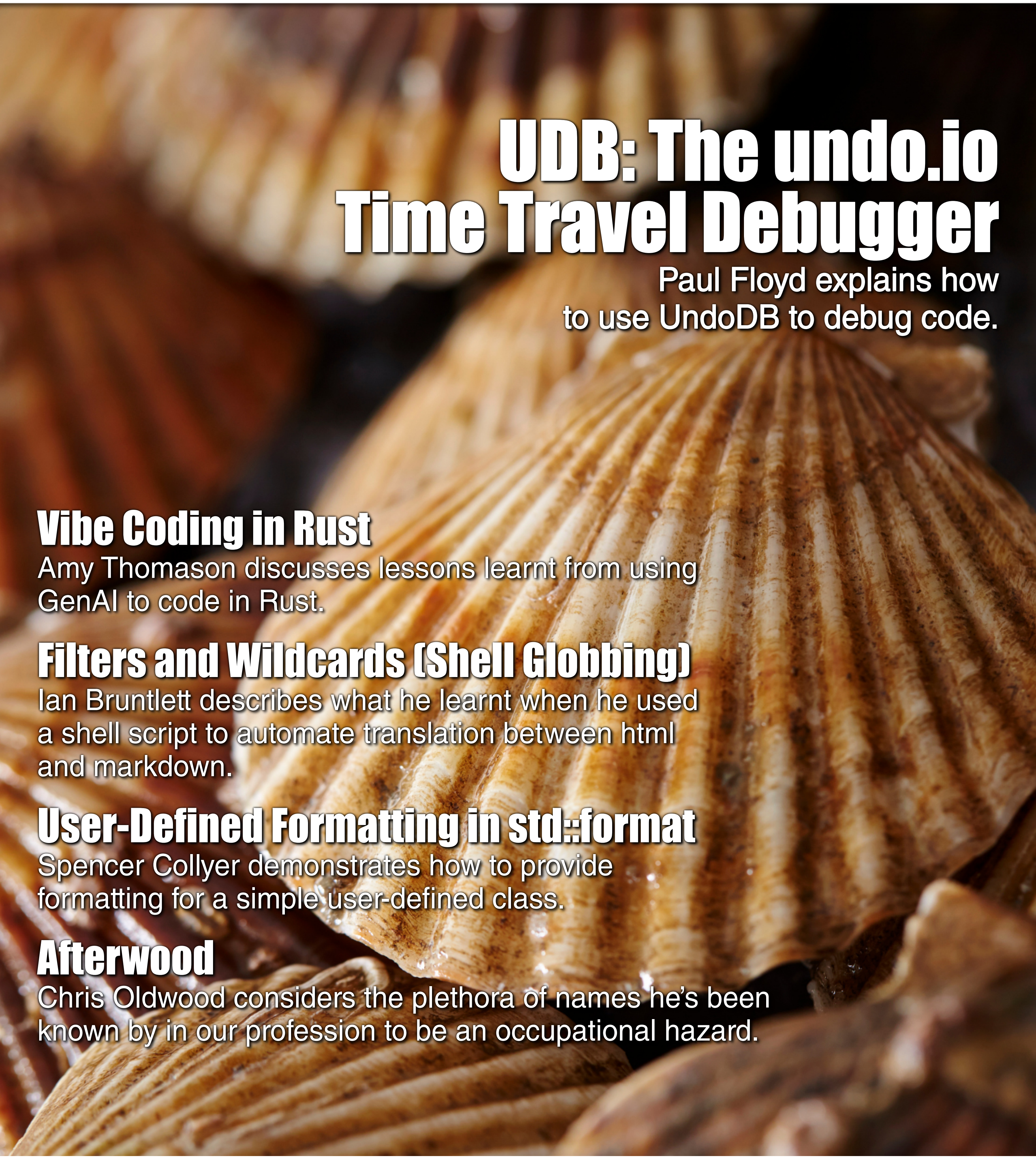
overload 189

OCTOBER 2025 £4.50



UDB: The undo.io Time Travel Debugger

Paul Floyd explains how to use UndoDB to debug code.

Vibe Coding in Rust

Amy Thomason discusses lessons learnt from using GenAI to code in Rust.

Filters and Wildcards (Shell Globbing)

Ian Bruntlett describes what he learnt when he used a shell script to automate translation between html and markdown.

User-Defined Formatting in std::format

Spencer Collyer demonstrates how to provide formatting for a simple user-defined class.

Afterwood

Chris Oldwood considers the plethora of names he's been known by in our profession to be an occupational hazard.

A magazine of ACCU

ISSN: 1354-3172

accu

Professionalism in Programming

Professional development
World-class conference

Printed journals
Email discussion groups

Individual membership
Corporate membership

Visit accu.org
for details

October 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 Mnimage from
Adobe Stock Photos.

ACCU

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

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

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

4 Vibe Coding in Rust
Amy Thomason discusses lessons learnt from using GenAI to code in Rust.

7 UDB: The undo.io Time Travel Debugger
Paul Floyd explains how to use UndoDB to debug code.

10 Filters and Wildcards (Shell Globbing)
Ian Bruntlett describes how he used a shell script to automate translation between html and markdown, and what he learnt.

13 User-Defined Formatting in std::format
Spencer Collyer demonstrates how to provide formatting for a simple user-defined class.

20 Afterwood
Chris Oldwood considers the plethora of names he's encountered for our profession: an occupational hazard.

Copy deadlines

All articles intended for publication in *Overload* 190 should be submitted by 1st November 2025 and those for *Overload* 191 by 1st January 2026.

Copyrights and trademarks

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

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

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

Ignore Rants

Programmers often argue, sometimes based on a lack of knowledge. Frances Buontempo explores why people rant, and when it’s OK.

Getting sucked in to a good old rant, either trying to set the world to rights or simply letting off steam, can be useful. But off-putting. Possibly even derailing you from important tasks, like writing an editorial. (Yes, yet again, another excuse from Yours Truly). Some people start ranting because they don’t understand why certain processes or techniques exist. Why would you write code and put it in version control when you could simply use a strategic spreadsheet? If you’re not used to collaborating with people, version control and the like may seem over the top. Such ranting is a sign of frustration, often borne of ignorance, at having to adopt a new process.

Many people resist change, especially if the current process works for them. However, some change is worth embracing, for example some of the changes to C++ over the years. Simple things like range-based `for` loops or structured bindings make life easier. Some companies are stuck on slightly older versions of the standard because an upgrade often takes time. Even though C++ strives to be backwards compatible, new warnings often appear if you try a newer compiler. I’ve been involved in some upgrades to newer language versions and something surprising always seems to happen. Usually a nasty surprise, which makes everything take much longer than expected. But sometimes the change helped us find causes of bugs we had been pondering for a long while. Change can be difficult, but useful. And we got to use newer features. Worth it in the end.

