`** and **`capacity`**. **`size`** is the number of elements currently in the container. When **`size == capacity`**, the container becomes full and will need to grow, which means allocating more member, copying the elements from the old storage to the new storage and getting rid of the old storage.

Given this background, we assume that the **`vector`** is equipped with basic getters such as:

- **`std::size_t size()`**
- **`std::size_t capacity()`**
- **`bool empty()`**
- **`bool is_full()`**

The vector should support various constructors (see Listing 1).

```
TEST(VectorTest, DefaultConstructorTest) {
  dev::vector<int> v;
  EXPECT_EQ(v.empty(), true);
}

TEST(VectorTest, InitializerListTest){
  dev::vector<int> v{1, 2, 3, 4, 5};
  EXPECT_EQ(!v.empty(), true);
  EXPECT_EQ(v.size(), 5);
  EXPECT_TRUE(v.capacity() > 0);
  for(auto i{0uz}; i < v.size(); ++i){
    EXPECT_EQ(v.at(i), i+1);
  }
}

TEST(VectorTest, ParameterizedConstructorTest){
  dev::vector v(10, 5.5);
  EXPECT_EQ(v.size(), 10);
  for(auto i{0uz}; i < v.size(); ++i){
    EXPECT_EQ(v[i], 5.5);
  }
}

TEST(VectorTest, CopyConstructorTest){
  dev::vector v1{ 1.0, 2.0, 3.0, 4.0, 5.0 };
  dev::vector v2(v1);
  EXPECT_EQ(v1.size() == v2.size(), true);
  for (int i{ 0 }; i < v1.size(); ++i)
  {
    EXPECT_EQ(v2[i], i+1);
    EXPECT_EQ(v1[i], v2[i]);
  }
}
```
Listing 1

Quasar Chunawala is a quant-developer. He is deeply passionate about programming in C++, Rust, concurrency and performance-related topics. He enjoys long-distance hiking. He regularly discusses interesting code snippets/language features on his blog and runs a monthly C++ newsletter at https://quantdev.blog/newsletter.

```
TEST(VectorTest, MoveConstructorTest){
  dev::vector<int> v1{ 1, 2, 3 };
  dev::vector<int> v2(std::move(v1));
  EXPECT_EQ(v1.size(), 0);
  EXPECT_EQ(v1.capacity(), 0);
  EXPECT_EQ(v2.size(), 3);
  for(auto i{0uz}; i<v2.size(); ++ i)
    EXPECT_EQ(v2[i], i + 1);
}

TEST(VectorTest, CopyAssignmentTest)
{
  dev::vector<int> v1{ 1, 2, 3 };
  dev::vector<int> v2;
  v2 = v1;
  EXPECT_EQ(v1.size(), v2.size());
  EXPECT_EQ(v1.capacity(), v2.capacity());
  for (int i = 0; i < v1.size(); ++i) {
    EXPECT_EQ(v1[i], i+1);
    EXPECT_EQ(v1[i], v2[i]);
  }
}

TEST(VectorTest, MoveAssignmentTest)
{
  dev::vector<int> v1{ 1, 2, 3 };
  dev::vector<int> v2;
  v2 = std::move(v1);
  EXPECT_EQ(v1.size(), 0);
  EXPECT_EQ(v1.capacity(), 0);
  EXPECT_EQ(v2.size(), 3);
  for (int i = 0; i < v1.size(); ++i) {
    EXPECT_EQ(v2[i], i+1);
  }
}
```

Listing 1 (cont'd)

```
TEST(VectorTest, AtTest)
{
  dev::vector<int> v{ 1, 2, 3 };
  EXPECT_EQ(v.at(0), 1);
  EXPECT_EQ(v.at(1), 2);
  EXPECT_EQ(v.at(2), 3);
  EXPECT_THROW(v.at(3), std::out_of_range);
}

TEST(VectorTest, SubscriptOperatorTest)
{
  dev::vector<int> v{ 1, 2, 3 };
  for (int i{0uz}; i < v.size(); ++i) {
    EXPECT_EQ(v[i], i+1);
  }
}
```

Listing 2

```
TEST(VectorTest, EmptyTest)
{
  dev::vector<int> v;
  EXPECT_EQ(v.empty(), true);

  v.push_back(42);
  EXPECT_EQ(v.empty(), false);
}

TEST(VectorTest, SizeAndCapacityTest)
{
  dev::vector<int> v;
  EXPECT_EQ(v.size(), 0);
  EXPECT_GE(v.capacity(), 0);

  v.push_back(42);
  EXPECT_EQ(v.size(), 1);
  EXPECT_GT(v.capacity(), 0);

  v.push_back(v.back());
  EXPECT_EQ(v.size(), 2);
  EXPECT_EQ(v[1], 42);
}
```

Listing 3

```
TEST(VectorTest, ReserveTest)
{
  dev::vector<int> v1;
  v1.reserve(10);
  EXPECT_GE(v1.capacity(), 10);
  EXPECT_EQ(v1.size(), 0);

  dev::vector<int> v2{1, 2, 3, 4, 5, 6, 7};
  size_t old_capacity = v2.capacity();
  EXPECT_GE(v2.capacity(), 7);
  EXPECT_EQ(v2.size(), 7);
  size_t new_capacity = 2 * old_capacity;
  v2.reserve(new_capacity);
  EXPECT_GE(v2.capacity(), new_capacity);
  EXPECT_EQ(v2.size(), 7);
  for(auto i{0uz}; i < v2.size(); ++i)
    EXPECT_EQ(v2[i], i + 1);
}
TEST(VectorTest, ResizeTest)
{
  dev::vector<int> v{ 1, 2, 3 };
  v.resize(5);

  EXPECT_EQ(v.size(), 5);
  for(auto i{0uz}; i<3; ++i)
    EXPECT_EQ(v[i], i + 1);

  EXPECT_EQ(v[3], 0);
  EXPECT_EQ(v[4], 0);

  v.resize(2);
  EXPECT_EQ(v.size(), 2);
  EXPECT_EQ(v[0], 1);
  EXPECT_EQ(v[1], 2);
}
```

Listing 4

The **`vector`** data-structure should support element access through the array subscript operator **`[]`**, just like C-style arrays. The **`T& at(std::size_t idx)`** could also be used to access the element at index **`idx`** with bounds checking. (See Listing 2.)

We expect the container to perform the book-keeping of size and capacity correctly. (See Listing 3.)

We expect the container to support the getter methods **`front()`** and **`back()`**:

```
TEST(VectorTest, FrontAndBackTest)
{
  dev::vector<int> v{ 1, 2, 3 };
  EXPECT_EQ(v.front(), 1);
  EXPECT_EQ(v.back(), 3);
}
```

The container should support **`reserve(size_t new_capacity)`** and **`resize(size_t  new_size)`**. These are explained at length further ahead. (See Listing 4.)

The container should support appending elements or removal elements at the back. (See Listing 5, next page.)

The container should support a **`.insert`** method that allows us to insert elements from a source range to a specified position in the target vector. You can write a variety tests, like inserting at the beginning, middle, end of the vector, self-referential insertion etc. For brevity, I skip listing all of the tests here. The Compiler Explorer online source listing for this entire article is available in the conclusion section.

vector member data

We start with coding up the **`vector`** as a class template. It is templated by the type **`T`** of the elements stored in the container. We also define various type aliases (Listing 6).