‘Modern’ C++ is so much better than older versions, but some new and perhaps over-complicated things have been introduced in C++ too. Can you write a coroutine without needing to look up what’s needed? I can’t. But I can now look at my *Learn C++ by Example* book [Buontempo24], because I know there’s a chapter in there that will help. Thank you to everyone who helped me learn how to use coroutines, especially Phil Nash’s *C++ On Sea* workshop in 2023 [Nash23]. If you missed that, his ‘C++ Coroutines: Gods from the machine’ talk is a great resource too [Nash25]. Change usually means you need to learn a new way to do stuff. That can be a challenge, but if you can find the right resources, or better someone to show you how, you’re off to a good start. It’s OK to have a brief “Oh no, I’m back to knowing nothing” to begin with. We’ve all been there, and are aware of the Dunning-Kruger effect, I presume [Wikipedia-1]. If you have limited competence, you might think you know it all. And the reverse effect notes if you have more competence, you are more likely to be aware of your weak spots. When faced with change, ranting and refusing to move with the times might not be the best option.

Some rants are entirely justified, or manifest as a silent scream. Has your machine done an auto-update recently, which failed? Ask our esteemed *CVu* editor about his laptop. Fortunately, he manage to retrieve his product key from bit locker, and persuade the machine to boot up again. Nobody wants to see a BSD (blue screen of death). If you don’t let off steam when you are frustrated, you will likely implode at a later date, possibly over something tiny that doesn’t really matter. Some things are simply annoying, and maybe result from a careless update or lack of testing. It’s OK to get angry, as long as you can move on and find a workaround or Plan B.

So much software is rubbish. There I said it. And I include some code I have previously written. However, I’m thinking about various retail websites. How many times do you need to accept cookies if you go to a certain DIY website to buy some building materials? Clue: more than once. How many times do address finders fail, especially if like us, you don’t have a house number? It’s no surprise people tend to go, “Ugh!” when I tell them I’m writing a book about computers. (I just ‘finished’ my final chapter of my latest book [Buontempo26]). People like the idea of a book, but in the main recoil at the mention of computers. Many people claim they just can’t get the hang of technology. Trying to use a badly designed website doesn’t mean you are bad at computers. If you design UIs, just watching someone use your product for a few minutes often reveals problems you can easily fix. Some people are very good at UIs, but evidence might suggest they are few and far between. Thank you for listening. Now I’ve got that rant out, I can move on.

Talking of moving on, people are still hailing Rust as a successor language to C++. We’re moving with the times, so have an article in this edition to give you a glimpse of some basics if you’ve not tried Rust yet. People claim ‘Rust is safer than C++’. If I search that phrase, the first hit says “Rust eliminates many common bugs that plague C++ development.” Eliminates? Common? Plagues? Hyperbole?! Maybe. I suspect some of the anti-C++ diatribes are based on wishful thinking. You can, of course, write `unsafe` code in Rust (yes, `unsafe` is a Rust keyword) [Rust]. I have nothing against Rust, and have yet to try it properly. I don’t like it when people tell me C++ is terrible, though. I feel the need to defend C++. It’s flawed, like any language. It’s hard to learn and teach. But, it’s powerful and you can use it for almost anything. And yes, you might have buggy and potentially unsafe code. But I am certain that is true of any language. I guess I need to ignore rants and try to think straight, no matter how contentious people are being.

Ignorance means not knowing, so can be fixed with knowledge. I say ‘can be’. Some ignorance is willful, and even has a legal definition [Wikipedia-2]:

> When a person seeks to avoid civil or criminal liability for a wrongful act by intentionally keeping themselves unaware of facts.

I could rant about people who stick their heads in the sand and choose to ignore evidence. Such deliberate selective hearing is down-right dangerous

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.

at times. Just consider anti-vaxxers and recent measles outbreaks. Nothing new: anti-vaxxers have existed since the first smallpox vaccine in 1796 [NIH]. You can't reason with unreasonable people, so I'll park that train of thought. You can spot your own ignorance and try to fix it. How do you find reliable sources for increasing your knowledge? Some might claim the internet has made the situation worse, but I am certain people complained from the moment we had a printing press that pamphlets and the like might contain untruths. Books are sometimes wrong too. That doesn't invalidate all books. Likewise, the internet can be useful. In both cases, you need to develop some discernment, or ask others to guide you through the vast number of articles, courses, books and the like. You can see product reviews online, but sometimes these are fake! Shock, horror!! If you are considering buying a book, do look at the ACCU book reviews [ACCU]. And note, if you are an ACCU member you can request a copy of a book to review from our list. You get a free book, which might just cover the cost of ACCU membership. What's not to like? Anyway, take a look at our book reviews.

Some rants are purely to let off steam. Have you ever caught yourself in a discussion about tabs versus spaces? Or the one true way to place braces? Or written a language that doesn't use braces, and kinda forces one true way of indenting code? Maybe the latter is less common, but hey. Even Python, which use indentation rather than braces, can still cause 'holy wars'. Have you ever found a Python script with a mix of tabs and spaces? Should you use type hints or embrace duck typing? People are always going to find something to have Opinions (with a capital O), and those opinions are frequently divided. Such disagreements can be a good way to explore different viewpoints and understandings. You should probably be deeply suspicious if everyone agrees. Conflict can enable discussion and discovery. My Dad was a mathematics teacher, and went on to teach trainee teachers. He always said arguments and conflicts facilitate learning. If you can come out with a good example of surprising behaviour, you have a good teaching example. I recall my mind being blown by non-transitive dice when I was very young. My Dad made some: you use three, with six faces, like usual dice, but the numbers differ. They have three pairs of values each:

- Die A has sides 2, 2, 4, 4, 9, 9.
- Die B has sides 1, 1, 6, 6, 8, 8.
- Die C has sides 3, 3, 5, 5, 7, 7.

Each can beat the other two. You could do some sums on the probabilities of each die winning against the others [Wikipedia-3]. Not everything has an ordering. The Pragmatic Programmers are running a series of 'Brain teaser' books [Pragmatic]. I've read the Numerical, C++ and C# books in this series. They are nice and short, and give you plenty to think about, including some very surprising behaviour, which will improve your understanding of the topics covered.

If you find yourself faced with people in conflict and can't manage to bring everyone to agreement, consider Timur Doumler's 'Herding Cats' talk [Doumler25]. He spoke about trying to get people to agree on contracts for C++. This has been a work in progress for a very long time, making the drawn out discussions over concepts look like a walk in the park. Though the focus of the talk is on contracts themselves, Timur's overview says:

> Within the C++ standardisation process, we face significant obstacles: the infamously slow and challenging ISO process, the absence of a holistic strategy, and the lack of a shared understanding of what "safety" even means.

He ends,

> We also explore the human side of standardisation – the art of helping fiercely opinionated engineers find common ground – and what we can learn from it.

Trying to ensure everyone can make their point, but also listen to opposing views and find a way to agree is a challenge. Listen to his talk: he suggests some ways to make progress. C++'s design by committee can make the process slow, but it also means a variety of people can give input. Thank you committee for all the hard work.

So, ignorance can be fixed, but you need to become aware of your lack of knowledge. Perhaps you don't need to know something. I can live with not knowing Rust, at the moment. If the time comes, I might give it a go. There are plenty of resources out there that can help you if you want to learn something new, but be picky. Eyeball the ACCU book reviews, or chat to someone before you invest time on badly written books or training. Another good way to learn is by explaining to someone else. Maybe write an article for us? You will find you need to fill in some details you hadn't thought about, or get feedback nudging you to be more precise. If you do have something to rant about, that could make a good article too. Offer a proof of concept to get it off your chest or show an alternative.

Notice the Dunning-Kruger Wikipedia [Wikpedia-1] article points out towards the end:

> Ignorance is sometimes bliss. In this sense, optimism can lead people to experience their situation more positively, and overconfidence may help them achieve even unrealistic goals

I suspect I would never have embarked on some of my personal projects if I had realized how difficult or time consuming they would be. Ignorance isn't always bad, but learning and thinking is almost always good.

References

[ACCU] ACCU book reviews: https://accu.org/menu-overviews/reviews-overview/

[Buontempo24] Frances Buontempo (2024) *Learn C++ by Example* ISBN 9781633438330, available at https://www.manning.com/books/learn-c-plus-plus-by-example

[Buontempo26] Frances Buontempo *Introducing C++* (due March 2026), see https://www.oreilly.com/library/view/introducing-c/9781098178130/

[Doumler25] Timur Doumler 'Contracts, Safety, and the Art of Cat Herding' given at *C++ on Sea 2025*, available at: https://www.youtube.com/watch?v=gtFFTjQ4eFU

[Nash23] Phil Nash 'Introduction to C++ Coroutines', abstract and relevant links at https://old.cpponsea.uk/2023/sessions/introduction-to-cpp-coroutines.html

[Nash25] Phil Nash (2025) 'C++ Coroutines – Gods from the Machine' presented at ACCU 2025, available at: https://www.youtube.com/watch?v=b6pYieNd_OY

[NIH] Miguel Gallegos, Viviane de Castro Pecanha and Tomás Caycho-Rodríguez, 'Anti-vax: the history of a scientific problem', published in *Journal of Public Health* on 16 April 2022 and available from National Library of Medicine at https://pmc.ncbi.nlm.nih.gov/articles/PMC9383768/

[Pragmatic] 'Brain Teasers' series of books: https://pragprog.com/categories/brain-teasers/

[Rust] Rust standard: `unsafe` keyword: https://doc.rust-lang.org/std/keyword.unsafe.html

[Wikipedia-1] Dunning-Kruger effect: http://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

[Wikipedia-2] Willful ignorance: https://en.wikipedia.org/wiki/Willful_ignorance

[Wikipedia-3] Intransitive dice: https://en.wikipedia.org/wiki/Intransitive_dice

Vibe Coding in Rust

Many people are learning how to herd LLMs right now. Amy Thomason discusses lessons learnt from using GenAI to code in Rust.

I've been writing code since the '70s and things have gone through a number of changes since then, to say the least. From machine code to assembly to early compilers, coding standards and practices up to the present day where a new paradigm is emerging in the form of LLM models focused on generating and analysing code.

All of this has been change, relentless change, and you probably should not be in the software business if you are not quick to adapt to the pace of technology.

As a certified old fart, I was very sceptical of the idea of LLMs generating code and to a degree I remain so. Computational linguistics was very definitely a thing when I started my career, we founded the Manchester Aristotelian Society in the '80s to discuss philosophy with a group of computational linguists who were using Prolog to translate to and from Japanese.

Using Prolog for language was a highly Asimovian aproach to the problem. Like the positronic brain, every detail needed to be worked out by hand, every rule of grammar raked over and converted to statements. The idea of doing this with neural networks would have seemed crazy as the memory and computational requirements would have been impossible with the hardware of the day and the text required to train the networks was not available.

These days we have a huge number of language models to choose from for every situation from text-to-image to biological paper scanning. I'm going to do a little experiment using the default settings of the copilot extension to VS Code to see if we can actually write something that works.

Note that I'm not an expert at this, not doubt there will be others who have done little else for the last few years who can speak more confidently about how to do this. There are also a myriad of YouTube videos to help you on your way.

We're using Rust for this as it is the language that I teach and is increasingly the choice for startups and new projects and even a few legacy ones like Linux. If you are looking to book a training course in Rust for your existing C++ teams then this is a shameless plug.

This article is based on a workshop we held at Oxford ACCU/Rust the first wednesday of July. Come to our sessions if you want to learn more!

Amy Thomason After reverse engineering the Namco console in the '70s, Amy went on to develop typesetting systems and GUI operating systems in the '80s and game engines in C++ in the '90s, switching to developing the PS3 compiler in the 2000s. She's been teaching C++ game engine development at Goldsmiths, moving to Genetics in Oxford where she runs the ACCU meetups today.

Amy has a Rust training business and can help you with your conversion from C++ to Rust and loves complex technical projects like blockchain nodes and procedural geometry for 3D printing – get in touch on LinkedIn.

Step 1, plan what we are going to do.

As with all coding projects, we need to plan what we are doing and decide on the end goal we are going to reach. We are going to have to proceed in steps and use our existing knowledge of programming to guide us on the way.

Our goal is to write a simple chess board visualisation where two players take it in turn to move pieces on the board. For a stretch goal, we could check the moves.

As with all Rust programs we start with `cargo new` to create a hello world program and then edit with VS Code.

```
$ cargo new overload-vibe-coding
$ code overload-vibe-coding
```

This generates our start point:

```
fn main() {
  println!("Hello, world!");
}
```

We have pre-installed the rust-analyser and copilot extensions to VS Code and are using the default language model, GPT 4. You have a choice of several language models, so have a play with them all.

We can now use CTRL-I to open the Copilot prompt.

I asked it:

> > Write a plan for a two player chess program as a module comment

I'm told that creating a plan first is a good first step as it give you a chance to review what you are about to do.

Of course, we could just write

> > Write me a two player chess program

And I did try this, but the results were quite disappointing and no doubt just cribbed from something on the internet.

The plan it generated looks like Listing 1 (next page).

This sounds credible, so let's proceed. We should be cautious of the origins of such text as it may be subject to copyright, however.