C++ containers usually expose iterators as part of their interface and ours will be no exception. We define type aliases for the **`const`** and non-**`const`** iterator types, as this makes it simpler to implement alternatives. (See Listing 7).

```
TEST(VectorTest, PushBackTest)
{
  dev::vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  EXPECT_EQ(v.size(), 3);
  for(auto i{0uz}; i<v.size(); ++i)
    EXPECT_EQ(v[i], i + 1);
}
TEST(VectorTest, PushBackSelfReferenceTest)
{
// The design of push_back/insert is slightly
// hard to get right. If the vector is full, then
// you reallocate(grow) the vector. If the value
// to be added is a reference to an existing
// vector element, then value in
// vec.push_back(value) may become a dangling
// reference, if it refers to the old storage (an
// element of the vector itself e.g. vec.back()).
// This test is meant for such an edge case.
  dev::vector<int> vec{ 1 };
  for (auto i{0uz}; i < 64; ++i) {
    vec.push_back(vec.back());
    EXPECT_EQ(vec.back(), 1);
  }
}
TEST(VectorTest, EmplaceBackTest)
{
  struct Point
  {
    int x, y;
    Point(int a, int b)
      : x(a)
      , y(b)
    {
    }
  };
  dev::vector<Point> v;
  v.emplace_back(1, 2);
  v.emplace_back(3, 4);
  EXPECT_EQ(v.size(), 2);
  EXPECT_EQ(v[0].x, 1);
  EXPECT_EQ(v[0].y, 2);
  EXPECT_EQ(v[1].x, 3);
  EXPECT_EQ(v[1].y, 4);
}
TEST(VectorTest, PopBackTest)
{
  dev::vector<int> v = {1, 2, 3};
  EXPECT_EQ(v.size(), 3);
  v.pop_back();
  EXPECT_EQ(v.size(), 2);
  EXPECT_EQ(v, dev::vector<int>({1, 2}));
}
```

Listing 5

vector constructors

Alluding to the rule-of-five, we implement a copy constructor, copy-assignment operator, move constructor, move assignment operator and a destructor.

```
template <typename T>
  class vector {
  using value_type = T;
  using size_type = std::size_t;
  using pointer = T*;
  using const_pointer = const T*;
  using reference = T&;
  using const_reference = const T&;
  using iterator = pointer;
  using const_iterator = const_pointer;
private:
  pointer m_data{nullptr};
  size_type m_size{0uz};
  size_type m_capacity{0uz};
  constexpr static unsigned
    short growth_factor{2};
```

Listing 6

```
template <typename T>
class vector {
  // ...
public:
  iterator begin(){ return m_data; }
  const_iterator begin() const{ return m_data; }
  iterator end(){ return begin() + m_size; }
  const_iterator end() const{
    return begin() + m_size; }
// ...
```

Listing 7

```
template<typename T>
class vector{
  private:
  struct init_capacity_tag { size_type cap; };
};
// If an exception happens after this has been
// called, the destructor will run and deallocate
// the memory.
explicit vector(init_capacity_tag cap)
: m_data{ allocate_helper(cap.cap).release() }
, m_capacity{ cap.cap }
{ }
```

Listing 8

To simplify things, we first code up a private constructor (**`vector(init_capacity_tag)`**) whose job is to allocate memory and construct a **`vector`** object. (See Listing 8.)

All other constructors delegate to this private **`vector(init_capacity_tag)`** constructor. After that constructor completes, the object is fully constructed, and if any execution happens later which throws, the destructor will always be called.

The destructor is called for all fully constructed objects. The object is considered fully constructed once any constructor has finished, including the delegated constructor. (See Listing 9.)

```
vector() noexcept
{}
// If an exception happens after this has been
// called, the destructor will run and deallocate
// the memory.
explicit vector(init_capacity_tag cap)
: m_data{ allocate_helper(cap.cap).release() }
, m_capacity{ cap.cap }
{ }

vector(size_t n, const T& initial_value)
: vector(init_capacity_tag(n))
{
  std::uninitialized_fill_n(m_data, n,
    initial_value);
  m_size = n;
}

vector(std::initializer_list<T> list)
: vector(init_capacity_tag(list.size())) {
  std::uninitialized_copy(list.begin(),
    list.end(), m_data);
  m_size = list.size();
}

vector(const_iterator first, const_iterator last)
: vector(init_capacity_tag(std::distance(first,
last)))
{
  if constexpr (
      std::is_nothrow_move_constructible_v<T>) {
    std::uninitialized_move(first, last, m_data);
  } else {
    std::uninitialized_copy(first, last, m_data);
  }
  m_size = std::distance(first, last);
}
```

Listing 9

```
vector(const vector& other)
: vector(init_capacity_tag(other.capacity())) {
  // Perform a deep-copy of all the Ts
  std::uninitialized_copy(other.m_data,
    other.m_data + other.m_size, m_data);
  m_size = other.size();
}
vector(vector&& other) noexcept
: m_data{std::exchange(other.m_data, nullptr)},
m_size{std::exchange(other.m_size, 0)},
m_capacity{std::exchange(other.m_capacity, 0)}
{}
void swap(vector& other) noexcept {
  std::swap(this->m_data, other.m_data);
  std::swap(this->m_size, other.m_size);
  std::swap(this->m_capacity, other.m_capacity);
}
vector& operator=(const vector& other) {
  vector(other).swap(*this);
  return *this;
}
vector& operator=(vector&& other) {
  vector(std::move(other)).swap(*this);
  return *this;
}
~vector(){
  std::destroy(begin(), end());
  raw_deleter{}(m_data);
}
```

Listing 9 (cont'd)

If any of the delegating constructors fails in the constructor body – such as `vector(size_t n, const T& initial_value)` – the `~vector()` destructor has to be run. This makes memory handling almost entirely automatic.

Basic services of a vector-like class

Implementing front() , back() and operator[](size_t idx)

There is more to writing a convenient dynamic array type. For example, member functions that let you access the elements at front or rear-end of the vector are to be expected. Similarly, an implementation of `operator[]` to access the element at a specific index in the array is also expected. (See Listing 10.)

Comparing two `vector<T>` objects for equivalence or lack thereof is a relatively easy matter if we use algorithms:

```
//...
bool operator==(const vector& other) const{
  return size() == other.size() &&
    std::equal(begin(), end(), other.begin());
}
```

Dynamic memory allocation and deallocation

In general, we want to separate allocation of raw memory from construction of `T` objects. `operator new(size_t count)` attempts to allocate `count` bytes on the heap. The newly allocated memory is uninitialized. This is different from the `new` expression, `new T(Args)` or `new T[]()`, which performs both allocation and zero initialization (invokes the default constructor `T()`).

```
// ...
reference operator[](size_type idx){
  return m_data[idx];
}

const_reference operator[](size_type idx) const{
  return m_data[idx];
}

// precondition: !empty()
reference front(){ return (*this)[0]; }
const_reference front() const {
  return (*this)[0]; }
reference back(){ return (*this)[m_size - 1]; }
const_reference back() const{
  return (*this)[m_size - 1]; }
```

Listing 10

```
struct raw_deleter {
// only frees the memory, doesn't destroy objects
  void operator()(T* ptr) noexcept {
    operator delete(ptr,
      std::align_val_t(alignof(value_type)));
  }
};
using raw_ptr = std::unique_ptr<T, raw_deleter>;
raw_ptr allocate_helper(size_type new_capacity) {
  auto ptr = operator new(
    sizeof(value_type) * new_capacity,
    std::align_val_t(alignof(value_type))
  );
  return raw_ptr(static_cast<pointer>(ptr));
}
```

Listing 11

The memory subsystem on a modern CPU is restricted to accessing memory at the granularity and alignment of its word size. The CPU always reads at its word size (8 bytes on a 16-bit processor), so when you do an unaligned address access – on a processor that supports it – the processor is going to have to read multiple words. The CPU will read each word of memory that your requested address straddles. That's why it's important to always align custom types, when allocating memory.

The regular `new` operator only guarantees alignment upto `alignof(std::max_align_t)`; it works well for fundamental types. But, for custom types where

```
alignof(T) > alignof(std::max_align_t)
```

it would fail. C++17 supports an overloaded version of the `new` operator with alignment as an additional argument.

We introduce the helper functions `allocate_helper` and a custom deleter function object. We also declare a `raw_deleter` type alias. (See Listing 11.)

In `allocate_helper`, I chose to wrap the result of `operator new` into a unique pointer before returning to the caller. Again, this makes memory management automatic at the call-site.

Implementing reserve()

`reserve(size_type new_capacity)` increases the capacity of the vector(the total number of elements that the vector can hold without requiring reallocation) to a value that's greater or equal to `new_capacity`. If `new_capacity` is greater than the current `capacity()`, new storage is allocated, otherwise the function does nothing. (See Listing 12, next page.)

If `new_capacity > capacity()`, we must:

- Allocate a chunk `new_capacity * sizeof(T)` bytes large on the heap dynamically.
- Copy the existing container elements from the old storage area to the new block of memory.
- Destruct the elements in the old storage and deallocate the memory occupied.
- Update the `vector`'s `m_data` pointer and `m_capacity` field.

After the allocation, we want to copy the elements in the range `m_data[0...m_size-1]` to `ptr_new_blk`.