Step 2, A chess piece and board square

Let's define an `enum` for a chess square. We will include a blank square so that we don't need to use an `Option` for blank squares.

The query

> > define an enum for chess pieces with colour and a blank square

was disappointing and the answer was quite verbose.

A few more queries produced a very similar result and I had to repeat the query a few times until I had a similar result to what I expected. This enum should occupy two bytes, one for the discriminator and one for the colour. (See Listing 2, next page.)

```
//! # Two-Player Chess Program
//!
//! This module provides the foundation for
implementing a two-player chess game in Rust.
//! The program will include the following key
components:
//!
//! ## Features
//! - **Game Board Representation**: A data
structure to represent an 8x8 chessboard,
including pieces and their positions.
//! - **Piece Movement Rules**: Logic to validate
legal moves for each type of chess piece (pawn,
knight, bishop, rook, queen, king).
//! - **Turn-Based Gameplay**: Alternating turns
between two players, enforcing the rules of
chess.
//! - **Check and Checkmate Detection**:
Mechanisms to detect when a king is in check or
checkmate.
//! - **Stalemate and Draw Conditions**: Logic
to handle stalemates, insufficient material, and
other draw scenarios.
//! - **Move History**: A record of all moves
made during the game for undo functionality or
analysis.
//!
//! ## Plan
//! 1. **Define Data Structures**:
       - Create enums for `PieceType` (pawn,
knight, etc.) and `Color` (white, black).
//!    - Define a `Piece` struct to represent a
chess piece with its type and color.
//!    - Implement a `Board` struct to represent
the chessboard as an 8x8 grid.
//!
//! 2. **Implement Movement Logic**:
//!    - Write functions to validate moves for
each piece type based on chess rules.
//!    - Ensure moves respect the board
boundaries and piece interactions (e.g.,
capturing, blocking).
//!
//! etc.
```

Listing 1

I would imagine that the source weights had a few variants that it was cycling through.

I tried to add an extra element **`Blank`** to the enum, but manually editing worked better. I also corrected the spelling to European English.

Next:

```
// > make a struct GameState with an 8x8 board
// and a player colour.
#[derive(Debug)]
struct GameState {
  board: [[ChessPiece; 8]; 8],
  current_player: Colour,
}
```

From now on I'll show just the final result of each query. I would imagine that the queries would be better with a more experienced 'vibe' coder.

```
// > make an enum ChessPiece
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
enum ChessPiece {
    Pawn(Color),
    Knight(Color),
    Bishop(Color),
    Rook(Color),
    Queen(Color),
    King(Color),
}
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
enum Color {
    White,
    Black,
}
```

Listing 2

```
// > add a constructor to GameState using an 8x8
//   const array
fn new() -> Self {
  use ChessPiece::*; // These were manual edits
                     // to make the code shorter.
  use Colour::*;
  const INITIAL_BOARD: [[ChessPiece; 8]; 8] = [
    [Rook(White), Knight(White), Bishop(White),
      Queen(White), King(White), Bishop(White),
      Knight(White), Rook(White)],
    [Pawn(White); 8],
    [Blank; 8],
    [Blank; 8],
    [Blank; 8],
    [Blank; 8],
    [Pawn(Black); 8],
    [Rook(Black), Knight(Black), Bishop(Black),
      Queen(Black), King(Black), Bishop(Black),
      Knight(Black), Rook(Black)],
  ];
  GameState {
    board: INITIAL_BOARD,
    current_player: Colour::White,
  }
}
```

Listing 3

Step 3, The game state

The local **`use`** items here is not in common usage in Rust, but I find it reduces the source code a great deal so I edited the result. Very few Rust examples use local **`use`** to reduce code bloat like this. (See Listing 3.)

A general observation is that I'm being steered in the direction of the original coder of this problem and making the code *yours* would be more challenging for a less experienced coder.

But on the whole, it has made something that seems to work with very few edits.

```
// > implement a method to convert Square to a
// row and column
impl Square {
  fn to_row_col(self) -> (usize, usize) {
    let index = self as usize;
    let row = index / 8;
    let col = index % 8;
    (row, col)
  }
}
```

And now let's show the board and next player (see Listing 4).

```
// > implement Display for GameState with labeled
// rows and columns
impl fmt::Display for GameState {
  fn fmt(&self, f: &mut fmt::Formatter<'_>)
        -> fmt::Result {
    writeln!(f, "  A B C D E F G H")?;
    for (i, row) in self.board.iter().enumerate()
{
      write!(f, "{} ", 8 - i)?; // Row labels
                                // (8 to 1)
      for piece in row.iter() {
        let symbol = match piece {
          ChessPiece::Pawn(Colour::White) => "♙",
          ChessPiece::Knight(Colour::White)
             => "♘",
          // ...
          ChessPiece::King(Colour::Black) => "♚",
          ChessPiece::Blank => ".",
        };
        write!(f, "{} ", symbol)?;
      }
      writeln!(f)?;
    }
    writeln!(f, "  A B C D E F G H")?;
    writeln!(f, "Current Player: {:?}",
      self.current_player)
  }
}
```

Listing 4

```
// > implement FromStr for square
impl FromStr for Square {
  type Err = String;
  fn from_str(s: &str) -> Result<Self, Self::Err>
  {
    match s.to_uppercase().as_str() {
      "A1" => Ok(Square::A1),
      "A2" => Ok(Square::A2),
      "A3" => Ok(Square::A3),
      "A4" => Ok(Square::A4),
      // ...
      "H5" => Ok(Square::H5),
      "H6" => Ok(Square::H6),
      "H7" => Ok(Square::H7),
      "H8" => Ok(Square::H8),
      _ => Err(format!("Invalid square: {}", s)),
    }
  }
}
```

Listing 5

Let's convert strings into squares. This is a bit of a hacky solution as we could have parsed the chars and `to_uppercase()` allocates memory which is a no-no in system languages. (See Listing 5.)

Test generation did a lot more work than I asked and made a number of tests for `Square`. (See Listing 6.)

Finally, we add a game loop (Listing 7), and we're now ready to run a system test (see Figure 1).

Clearly something is wrong.

- What seem to be white pieces are actually black.
- The board is drawn upside down.
- `from.to_row_col` swaps rows for columns.

Interestingly, these are all mistakes that humans would make. The code *looked* good but was rotten at its core.

Fixing these problems (manually, because for me it is easier) gives a better result where the moves actually work correctly. (See Figure 2.)

Clearly we can continue to develop the plan, but we will run out of paper.

Conclusions

Yes. It is possible to write code with LLMs but it does need some looking after. It will try to push you towards the program it was trained on. No doubt this will improve with more training as more examples become available.

The quality of code generated by LLMs is highly dependent on the quality of the code of the training set. For example, the use of `to_uppercase()` shown above isn't ideal. A junior coder would not know why this is a bad idea having not spent hours on Godbolt checking the codegen.