`copy_old_storage_to_new` is a helper function to copy `num_elements` from the memory location `source_first` to `destination_first`.

```
// Copies elements from old storage to new
// If T's copy/move ctor throws, the objects
// already constructed are destroyed and the
// exception is propagated to the caller.
void copy_old_storage_to_new(pointer source_first,
  size_t num_elements, pointer destination_first)
{
  if constexpr(
      std::is_nothrow_move_constructible_v<T>){
    std::uninitialized_move(source_first,
      source_first + num_elements,
      destination_first);
  }
  else{
    std::uninitialized_copy(source_first,
      source_first + num_elements,
      destination_first);
  }
}
void reserve(size_type new_capacity){
  if(new_capacity <= capacity())
    return;
  raw_ptr mem = allocate_helper(new_capacity);
  copy_old_storage_to_new(m_data, m_size,
    mem.get()); // can throw
  std::destroy(m_data, m_data + m_size);
  pointer new_ptr = mem.release();
  mem.reset(m_data);
  m_data = new_ptr;
  m_capacity = new_capacity;
}
```

Listing 12

C++17 introduced `std::uninitialized_copy` and `std::uninitialized_move` algorithms. `std::uninitialized_copy(first, last, d_first)` accepts a source range `[first,last)` and copies the elements from the source range to an uninitialized memory area beginning at `d_first`. The `std::uninitialized_move` algorithm uses move semantics.

The beauty of these uninitialized memory algorithms are that they are exception safe. If one of the `T(const T&)` constructors invoked by `uninitialized_copy` ends up throwing, then the objects it managed to create before the exception was thrown will be destroyed (in an unspecified order), before the exception leaves the function.

The type-trait `std::is_move_constructible_v<T>` is a meta-function that returns `true` , if the argument `T` is move constructible.

If `copy_old_storage_to_new` throws, `mem` will go out of scope and, being a smart pointer, it will automatically release `new_capacity` on the heap.

There's a general trick that you would have seen in all of this. Do not modify your object until you know you can safely do it. Try to do the potentially throwing operations first, then do the operations until you can mutate your object. You will sleep better, and the risks of object corruption will be alleviated.

Implementing resize()

The distinction between `resize()` and `reserve()` is that `reserve()` only affects the capacity of the container, whereas `resize()` modifies the size and capacity both.

The `resize(size_type new_size)` method (see Listing 13) resizes the container to contain `count` elements:

- If the `new_size` is equal to the current size, do nothing.
- If the current size is greater than the `new_size`, the container is reduced to its first `new_size` elements.
- If the current size is less than `new_size`, then additional default-constructed elements are appended.

```
void resize(size_type new_size){
  size_type current_size = m_size;
  if(new_size == current_size)
    return;
  if(new_size < current_size)
  {
    // Reduce the container to count elements
    std::destroy(m_data + new_size,
      m_data + m_size);
  }
  if(new_size > current_size)
  {
    reserve(new_size);
    // Default construct elements at indicates
    // [current_size,...,new_size-1]
    std::uninitialized_value_construct(begin()
      + current_size, begin() + new_size);
  }
  m_size = new_size;
}
```

Listing 13

How to think about adding elements to our container?

We will code up a `push_back(T&&)` member function that accepts a universal reference `T&&`. If `T` is move constructible, then the value will be moved. If `T` is copy constructible then the value will be copied.

The `emplace_back(Args...)` will take a variadic pack of constructor arguments, and then perfectly forward them to the constructor of a `T` object, that will be placed at the end of the container. A reference to the newly constructed object is returned by `emplace_back()`, for convenience, in case the user-code would like to use it right away.

We would like to first check whether the container is full. We have a dichotomy. If the container is full, we take the so-called slow path, else we take the fast lane.

push_back_slow_path(value)

In this case, we would like to grow our container; we allocate more memory, than what the container currently holds. We leave the memory uninitialized. Memory allocation, can of course, fail.

We then add the new value at the index `m_size`. Appending the new element may fail.

We copy/move construct the existing elements of the container from the old storage to the new block of storage.

If all three steps were successful, we deallocate the old storage and return it back before replacing the values in the member variables `m_data`, `m_size` and `m_capacity`.

If either of the last couple of steps fail, we free the newly obtained block of storage.

push_back_fast_path(value)

In this case, we simply copy/move construct `value` at the end of the container and update the size of the container.

Edge-case

Consider the following edge-case, where the `value` to be added is an element of the vector itself. If there is a reallocation, then the elements of the container are relocated to a new region. So, `value` might become a dangling reference.

```
dev::vector<int> vec{ 1 };
for (int i = 0; i < 10; ++i) {
  vec.push_back(vec.back());
  EXPECT_EQ(vec.back(), 1);
}
```

Our design takes care of this edge case. (See Listing 14, next page.)

Coding up emplace_back

Similar to **push_back**, **emplace_back** also appends an element to the end of the container. The only difference is, **emplace_back** constructs a **T** element in-place in the vector , using the constructor arguments of type **T**.

```
std::construct_at(mem.get() + m_size,
  std::forward<Args>(args)...);
```

Implementing pop_back()

pop_back() should call the destructor of the element at index **m_size - 1**. **std::destroy_at(T* p)** calls the destructor of the object pointed to by **p**. It is equivalent to **p->~T()**. We must not forget to decrement the size of the container.

```
void pop_back() {
  T* ptr_to_last = m_data + m_size - 1;
  std::destroy_at(ptr_to_last);
  --m_size;
}
```

Implementing insert(const_iterator position, It first, It last)

The **insert** function inserts the given value into the vector before the specified **position**, possibly using move-semantics. Note that, this kind of operation could be expensive for a vector, and if it is frequently used, it can trigger reallocation.

Our **insert** function will be generic enough with the following interface:

```
template<class It>
iterator insert(const_iterator pos,
                It first, It last)
```

It inserts the range **[first,last)** at position **pos** (immediately prior to element currently at **pos**).

I spent some time thinking about **.insert**, and drawing some quick diagrams helped me generalize the algorithm.

Step 1. We first determine if the elements in the range **[first,last)** can fit into the **remaining_capacity = capacity() - size()**. If **n** exceeds the **remaining_capacity**, the **excess_capacity_reqd** we require is **std::max(n - remaining_capacity,0)**. So, we invoke **reserve(capacity() + excess_capacity_reqd)**.

Step 2. Assume that we have sufficient room for the range **[first,last)**.

```
template<typename U>
void push_back_slow_path(U&& value){
  // allocate more memory
  size_type new_capacity =
    capacity() ? growth_factor * capacity() : 1;
  size_type offset = size();
  size_type new_size = m_size + 1;
  auto mem = allocate_helper(new_capacity);
  std::construct_at(mem.get() + m_size,
     std::forward<U>(value));
  try{
    copy_old_storage_to_new(m_data, m_size,
      mem.get());
  }catch(std::exception& ex){
    std::destroy_at(mem.get() + m_size);
  }
  pointer ptr_new = mem.release();
  mem.reset(m_data);
  m_data = ptr_new;
  ++m_size;
  m_capacity = new_capacity;
}
template<typename U>
void push_back_fast_path(U&& value){
  std::construct_at(m_data + m_size,
    std::forward<U>(value));
  ++m_size;
}
template<typename U>
void push_back(U&& value)
{
  if(is_full())
  {
    push_back_slow_path(std::forward<U>(value));
  }
  else{
    push_back_fast_path(std::forward<U>(value));
  }
}
```

Listing 14

How many elements should be copied from the **[begin(), end())** sequence to the raw memory block at the end of the container?

Consider the scenario, where the range **[first,last)** is smaller than **[pos,end)** – see Figure 2. In this scenario, the elements **[end()-n, end())** need to be copied or moved to the uninitialized memory.

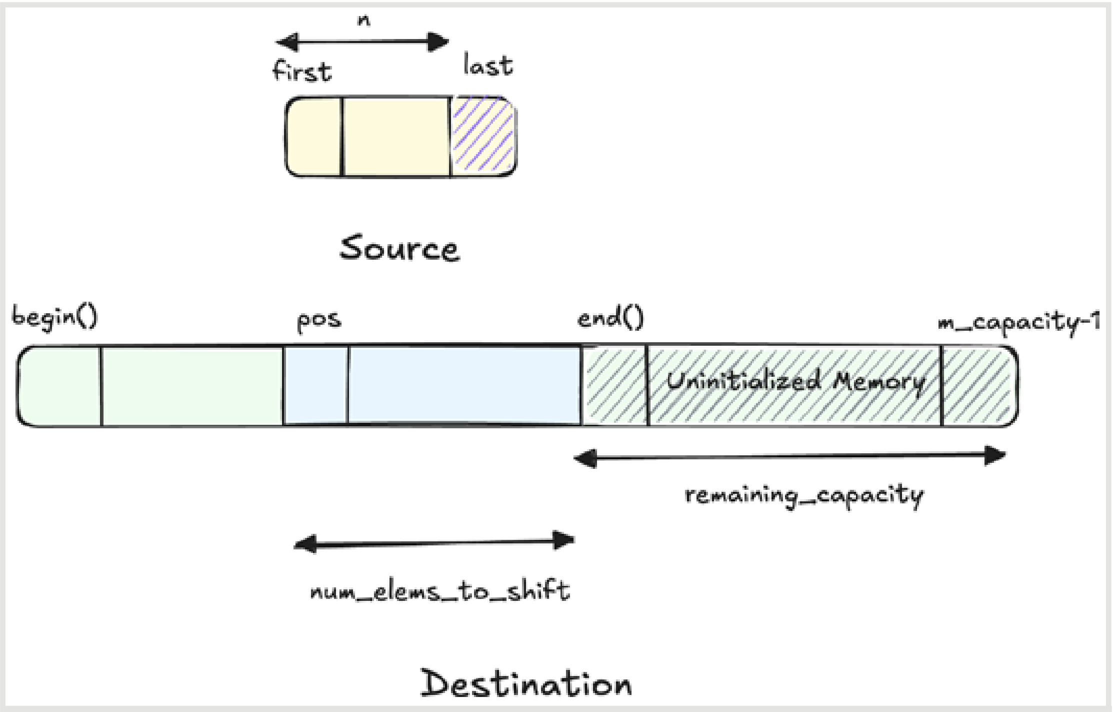


Figure 2

If there are elements to copy or move from **[pos,end())** sequence as a replacement to existing objects in the container (there could be none), how many should be there? Looking at Figure 3 (next page), the subsequence **[pos, end() - n)** will be mapped to **[pos + n, end())** upon insertion.

Consider the scenario where the range **[first,last)** is larger than **[pos,end)** (Figure 4, next page).

In this case, let’s define **num_tail** as the trailing number of elements from the source range **[first,last)** that would be copied/moved to uninitialized memory. Clearly, **num_tail = std::max(n - end() + pos,0)**. So, the tail **`[last-num_tail,last)`** will be mapped to **`[end(),end()+num_tail)`** upon insertion.

To make room for the insertion, the elements **`[pos,end())`** will have to be moved to **`[end() + num_tail, end() + num_tail + end() - pos)`**.

Listing 15 is a possible implementation based on our ideas above.

Conclusion

Implementing a custom **`vector<T>`** from scratch is a rewarding exercise that deepens understanding of C++ fundamentals.

The standard library implementation handles additional complexities I haven't addressed: custom allocator support, small object optimizations, and numerous other edge cases discovered through decades of production use.

Instead of pointer/size/capacity, we may use pointers: **`m_start`** , **`m_end`** and **`m_end_of_storage`**. While both layouts occupy 3 words (24 bytes on a 64-bit machine), **`end()`** is marginally faster, does not require pointer arithmetic and generates fewer assembly instructions.

However, the journey of building this container teaches invaluable lessons. You learn to think carefully about exception safety, understand the tradeoffs between copy and move operations, appreciate the elegance of algorithms like **`std::uninitialized_copy`**, and recognize why seemingly simple operations like **`insert()`** require careful reasoning about memory layout and iterator invalidation.

If you enjoyed this deep dive, I recommend exploring **`deque`**, **`std::inplace_vector`**, or the more complex associative containers. Each presents unique challenges and design decisions that will further sharpen your C++ skills. ■

You can find the complete source listing and unit tests online at https://compiler-explorer.com/z/Y6q1Tb3GK.

References

The deepest code review of the simplest data structure, vector: https://www.youtube.com/watch?v=GfIxO_vpM4g

libstdc++ vector test suite, https://gnu.googlesource.com/gcc/+/trunk/libstdc++-v3/testsuite/23_containers/vector

C++ Memory Management, by Patrice Roy, Packt Publishing.