The LLM is very useful for looking things up, like ANSI escape codes or unicode characters. It is also useful for completion, although rust-analyser's completion is very often more targeted.

LLMs are very useful for teaching you a new programming language. For learners of Rust, they must be a great help. I've learned most of my programming languages from reference manuals, and I would not wish that on a new generation, especially one with hundreds of programming languages to learn.

```
// > test that to_row_col returns (0,0) for A1
#[cfg(test)]
mod tests {
  use super::*;

  #[test]
  fn test_to_row_col_a1() {
    let square = Square::A1;
    let (row, col) = square.to_row_col();
    assert_eq!((row, col), (0, 0));
  }
  // ...
}
```

Listing 6

```
// > add a loop to display the board and accept
// moves
fn main() {
  let mut game_state = GameState::new();
  loop {
    println!("{}", game_state);
    println!("Enter your move (e.g., 'e2 e4') or"
      " 'quit' to exit:");
    let mut input = String::new();
    std::io::stdin().read_line(&mut input)
      .expect("Failed to read input");
    let input = input.trim();
    // ...
    if let Err(err) = game_state.make_move(
        from_square, to_square) {
      println!("Invalid move: {}", err);
      continue;
    }
  }
}
// > implement make_move
fn make_move(&mut self, from: Square, to: Square)
-> Result<(), String> {
  let (from_row, from_col) = from.to_row_col();
  let (to_row, to_col) = to.to_row_col();
  // ...
}
```

Listing 7

For tests, LLMs take a lot of the pain out of the process and seem to be able to get good coverage. You should review the results, however.

On the whole, I think that LLMs are here to stay. For CRUD farming and front-end development it will become a necessary tool and even more eclectic things such as compiler development can benefit from the completion and lookup LLMs offer. We do need to take care and review the generated code, however, as what looks convincing may not actually work. ■

```
$ cargo run
    Finished 'dev' profile [unoptimized +
debuginfo] target(s) in 0.01s
     Running 'target/debug/overload-vibe-coding'
  A B C D E F G H
8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
7 ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
6 . . . . . . . .
5 . . . . . . . .
4 . . . . . . . .
3 . . . . . . . .
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
1 ♜ ♞ ♝ ♕ ♔ ♝ ♞ ♜
  A B C D E F G H
Current Player: White

Enter your move (e.g., 'e2 e4') or 'quit' to
exit:
E2 E4
Invalid move: No piece at the source square.
```

Figure 1

```
Enter your move (e.g., 'e2 e4') or 'quit' to
exit:
e2 e4
  A B C D E F G H
8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
7 ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
6 . . . . . . . .
5 . . . . . . . .
4 . . . . ♟ . . .
3 . . . . . . . .
2 ♟ ♟ ♟ ♟ . ♟ ♟ ♟
1 ♜ ♞ ♝ ♕ ♔ ♝ ♞ ♜
  A B C D E F G H
Current Player: Black
```

Figure 2

UDB: The undo.io Time Travel Debugger

Finding problems in code can be difficult and time consuming. Paul Floyd explains how to use UndoDB to debug code.

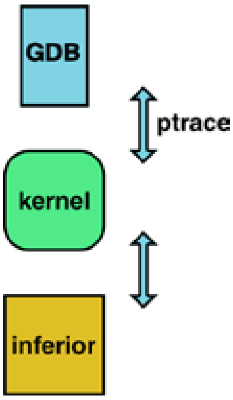
Introduction

UDB is a Linux interactive debugger sold by undo [undo-1]. It uses the same command interface as GDB. Like GDB you can use it either in a text terminal or integrated with an IDE such as VS Code or CLion. Integration means that you will get extra buttons/keyboard shortcuts to access some of the additional UDB features. I'm not going to describe using UDB with a kind of GUI. There is plenty of value in using GUIs (in particular, having automatically refreshed views of local variables and the call stack). That said, if you know the UDB command line then using a GUI will be easy. Personally, I use UDB and GDB mainly in TUI mode[1] (text user interface) which presents a split text screen with panes for source code, assembler and debugger commands.

I'll start with a quick overview of how the GDB command line works and then I'll go on to describe some of the things that you can go with UDB.

GDB basics

GDB is the GNU debugger [GDB-1]. With it you can debug executables (the 'inferior' in GDB lingo). GDB controls the inferior using the `ptrace` system call [ptrace]. Effectively this means that GDB is going via the kernel to start/stop/step/read registers and memory in the inferior process.



Normally you will have prepared your executable by building it with DWARF debug information [DWARF] (adding the `-g` or `-g3` flag to GCC or LLVM compiler). You can still debug without debug information, but you won't have the source code so you will have to work at the assembly code level.

Once your executable is prepared you can debug it

```
gdb --args your_executable
 [options for your_executable]
```

This will give you the gdb prompt

```
(gdb)
```

The inferior isn't running at this point. Unless you need to debug the startup code that runs before `main()`, you can use the `start` command which does the equivalent of setting a temporary breakpoint on `main()`, running and then stopping at the start of `main()`. If you are using a modern Linux and GDB, it will then ask you if you want to download any missing debuginfo for libraries like libc and libstdc++. This is a great feature, saving you from having to install endless debuginfo packages.

GDB then prints the first line of source and the prompt. This isn't a tutorial on GDB, so I'll just add a table of frequently used commands related to executing code. Commands can be abbreviated as long as the abbreviation is not ambiguous.

Command	Action	Comments
help [command]	Show help for the given command.	'help' on its own will print a list of the main subjects.
<enter>	Repeat previous command.	Just the <enter> key.
r[un]	Run the executable from the start.	You can interrupt with ctrl-c.
b[reak] [location]	Set a breakpoint.	GDB will run until the location is reached.
wa[tch]	Set a watchpoint.	GDB will run until the expression in the watchpoint changes.
c[ontinue]	Continue running.	
n[ext]	Execute the next C++ statement.	Passes through functions. Sometimes called 'step over'. Also ni for next assembler instruction.
s[tep]	Execute the next C++ statement.	Steps into functions. Sometimes called 'step in'. Also si for step assembler instruction.
fin[ish]	Execute until the current function ends.	Sometimes called 'step out'.
bt or backtrace or where.	Prints the current callstack.	
info	Many suboptions, gets info on some GDB state.	Examples: b[reakpoints] to list active breakpoints, frame to see the current active call frame

There is one thing that is missing. Those commands are c[ontinue] *forwards*, n[ext] statement *forwards*, s[tep] statement *forwards* and fin[ish] function *forwards*. What if you could do all those *backwards* as well? That is where UDB comes in.