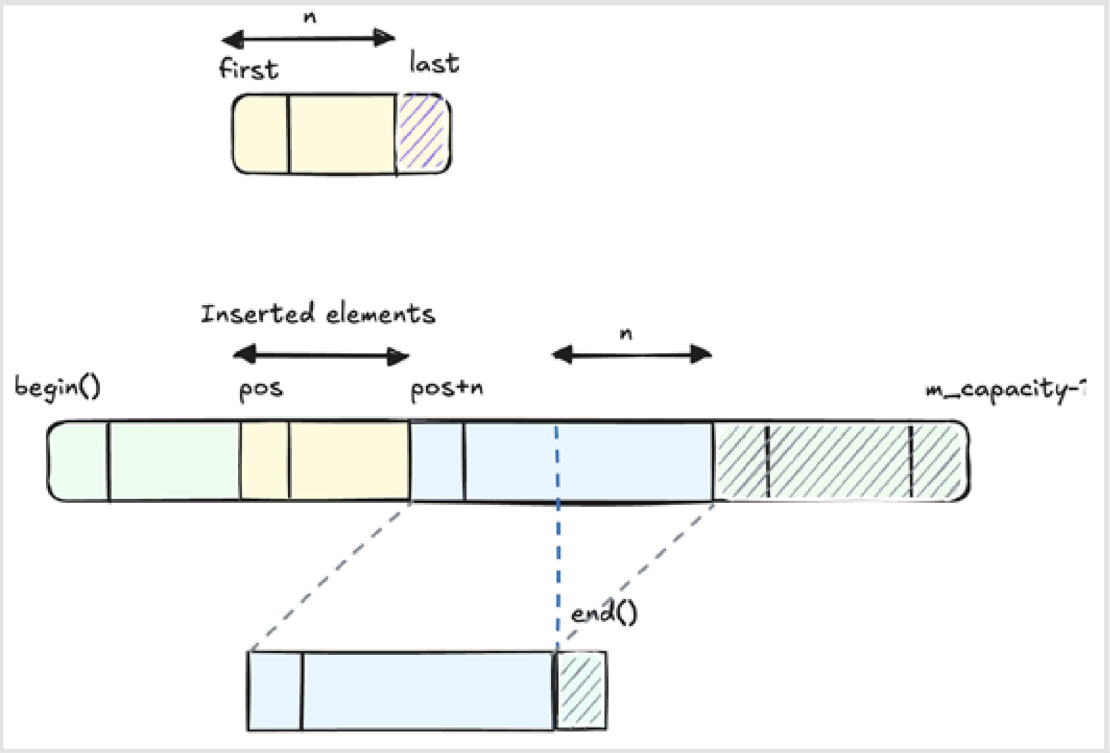


Figure 3

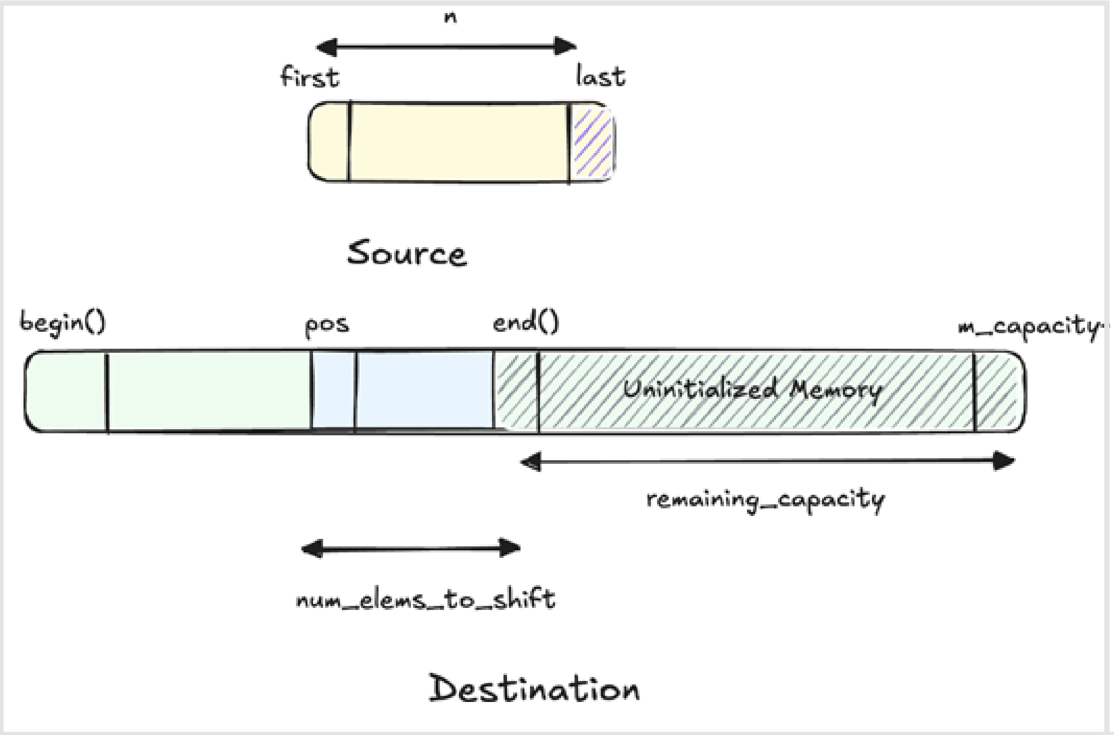


Figure 4

```
template<typename It>
iterator insert(const_iterator pos, It first,
    It last){
  auto pos_ = const_cast<iterator>(pos);
  auto first_ = first;
  auto last_ = last;
  if(first != last)
  {
    size_type offset = std::distance(begin(),
      pos_);
    size_type n = std::distance(first, last);
    size_type num_elems_to_shift
      = std::distance(pos_, end());
    size_type remaining_capacity
      = capacity() - size();

    dev::vector<T> temp;
    // handle self-referential insertion
    if((first_ >= begin() && first_ < end())
    && last_ > begin() && last_ <= end())
    {
      for(auto it{first_}; it!=last_; ++it)
        temp.push_back(*it);
      first_ = temp.begin();
      last_ = temp.end();
    }
    if(n > remaining_capacity)
    {
      size_type excess_capacity_reqd
        = std::max(n - remaining_capacity, 0uz);
      reserve(capacity() + excess_capacity_reqd);
      // The iterator pos is invalidated. Update
      // it.
      pos_ = std::next(begin(), offset);
    }
    // objects to displace (move or copy) from the
    // [begin, end()] sequence into raw
    // uninitialized memory
    if(n < num_elems_to_shift)
    {
      if constexpr(
        std::is_nothrow_move_constructible_v<T>)
      {
        std::uninitialized_move(end() - n, end(),
          end());
      }
      else
      {
        std::uninitialized_copy(end() - n, end(),
          end());
      }
    }else{
      size_type num_tail
        = std::max(n - num_elems_to_shift, 0uz);
      if constexpr(
    dev::vector<T> temp;
    // handle self-referential insertion
    if((first_ >= begin() && first_ < end())
    && last_ > begin() && last_ <= end())
    {
      for(auto it{first_}; it!=last_; ++it)
        temp.push_back(*it);
      first_ = temp.begin();
      last_ = temp.end();
    }
    if(n > remaining_capacity)
    {
      size_type excess_capacity_reqd
        = std::max(n - remaining_capacity, 0uz);
      reserve(capacity() + excess_capacity_reqd);
      // The iterator pos is invalidated. Update
      // it.
      pos_ = std::next(begin(), offset);
    }
```

Listing 15

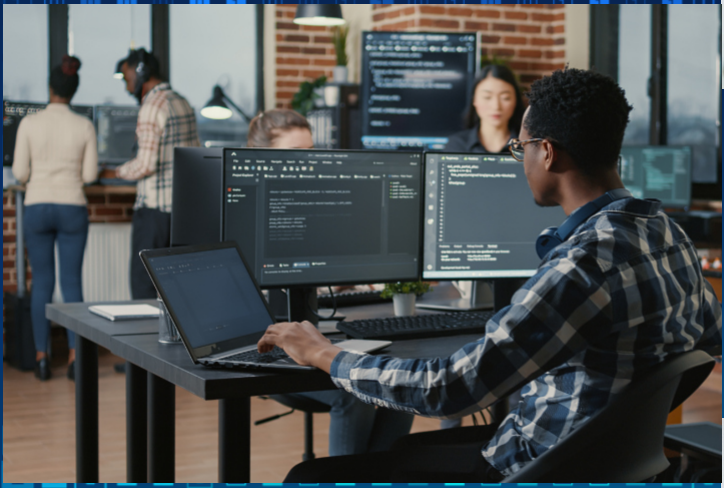
```
    // objects to displace (move or copy) from the
    // [begin, end()] sequence into raw
    // uninitialized memory
    if(n < num_elems_to_shift)
    {
      if constexpr(
        std::is_nothrow_move_constructible_v<T>)
      {
        std::uninitialized_move(end() - n, end(),
          end());
      }
      else
      {
        std::uninitialized_copy(end() - n, end(),
          end());
      }
    }else{
      size_type num_tail
        = std::max(n - num_elems_to_shift, 0uz);
      if constexpr(
        std::is_nothrow_move_constructible_v<T>)
      {
        std::uninitialized_move(pos_, end(),
          end() + num_tail);
      }
      else
      {
         std::uninitialized_copy(pos_, end(),
          end() + num_tail);
      }
    }
    // objects to displace (copy or move) from
    // [pos,end()] to the end of the container
    if(n < num_elems_to_shift)
    {
      if constexpr(
        std::is_nothrow_move_constructible_v<T>)
      {
        std::move_backward(pos_, end() - n,
          end());
      }
      else
      {
        std::copy_backward(pos_, end() - n,
          end());
      }
    }
    // objects from [first,last) to insert into
    // raw uninitialized memory
    const size_type num_tail
      = std::max(n - num_elems_to_shift, 0uz);
    if(n >= num_elems_to_shift)
    {
      if constexpr(
        std::is_nothrow_move_constructible_v<T>)
      {
        std::uninitialized_move(last_ - num_tail,
          last_, end());
      }
      else
      {
        std::uninitialized_copy(last_ - num_tail,
          last_, end());
      }
    }
    // objects to copy from [first,last) to pos
    if(n < num_elems_to_shift)
    {
      std::copy(first_, last_, pos_);
    }
    else{
      std::copy(first_, first_ + n - num_tail,
        pos_);
    }
    m_size += n;
  }
  return pos_;
}
```

Listing 15 (cont'd)

accu

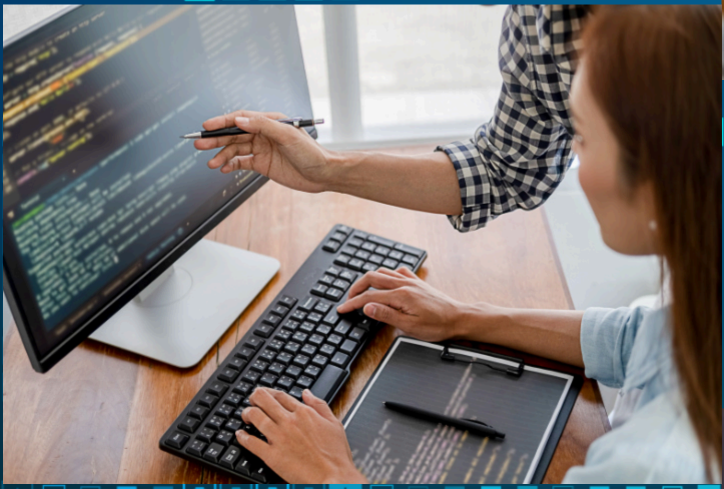
Professionalism in Programming

Professional development



Printed and online journals

Local groups



Join us!

Membership rates on our website.

Visit accu.org for details

accu

Professionalism in Programming

accu on sea

World-class conference

Where:

Leas Cliff Hall

Folkstone, Kent, UK

accu on sea

Jointly with C++ on Sea

Workshops:

15 & 16 June 2026

Conference:

17–20 June 2026

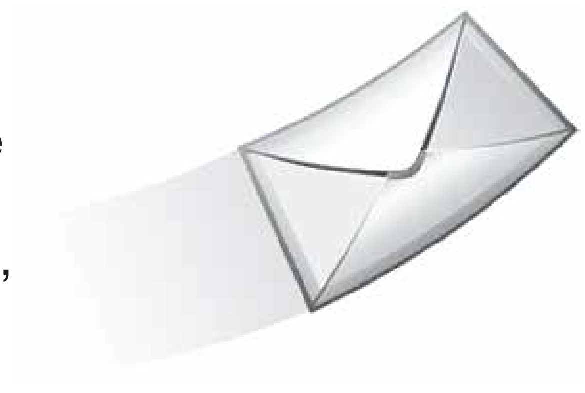
accu on sea

Visit accuconference.org for details

Early bird tickets are still available but may run out soon!

Letter to the Editor

Silas S. Brown wrote in following an article in the previous issue. The editor passed it to the author of that article (Andy Balaam), who has replied.



The 'letter'

Hi Andy, thank you for thoughtfully writing an ethical critique of LLMs in *Overload* 190. I share your concerns about what I call 'AI done wrong' but I also believe we can have 'AI done right'. (Analogy: solar power 'done wrong' is destroying nature with huge solar farms that disorient birds; 'done right' means using recycled silicon on roofs.)