Before I launch into some details, why would you want to do that? Quite often, when you are debugging, the problem is something 'nice' (from a debugging perspective) like a segmentation fault. You run your executable, it crashes and the debugger stops. You look at the line of code where it crashed, check the variables and see that something is `NULL` and you can see the problem. Life isn't always so easy.

Paul Floyd has been writing software, mostly in C++ and C, for about 35 years. He lives near Grenoble, on the edge of the French Alps, and works for Siemens EDA developing tools for analogue electronic circuit simulation. In his spare time, he maintains Valgrind. He can be contacted at pjfloyd@wanadoo.fr

---

[1] In GDB, there are often 3 or 4 ways of doing the same thing (command line options, keyboard shortcuts, GDB commands and python code). Personally, I use keyboard shortcuts like CTRL+X then A. There is a 15-minute video introduction to TUI [Law15].

> **We've all had that sinking feeling when you have just spent all afternoon debugging to reach an error and you accidentally hit 'next' one time too many.**

The kind of problem for which UDB excels is when you have a problem where:

1. It is not obvious where things are going wrong (the `NULL` pointer comes from far, far away, in extreme cases via JITed code that your IDE can't follow).
2. It takes a long time – maybe hours – to reach the code and condition where the problem is.
3. You don't know enough about the problem to set a breakpoint just before it happens. An example of that could be when you know which function to debug but the function gets called thousands of times and it's only when the arguments and class state have a particular combination of values that the error gets triggered. Even worse, that state condition could be non-deterministic so you can't even methodically note the steps to reach the error and redo them every time you need to restart a debug session.

We've all had that sinking feeling when you have just spent all afternoon debugging to reach an error and you accidentally hit 'next' one time too many. That is very easy to do when you have been stepping for a long time and stepping has become an automatic reflex. If you are using UDB, that is no longer a problem.

UDB prompt and recording

The first thing that you need to know about UDB is that it needs to record events in order to be able to replay them. Recording is turned on by default. When you start the debugging session the prompt will be `not running>`. Then, when you run and stop somehow, it will show the recording status. With recording on that might be something like `recording 128,774>`. The number after `recording` is the number of events recorded. The events correspond to basic blocks (blocks of linear machine code ending with a branch statement). If you time travel backwards, the prompt will show you a percentage of the record buffer and the event number, such as `99% 128,773>` That can be useful when you want to pinpoint exactly where a change occurs. Just keep a note of the event count before and after and then you can just keep going backwards and forwards until you find the change. In addition to the event count, UDB has commands that let you set a bookmark in the event history that you can return to later. UDB can also go back to a given event or wallclock time.

Since recording does have an overhead, you can turn it off with the `–defer-recording` command line option. You could then run to some breakpoint and avoid needlessly recording events that aren't of interest. When you know that you are getting close to the code that you want to analyse you can turn recording on with `urecord`.

Additional UDB commands

UDB adds a set of *reverse* and *undo* commands to GDB. I'll get to the *undo* commands in a moment. The *reverse* commands just do what the forward commands do but in the opposite direction.

Command	Action
rs, reverse-step	Step backwards, going into function calls.
rn, reverse-next	Next backwards, going back to function call points.
rf, rfin, reverse-finish	Go backwards to calling function.
rc, reverse-continue	Execute continuously backwards.

Let's take a small example:

```
 1  int a, b, c, d;
 2
 3  void g()
 4  {
 5     c = 3;
 6  }
 7
 8  void f(void)
 9  {
10      b = 2;
11      c = 3;
12      g();
13  }
14
15  int main()
16  {
17      a = 1;
18      f();
19      d = 4;
20  }
```

If you debug this with UDB and do the following

- `start` – advances to line 17
- `n` (for next) – advances to line 18
- `<enter>` (repeat next) – steps over `f()` and advances to line 19

At this point if you enter `rn` (reverse-next) it will take you back to line 18 before `f()` is called. At the same point, if you enter `rs` (reverse-step), it will take you back into function `f()` to line 12.

Now if you do the following after starting UDB:

- `b 6` – set a breakpoint on line 12
- `r` – runs to the breakpoint at line 12

At this point the debugger is stopped near the end of function `f()`. This time, if you enter `rfin` (reverse-finish) then it will take you back to line 18 where `f()` is called.

In addition to the *reverse* commands, there are also more generic *undo* commands. The main two commands are `uu` (`ugo undo`), which is a generic 'undo last command that changed your debug time'. That has a mirror command `ur` (`ugo redo`) which redoes the last command that changed the debug time. An example of when you might want to use `uu` is when you have just just done a `continue` to a breakpoint and you then realise that the breakpoint is too late. You you just use `uu` to go back to the point where you did the `continue`.

I have only covered the basics of using UDB. A couple of commands that I've never used yet are `usave` and `uload`, which allow you to save and reload the execution history. I imagine that can be a a great time saver when you are debugging large executables that take a long time to reach the problem code. There are numerous commands for getting information about the recorded history. I'm a bit fascinated by the `ublame` command, for debugging POSIX shared memory. That seems to me like a very niche thing to debug. There must have been a quite specific user request to develop such a feature.

How does it all work? UDB is a commercial product so the internals aren't documented. As I understand it, the key things are to only record the strict minimum of things to keep the size of the recording history down. When going backwards and forwards PTRACE syscalls can be used to restore the registers and memory. In some cases, where UDB needs to make something that is non-deterministic repeatable, it has to fake the non-deterministic call. For instance, if you call `clock_gettime` then reverse and redo the same call to `clock_gettime` then the second call will get the same time value even though the physical clock will have changed. UDB does some magic in the background.

UDB in practice

A few months ago I was working on a problem in a large executable which was giving different behaviour when running in two different modes. The expectation was that they should both be very similar. In the second mode, there was an error which was occurring a long time (in terms of statements) after the real cause of the problem. The code involved was developed by several teams and I only had moderate familiarity with some parts. I did try '`printf` debugging' using our log files. That was intractable as the log files were hundreds of megabytes in size and any kind of diff tool either took forever to work out the diffs or got completely lost synchronising the two logs. It took me several hours of parallel debugging to narrow down the place in the code where the two exes were diverging in behaviour. I could have done that with just GDB, setting breakpoints that slowly advance through the execution, restarting every time that I overshot the problem zone. With each restart of the debug session taking a few minutes that would have added a few more hours. With UDB I would just go backwards whenever I overshot. The final step in that debugging marathon did require extra work to compare some large structs containing many option flags. The source of the problem seemed obvious when I had isolated the correct flag.

More information

Undo.io has a YouTube channel [undo-2] I've watched a good few of the videos and they are of a good level technical quality. Greg Law, a co-founder of undo.io, is active on the conference circuit (*Meeting C++*, *CppCon*, *ACCU*, *C++ on Sea*). Mostly these talks are about GDB and debugging in general . There is one particular ACCU conference where he presented UDB that I recommend that you watch if you'd like to find out a bit more of what UDB does and how it works [Law24]. I did enter the raffle and got a 1 year licence which is part of the reason that I'm writing this (we also use it at work). undo.io is the main contributor to a Reddit community, r/cpp_debugging [Reddit], which is mainly links to the YouTube videos.

Last but not least, there is the undo.io documentation [undo-3]. I found the Quick Reference [undo-4] the best way to get familiar with the commands that UDB adds to the GDB interface.

UDB isn't the only debugger with time travel ability. Wikipedia [Wikipedia-1] lists several. If you don't have access to a licence for UDB then you could look at rr [Wikipedia-2] (developed by Mozilla) or the reverse debugging capabilities of GDB itself [GDB-2] (also a tutorial [Redhat]). I've never used these myself but I have heard that UDB has better performance and is more reliable.

Conclusion

UDB is great. I've had a few niggles with it like reverse debugging multi-threaded applications (this has been improved according to the release notes for more recent versions). I sometimes had issues with our licence server – it didn't seem to like UDB going back in time. Otherwise, it just works. I do have a vague idea of the gymnastics that it must be doing under the hood like JITting little bits of code so that it can patch up what you see executing with the recorded history. Getting all that to work is an amazing technical achievement. And to use it, most of the time all that is needed is type an extra 'r'. ■

References

[DWARF] https://dwarfstd.org/

[GDB-1] https://sourceware.org/gdb/

[GDB-2] Reverse debugging capabilities: https://sourceware.org/gdb/wiki/ReverseDebug

[Law] Greg Law, YouTube channel: https://www.youtube.com/results?search_query=%22Greg+Law%22

[Law15] Greg Law 'Give me 15 minutes and I'll change your view of GDB', presented at CppCon 2025 and available at: https://www.youtube.com/watch?v=PorfLSr3DDI

[Law24] Greg Law, 'Time Travel Debugging', presented at *ACCU 2024* and available at https://www.youtube.com/watch?v=n3OCQ35Xhco

[ptrace] https://www.man7.org/linux/man-pages/man2/ptrace.2.html

[Reddit] Debugging community: https://www.reddit.com/r/cpp_debugging/

[Redhat] 'Using GDB to time travel', published 8 August 2024 at https://developers.redhat.com/articles/2024/08/08/using-gdb-time-travel#

[undo-1] https://undo.io/

[undo-2] YouTube channel: https://www.youtube.com/@Undo-io

[undo-3] Documentation: https://docs.undo.io/

[undo-4] Quick reference: https://docs.undo.io/UDB-quickref.pdf

[Wikipedia-1] Time travel debugging: https://en.wikipedia.org/wiki/Time_travel_debugging

[Wikipedia-2] rr (debugging): https://en.wikipedia.org/wiki/Rr_(debugging)

Advertise with ACCU

Many of our readers make purchasing decisions, or recommend products for their organisations.

Reasonable rates. Flexible options. Member discounts available.

Contact ads@accu.org for details.

accu

Filters and Wildcards (Shell Globbing)

If you have documentation in html but want to provide markdown too, what can you do? Ian Bruntlett describes how he used a shell script to automate the translation and what he learnt.

A filter is a programme that reads its input stream (file descriptor 0 aka **`stdin`** aka **`cin`**), modifies it, then writes the results to its output stream (file descriptor 1 aka **`stdout`** aka **`cout`**). Errors get written to the error stream (file descriptor 2 aka **`stderr`** aka **`cerr`**).

I have been maintaining some HTML pages on I.T. related sources of information, ‘TECH-Manuals’, for personal use for quite some time. I wanted to put one of those pages online. Github seemed like a good idea and I uploaded it to GitHub [Bruntlett-1]. Unfortunately when you look at HTML pages on GitHub, you see the raw HTML. Apparently you have to upload MarkDown (.md) files instead. After searching the Ubuntu package repositories with Synaptic Package Manager, I discovered that the **`html2markdown`** command would do what I wanted. With some caution I performed this command to create `tm-free-software.md` ready to upload to GitHub [Bruntlett-2]:

```
$ html2markdown < tm-free-software.html \
> tm-free-software.md
```

That is all well and good but it relies on me not getting the input and output filenames wrong. As a filter, it works on one file only. That is OK but I also wanted the ability to use wildcards (shell globbing) to save typing and to make it easier to use with the find command.

The resulting shell script is modular – and, as a bonus, if you have another filter that you want to work on multiple files, all you have to do is rename the shell script, edit and rename function **`perform_html2markdown`** and edit **`strip_extension`**’s code to accept the new input filename extension(s).

The first bit of executable code checks that at least one parameter has been passed:

```
if [ $# -lt 1 ]; then
  Usage >&2
  exit 11
fi
```

As I thought was conventional, the error message got sent to **`stderr`** (it turns out I was wrong) and then the script exits with a status code of 11 so as not to collide with return values of **`html2markdown`** itself. See [Cooper14] for more information.

The next bit of script loops over the shell script’s parameters. As long as the script *has* parameters, it invokes my function **`perform_html2markdown`** with a single input filename. I had to name it something different to **`html2markdown`** so, in an obscure reference to COBOL’s PERFORM statement, I named the function **`perform_html2markdown`**.

The **`for`** loop is used to execute **`perform_html2markdown`** until it fails or all input files have been processed.

So, how does **`perform_html2markdown`** work? (Listing 1.)

Ian Bruntlett is a keen reader of software development books. He has promised himself a long stint at dealing with C++, once he has got to grips with Git.

```
function perform_html2markdown()
{
  local input_html_file output_markdown_file
  input_html_file="$1"

  if ! output_markdown_file=$(strip_extension \
      "$input_html_file").md ; then
    echo "Bad HTML filename: $input_html_file";
    return 10; # unsupported or bad filename
                # extension
  fi

  echo Translating  "$input_html_file" to \
    "$output_markdown_file"
  html2markdown --no-skip-internal-links < \
    "$input_html_file" > "$output_markdown_file"
}
```

Listing 1

In the interests of modularity and ease of development and maintenance, I use functions in bash, making sure that I declare working variables as local. Unlike in C++, variables used in a function are global. That has caused me problems in the past.

The function **`perform_html2markdown`** knows it has been passed a parameter and declares two local variables – **`input_html_file`**, and **`output_markdown_file`**. The **`input_html_file`** variable is, as to be expected, the name of the input file. I *could* have used **`$1`** instead but I decided to name it to make future maintenance work that little bit nicer. The **`output_markdown_file`** variable invokes another function:

```
function strip_extension
{
  local destination_file
  # cater for .html source
  destination_file=$(basename "$1" .html)
  # cater for HTML source
  destination_file=
    $(basename "$destination_file" .HTML)
  echo "$destination_file"
}
```

This version contained a bug – this **`strip_extension`** *also* strips preceding leading directory components! So, I paused writing this article to learn more, referring to [Newham05] and [GNU].