Environmental impact

Of interest is a finding in your O'Donnell25 source that an 8 billion parameter model took less than 2% of the energy of a 405 billion parameter model. This is encouraging in light of moves toward architectures that avoid energising the whole model at once, such as the Chinese *Kimi K2* model which, although having a trillion (1000 billion) parameters, organises them in a Mixture-of-Experts (MoE) architecture with only 32 billion of them active at any time. Informally, 'we don't need the quantum-physics parts when you're asking for cookery advice'. Extrapolating from that source's figures, inferencing on Kimi K2 should take under 8% of the power that Llama 3.1 took, and it's a more advanced open-source model.

Chinese engineers have also found a way to train their models on a much tighter budget, although exact figures are hard to find and verify. Much Chinese AI runs on Aliyun's cloud, which in 2024 claimed to use 56% clean energy, aiming for 100% by 2030. O'Donnell25 noted that carbon intensity varies by exact location and time of day, which could mean even today's infrastructure might let us train an LLM in a much more 'environmentally friendly' way simply by being careful about where and when loads run. I agree more transparency is needed though.

The same source also comments that:

> …average individual users aren't responsible for all this power demand. Much of it is likely going toward startups and tech giants testing their models, power users exploring every new feature, and energy-heavy tasks like generating videos.

I'm particularly concerned about videos, with 10 seconds of AI-generated video being estimated to take over 500 times the energy per query of that wasteful 405 billion parameter dense (non-MoE) model. If you look at a text query result on a monitor while thinking about it for 3 minutes, your monitor has probably taken more energy than the query even at 405B, but this excuse disappears with video – I hope not everyone will want to do that. Technically, video is off-topic if we're restricting our discussion to LLMs and not other kinds of AI, but several platforms now offer both.

Silas S. Brown is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

Exploiting and traumatising training workers

Your Stahl25 source found an instance where an intermediary company called SAMA absorbed some 85% of OpenAI's cash instead of passing it on to the actual workers. This is terrible, but the villain here is clearly not OpenAI but SAMA and the journalist was doing a good job to expose it and hopefully bring about change (the source says the projects mentioned were closed down). If OpenAI is paying much more than workers are receiving, they simply need to check their supply chain more carefully, just as the manufacturing industry is increasingly being pushed to do if it's the 'go-betweens' that are the problem.

Also in Stahl25 is an instance of someone training Meta's content filter on awful posts. Yes, that was horrible but sad to say it's off-topic if we are limiting ourselves to a discussion of LLMs, since that vile job was not for an LLM but for another kind of 'AI'. From my reading of how RLHF response-ranking works, I believe LLM training jobs are tedious but not traumatic. I'll update that view if a report emerges that specifically shows people being traumatised by LLM training, which is not the same as content-filter training.

Danger of using AI results

This is what I'm most concerned about as humans have huge automation bias ('computer says no'), but I believe that, with more research, we could learn exactly where LLMs are likely to be an asset versus a liability.

The 2024 *New York Times* article (Roose24) unfortunately fails to make a strong case that the LLM was the cause of Sewell Setzer III tragically ending his life. If that LLM had not been available, the words quoted in the article could have been written by any young human player who was not a professional therapist: it tried to tell him not to proceed, and then failed to pick up on a later hint that he was seeking validation using other words. This activity was (according to reports) being conducted against the specific advice of a real therapist. This is not a comment on their legal case which may be stronger; I'm simply saying the initial reports didn't do a very good job of showing us how the LLM is 'reponsible' for this tragedy. There are other cases (such as that of Juliana Peralta) and the BBC reported Character.AI made themselves 18+ on 25 November 2025 although it's unclear how good the enforcement is.

The Hill25a example is far more concerning, firstly because it relates to a more mainstream LLM (harder to get ChatGPT to make themselves 18+) and secondly it's a clearer case: ChatGPT became accidentally stuck in a suicide-reinforcing loop after a long conversation (long conversations are not so well tested) and that's why I think vulnerable people need some supervision when having this kind of conversation with a probabilistic model. The 'delusional spiral' failure mode has reportedly been significantly reduced in the more recent transition from ChatGPT 4 to ChatGPT 5 but when I say 'reportedly' here I'm looking at a non-peer-reviewed study on the shared blog LessWrong: I still want to see more human supervision of these games.

Unfair use of creative work

Copyright law does allow indexing and lexicography: you may write a dictionary of words seen in the books you read without it being copyright infringement, and model training is similarly supposed to 'average' its input so no one source can be reproduced from the model's 'knowledge' of how words and concepts are related to each other over a large collection of sources. This is also important for accuracy if we assume the training data has been curated such that the knowledge worth remembering is that on which many sources agree (which is a big assumption), and is the reason why LLMs don't tend to be able to remember your homepage without looking it up even if you've seen their bots crawl it.

But there are concerns of 'overfitting' where models memorise sources too precisely, such as the New York Times example in Carson25, and this needs to be (and is being) looked into.

Other reasons

Overpromised productivity gains: as has sadly been the case with many technologies. Pushing 'AI' just for the sake of it is never good.

Mental atrophy: The Black25 source's report of doctors forgetting how to identify cancer makes me think of a design decision in the construction of Norway's Laerdal Tunnel: drivers are prevented from becoming drowsy by placing gentle curves in the road instead of making it completely straight. If AI assistance is getting things right most of the time, perhaps we should throw in a few known defects to double-check the human is not asleep at the switch? This particular example is off-topic for LLMs since it's image classification, but we always have needed people to become more skilled at evaluating 'search results'.

Excuse to cut jobs: Correct but sadly irrelevant because companies will use any excuse to cut jobs anyway. Over the years I've lost jobs due to merging and acquisition, outsourcing, random relocation requirements, service obsolescence, and client cancelling project over unexpected trivial-patent lawsuit, and I've seen friends' job losses blamed on the thoughts of the President of the USA, so the fact that my most recent layoff justification included the words 'AI strategy' doesn't mean much: if it wasn't that, it would have been something else.

I'm more concerned about the jobs that are never created: startups trying to use 'AI' instead of developers, setting themselves up for problems later. (Source: informal conversations with young founders at Cambridge networking events who show me Web platforms they made in Cursor and confidently say they won't need developers. Carla's data shows a 62% drop in startup hiring between January 2022 and January 2025, and hiring is not rising with funding.) This isn't to say 'AI' itself is bad, just it's badly used. Sad to say this is similar to earlier trends of low-quality outsourcing. There might be some 'nonjudgmentally fix the founder's AI mess' jobs in surviving startups. I question business schools' teaching 'minimum viable product' when people fail to catch that middle word 'viable'.

Thanks again for bringing up this important subject.

Silas

References

The references that Silas refers to in his letter are from the original article. For convenience, they are replicated here.

[Black25] 'AI Eroded Doctors' Ability to Spot Cancer Within Months in Study', *Bloomberg*, published 12 August 2025 at https://www.bloomberg.com/news/articles/2025-08-12/ai-eroded-doctors-ability-to-spot-cancer-within-months-in-study

[Carson25] David Carson, 'Theft is not fair use', published 21 April 2025 at https://jskfellows.stanford.edu/theft-is-not-fair-use-474e11f0d063

[Hill25a] Kashmir Hill, 'A Teen Was Suicidal. ChatGPT Was the Friend He Confided In' *New York Times*, updated 27 August 2025 at https://www.nytimes.com/2025/08/26/technology/chatgpt-openai-suicide.html

[O'Donnell25] James O'Donnell and Casey Crownhart, 'We did the math on AI's energy footprint. Here's the story you haven't heard', *MIT Technology Review*, published 20 May 2025 at https://www.technologyreview.com/2025/05/20/1116327/ai-energy-usage-climate-footprint-big-tech/

[Roose24] Kevin Roose, 'Can A.I. Be Blamed for a Teen's Suicide?', *New York Times*, published 23 October 2024 at https://www.nytimes.com/2024/10/23/technology/characterai-lawsuit-teen-suicide.html (subscription required)

[Stahl25] Lesley Stahl, 'Labelers training AI say they're overworked, underpaid and exploited by big American tech companies' *CBS News*, updated 29 June 2025 at https://www.cbsnews.com/news/labelers-training-ai-say-theyre-overworked-underpaid-and-exploited-60-minutes-transcript/

[No AI was used in the writing of these words. The em-dashes in my writing probably helped teach the LLMs to do it.]

Andy's reply

Thank you for the thoughtful and detailed response.

I'm sure you are right in several cases, and I'm fairly sure I will have to soften my approach as this technology becomes more integrated into our lives. I do hope that the magical thinking around it will reduce as that happens.

In general I am deeply sceptical about all the activity coming from the self-obsessed and delusional billionaires who are running the tech industry. If this were a grass-roots movement I would have more hope about it being useful without abusing people. As it is, I strongly suspect its main use case will be to make the existing harmful social media apps even more addictive.

If you read something in *Overload* that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it? We'd love to hear from you!

Restrict Mutability of State

Changing state can cause problems in software. Kevlin Henney reminds us that when it is not necessary to change, it is necessary not to change.

What appears at first to be a trivial observation turns out to be a subtly important one: a great many software defects arise from the (incorrect) modification of state. It follows from this that if there is less opportunity for code to change state, there will be fewer defects that arise from state change!

Perhaps the most obvious example of restricting mutability is its most complete realization: immutability. A moratorium on state change is an idea carried to its logical conclusion in languages that embody a pure expression of the functional paradigm, such as Haskell and Clojure. But even the modest application of immutability in other programming languages and paradigms has a simplifying effect with architectural implications and benefits.

Immutability makes it easier to reason about state. If an object can't have its state changed when your back is turned, that's one less thing to track, one less thing to worry about, one less thing that needs to be remembered, and so one less thing that can be forgotten or overlooked. If an object is immutable it can be shared freely across different parts of a program without concern for aliasing problems or synchronization surprises.

Programmers often assume thread-safety is necessarily associated with locking. This assumption comes from focusing on what is being locked rather than appreciating what locking is supposed to protect something from. You don't use locks because you wish to prevent concurrent access to an object by other threads; you use locks to prevent concurrent access to an object whose state may change. What matters here is the possibility of change. Without change, there is no need to lock.

An object that does not change state is, therefore, inherently thread-safe and free to access – there is no need to synchronize and guard against state change if there is no state change! An immutable object does not need locking or any other palliative workaround to achieve safety.

> A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention. [Carmack12]

Depending on the language and the idiom, immutability can be expressed in the definition of a type or through the declaration of a variable. For example, Java, JavaScript, and .NET's `String` class represents objects that are essentially immutable — if you want another string value, you use another string object. Immutability is particularly suitable for value objects in languages that favour predominantly reference-based semantics.

By contrast, the `const` qualifier in C and C++ and, more strictly, `immutable` in D and `constexpr` in C++, constrain mutability through declaration. Such qualification restricts mutability in terms of compiler-enforced access rights, typically expressing the notion of read-only access rather than necessarily full immutability.

Perhaps a little counter-intuitively, copying offers an alternative technique for restricting mutability. In languages offering a transparent syntax for passing by copy, such as C#'s `struct` objects and C++'s default argument-passing mode, copying value objects can greatly improve encapsulation and reduce opportunities for unnecessary and unintended state change. Passing or returning a copy of a value object ensures that the caller and callee cannot interfere with one another's view of a value.

But be aware that an approach reliant on copying is not recommended if the passing syntax is neither easy nor transparent. If programmers have to make special efforts to remember to make a copy, such as explicitly calling a clone method, they are also being given the opportunity to forget to make a copy. Far from being a simplification, it becomes tedious and error-prone, a complication that is easy to overlook, a bug waiting to happen.

In general, make state and any modification to it as local as possible. For local variables, declare as late as possible, when a variable can be sensibly initialized. Try to avoid broadcasting mutability through public data, global and class static variables (which are essentially globals with scope etiquette), and modifier methods. Resist the temptation to mirror every getter with a setter.

> Encapsulation is important, but the reason why it is important is more important. Encapsulation helps us reason about our code. In well-encapsulated code, there are fewer paths to follow as you try to understand it. Encapsulation isn't an end in itself; it is a tool for understanding. [Feathers04]

The relationship between immutability and encapsulation is often overlooked. For (counter)example, a common code smell is methods or properties that return references to collections used as private representation. Not only does this expose and tie callers into a dependency on the private representation choice, it also grants them inappropriate — and often unintended – write-access to state. In addition to traversal and query, they can now modify the collection content, breaking any invariant protection that encapsulation was supposed to offer. That no-nulls and no-duplicates guarantee? No longer a guarantee. Anyone can insert nulls and duplicates once you've invited them in!

> Never ever invite a vampire into your house. And why? Because it renders you powerless. [LostBoys]

Instead of handing out the whole collection, which allows others to undermine an object's integrity, consider offering an iterable or streamable view of the elements. This takes different forms in different languages and libraries: Iterator or `Stream` in Java; `IEnumerator`, `IEnumerable`,

Kevlin Henney is an independent consultant, speaker, writer and trainer. His development interests include programming languages, software architecture and programming practices, with a particular emphasis on unit testing and reasoning about practices at the team level. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*. He is also editor of *97 Things Every Programmer Should Know* and co-editor of *97 Things Every Java Programmer Should Know*.

Much code that we consider complex is considered complex because of the mental highwire act we perform when trying to understand what (the hell) is going on

and LINQ in C#; iterators and ranges in C++; iterators, iterables, and `__iter__` in Python. By restricting callers to views [Bharambe15], they can look but they can't touch and, therefore, can't break.

Much code that we consider complex is considered complex because of the mental highwire act we perform when trying to understand what (the hell) is going on. The more that things can change – and the more that changes depend on one another – the harder it becomes to reason about them correctly and confidently. Thinking about code should not be a circus performance. The name for code we can't reason about? Unreasonable. Immutability makes code more reasonable.

When it is not necessary to change,
it is necessary not to change.
~ Lucius Cary

Restricting mutability of state is not, however, some kind of silver bullet you can use to shoot down all defects. The resulting code simplification and improvements in encapsulation nonetheless make it less likely you will introduce defects, and more likely you can change code with confidence rather than trepidation. ■

References

[Bharambe15] Ashwin Bharambe, Zack Gomez, Will Ruben 'Under the hood: Building Moments', posted 15 June 2015 on Engineering at Meta, available at https://engineering.fb.com/2015/06/15/android/under-the-hood-building-moments/.

[Carmack12] John Carmack, 'In-depth: Functional programming in C++', posted 30 April 2012 on Gamasutra, available at https://web.archive.org/web/20190122134815/http://gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php

[Feathers04] Michael Feathers (2004) *Working Effectively with Legacy Code*, published Pearson.

[LostBoys] *The Lost Boys* (film), details at https://www.imdb.com/title/tt0093437/

The Results of the 2025 Favourite Articles Survey

In *CVu*, there is a 4-way tie for 1st place:

- 'C#'s Unsung Heroes: the Value Tuple' by Steve Love, published in *CVu* 37.2 in May 2025 and available to members at https://accu.org/journals/cvu/37/2/love-2/
- 'Beginners' Python on Amazon Alexa' by Silas S. Brown, published in *CVu* 37.3 in July 2025 and available to members at https://accu.org/journals/cvu/37/3/brown-1/
- 'Long-running Actions on GitHub' by Silas S. Brown, published in *CVu* 37.3 in July 2025 and available to members at https://accu.org/journals/cvu/37/3/brown-2/
- 'What M3GAN Can Tell Us About Software Engineering' by Silas S. Brown, published in *CVu* 37.4 in September 2025 and available to members at https://accu.org/journals/cvu/37/4/brown/

In *Overload*, there is a winner and a 3-way tie for 2nd place

The winner:

- 'Concurrency Flavours' by Lucian Radu Teodorescu, published in *Overload* 190 in December 2025 and available at https://accu.org/journals/overload/33/190/teodorescu/

Second place:

- 'Using Senders/Receivers' by Lucian Radu Teodorescu, published in *Overload* 185 in February 2025 and available at https://accu.org/journals/overload/33/185/teodorescu/
- 'Bit Fields, Byte Order and Serialization' by Wu Yongwei, published in *Overload* 185 in February 2025 and available at https://accu.org/journals/overload/33/185/wu/
- 'Local Reasoning Can Help Prove Correctness' by Lucian Radu Teodorescu and Sean Parent, published in *Overload* 188 in August 2025 and available at https://accu.org/journals/overload/33/188/teodorescu-parent/

Congratulations to the winners, and thank you to everyone who took the time to vote!

Afterwood

We spend a lot of time hammering away. Chris Oldwood reminds us that spending time mulling things over can also be productive.

While on a recent trip to an alternate recycling centre with my daughter, she noticed they had a bunch of smaller specialist bins separate from the huge containers used for the main bulk of waste. There were a couple for CDs but, more interestingly, there was one filled with books. She started to have a rummage and then called over to me as she noticed a programming book in amongst the James Pattersons. As expected, it was one of those ‘Learn to Program in 24 Minutes’ style books from the ’90s. However, as I dug deeper, I noticed a whole bunch of far more useful programming books, and all in pristine condition. Although slightly incensed that someone decided to bin them rather than take them directly to a charity shop, I was grateful they hadn’t disposed of them in the generic waste containers to be tossed on the landfill or incinerated. (Hopefully, the recycling centre will ensure they eventually find another bookshelf to live out their days.)

Anyway, one of the eleven books I liberated was Andy Hunt’s 2008 classic Pragmatic Thinking & Learning – Refactor Your Wetware, which had been on my ever-growing wish-list for years. Being excited about yet another non-technical book probably adds further weight to J.B. Rainsberger’s observation about a programmer’s bookshelf largely consisting of books on applied psychology once they reach a certain level of proficiency.

The book uses the model of the brain from Dr Betty Edwards which introduced the terms L-mode and R-mode as an alternative to the older, more simplistic left-brain/right-brain version. R-mode is the background asynchronous mode which chews over problems while you’re doing more mundane tasks like going for a walk, doing the dishes, or having a shower. This is the basis for the age-old advice about stepping away from the keyboard when you have a problem for which there is no immediate solution. In fact, just this very morning I awoke to discover that I’d worked out in the night while fast asleep that I could replace 13 lines of code in my colleague’s PR with just 2. (Something didn’t feel right when reviewing it yesterday, but I couldn’t put my finger on it.)

Being a contractor it’s not uncommon to be forced to take the two weeks off at Christmas to save the company money, and because I suspect we can’t be trusted for some reason. (Maybe they read my previous Afterwood, where I admitted to Santa that I might have played a bit too much Doom during Christmas of ’93.) This practice hasn’t bothered me personally as I’m happy to take the time off and spend it with my own family, along with taking a welcome break entirely from programming. That’s not entirely true of course because it just gives my brain’s R-mode a chance to start chewing over the backlog of less immediate issues instead. Even when we’re supposedly ‘off the clock’ our brain still manages to find some background computation to amuse itself.

Consequently, two years ago I found myself thinking about thinking, which led to my 2024 opening piece ‘Thought Experiments’. Clearly I hadn’t spent enough time thinking about it because I realised this Christmas that I forgot to cover the ‘when’, which was no doubt asynchronously triggered by a conversation about the 1987 book Peopleware by Tom DeMarco & Tim Lister. (The latter of which I got to chat to in the bar at the ACCU 2012 conference.)

That book is legendary, and contains one of my favourite stories:

> One day, while Wendl was staring into space pondering problems of extreme complexity with his feet propped up on the desk, their boss came in and asked, “Wendl! What are you doing?” Wendl said, “I’m thinking.” And the boss said, “Can’t you do that at home?”

This story immediately resonated with me as I read it not long after having a conversation with my project manager where he was describing how ridiculous it was that someone he knew was paid for his ‘thinking time’. While not as explicit, this opinion was undoubtedly shared by other project managers I had worked with in the past and who probably felt that typing was the only true measure of productivity and value.

In the intervening couple of decades, more and more teams I’ve worked in have adopted an agile way of working where there is a drive to try and distil every bit of work into tiny units, meaning that any ‘hammock time’ you might want to think things over is timeboxed and scheduled on the Scrum board. If Wendl was in a corporate agile team today the Scrum Master would likely be interrogating him every morning to ask him to estimate how much longer his thinking was going to take and when he was going to get back to ‘product work’.

With the focus so firmly on solving ever smaller problems and hoping that the right design will eventually emerge there is no time to simply stand back and take in the bigger picture. There is time in the diary for a retrospective on how the team delivers, but where is the time for reflection on the architecture and design? Who is watching the evolution of the codebase and thinking about the direction it’s heading, and whether that really fits in with the intended direction of travel? Changes shouldn’t be held to ransom by pure speculation but as plans solidify so does the opportunity to make smaller course corrections instead of taking sharp turns.

I’ll freely admit that I resent the attitude of that project manager who scolded Wendl, and those who have propagated the same opinion. With L-mode constantly engaged during the time in the office, R-mode only gets a look-in at home – code by day, design by night. Fortunately, with the rise of remote working in the last 5 years my R-mode is finally getting the opportunity to see more daylight.

Wandering over to the sink in the office to wash up the cups would likely be met with awkward questions, but at home my actions are less suspicious and R-mode can safely engage itself. The danger then is picking the wrong mundane task – packing the dishwasher might seem trivial but is effectively a game of Tetris, and pairing socks is akin to Candy Crush. Maybe it’s best to play it safe and just put my feet up, I wouldn’t want to overthink it. ■

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

Join ACCU

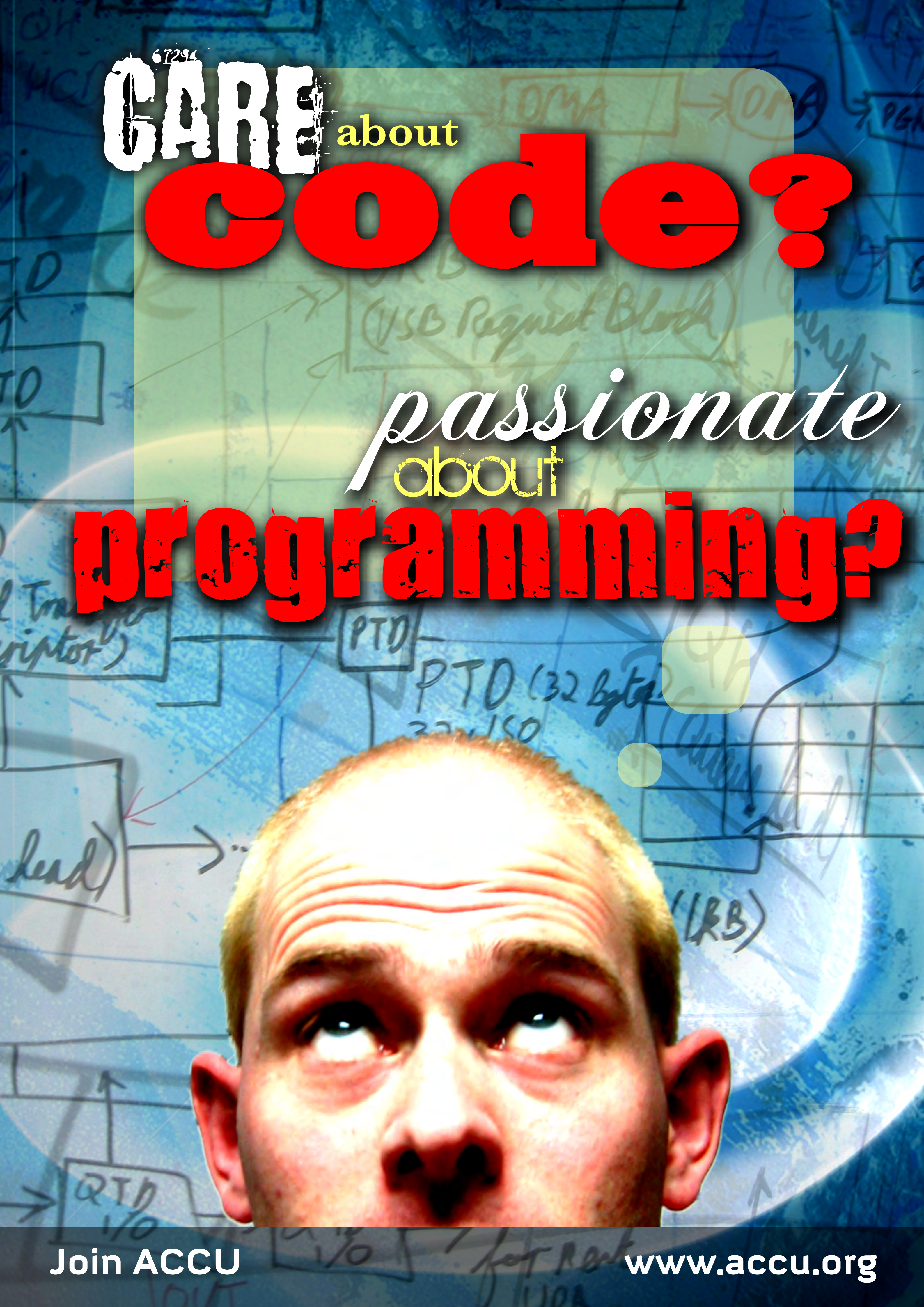
Run by programmers for programmers, join ACCU to improve your coding skills

- 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


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