Aided by the power of functions and help from accu-general [ACCU] I came up with Listing 2 (next page).

Confident that I had fixed the problem, I asked on accu-general for comments. I received some very interesting implementations of the function **`strip_extension`**.

Sven opted for two approaches – one using basename (Listing 3, next page). This works… after a fashion. If the user had provided a directory name, that directory name would be lost. So basename `dir-of-html/blank.html` became `blank.html`. The script was in use by me for quite some time before I discovered the bug and consequently stopped writing this article.

```
# remove .HTML or .html from parameter 1 and then
# output/return that result.
function strip_extension
{
  declare -i length
  local length filename \
    filename_stub dot_extension
  filename=$1
  length=${#filename}

  if [ $length -lt 6 ]; then
    return 2 # given filename too short
  fi
  dot_extension=${filename:$length-5:5}
  if [ "$dot_extension" == ".html" ] ||
     [ "$dot_extension" == ".HTML" ]; then
    filename_stub=${filename:0:-5}
    echo "$filename_stub"
  else
    echo
    return 1; # unsupported or bad filename
              # extension
  fi
}
```

Listing 2

Sven came up with an approach that uses sed – a stream editing tool that I have a little bit of experience with. See Listing 4. This works. By running sed, an external command, it is slower but it has the benefit of being correct!

Hans Vredeveld came up with another solution (see Listing 5). This uses bash's pattern-matching operators and is minimally documented by [Newham05] and [GNU]. To quote the former:

> ${variable%pattern} If the pattern matches the end of the variable's value, delete the shortest part that matches and return the rest.

As with other aspects of shell usage, I had to experiment with this to get a better idea of it. The pattern looks at the environment variable specified (note the absence of a preceding `$`), separated by a `%` sign to tell bash what to do and pattern is the thing to delete. In this case, `strip_extension` is using a pattern of `.*` which means match a dot followed by any number of following characters. So, not only does it work for .html or .HTML, it works for .odt etc. The filename does not have to exist, bash is working with characters. Here are some examples:

```
$ filename=dir.html/blank.html
$ echo ${filename%.*}
dir.html/blank
$ filename=dir.html/blank2.html
$ echo ${filename%tml}
dir.html/blank2.h
```

This was then further refined by Sven to avoid using echo:

```
>    stub=$(echo "${filename%.*")
```

You don't need command substitution here: the following should be enough:

```
>    stub="${filename%.*}"
```

```
function strip_extension
{
  filename=$1
  case "$filename" in
    *.html)
        stub=$(basename "$filename" .html)
        ;;
    *.HTML)
        stub=$(basename "$filename" .HTML)
        ;;
    *)
      echo "Not an HTML file name."
      return 1
      ;;
  esac
  echo "$stub"
}
```

Listing 3

```
function strip_extension
{
  filename=$1
  case "$filename" in
    *.html | *.HTML)
      stub=$(echo "$filename" \
        | sed -e 's/\.[^.]*$//')
      ;;
    *)
      echo "Not an HTML file name."
      return 1
      ;;
  esac
  echo "$stub"
}
```

Listing 4

Also discussed by Sven, was my use of a `while` loop to iterate over the command line parameters:

```
while [ "$1" != "" ]; do
  if ! perform_html2markdown "$1" ; then
    exit $?
  fi
  shift
done
```

This is a hang-over from my MS-DOS days, where to access more than 9 (I think) parameters, you had to use the `shift` command. Another quirk is that `exit $?` can be replaced with `exit`.

He proposed:

```
for f in "$@"; do
  if ! perform_html2markdown "$f" ; then
    exit $?
  fi
done
```

Which does the same job without making processed parameters unavailable. The use of `$@` and `*` are… subtle. To quote Cameron Newham and Bill Rosenblatt [Newham05]:

> “$*” is a single string that consists of all of the positional parameters, separated by the first character in the value of the environment variable IFS (internal field separator), which is a space, TAB, and NEWLINE by default. On the other hand, “$@” is equal to “$1” “$2”.. “$N”, where N is the number of positional parameters. That is, it's equal to N separate double-quoted strings, which are separated by spaces. If there are no positional parameters, “$@” expands to nothing. We'll explore the ramifications of this difference in a little while.

Dabbling further, I used Newham and Rosenblatt's [Newham05] function `countargs` (see Listing 6, next page).

When run, this illustrates the difference between `$*` and `@`.

```
$ ./countargs Hello World
$* : 1 args.
Hello,World
$@ : 2 args.
Hello World
```

```
function strip_extension
{
  filename=$1
  case "$filename" in
    *.html | *.HTML)
        stub=$(echo "${filename%.*")
        ;;
    *)
        echo "Not an HTML file name."
        return 1
        ;;
  esac
  echo "$stub"
}
```

Listing 5

```
#!/usr/bin/env bash
# experimenting from Learning the bash shell,
# chapter 4, page 90

function countargs
{
  echo "$# args."
}
IFS=,
echo -n '$* : '
countargs "$*"
echo "$*"

echo -n '$@ : '
countargs "$@"
echo "$@"
```

Listing 6

Note I set the **`IFS`** global variable to a single comma. This is to illustrate the usefulness of **`$*`** using the first character of **`IFS`** (which defaults to a space) when outputting parameters.

The main loop which iterated over the HTML filenames looked like this:

```
while [ "$1" != "" ]; do
  if ! perform_html2markdown "$1" ; then
    exit $?
  fi
  shift
done
```

which did not preserve the return code of (aka **`$?`**) **`perform_html2markdown`**. It was possible to immediately copy the value of **`$?`** into a variable and return that – I felt that was clunky. Taking into account Sven's recommendation to use **`for`** rather than **`while`** and fixing the bug, I came up with this:

```
for f in "$@"; do
  perform_html2markdown "$f" || exit
done
```

The current version of the code looks like Listing 7. ■

References

[ACCU] https://accu.org/faq/mailing-lists-faq/accu-general – in particular, Mathias, Hans, and Sven.

[Bruntlett-1] Github repository: https://github.com/ian-bruntlett/studies/tree/main/bash

[Bruntlett-2] TechManuals: https://github.com/ian-bruntlett/TECH-Manuals

[Cooper14] Mendel Cooper (2014) *Advanced Bash Scripting Guide*: Appenidx E, 'Exit codes with special meanings', available at: https://tldp.org/LDP/abs/html/exitcodes.html

[GNU] *The GNU Bash Reference Manual*, for Bash, Version 5.3, last updated 18 May 2025, available at: https://www.gnu.org/software/bash/manual/bash.html

[Newham05] Cameron Newham and Bill Rosenblatt (2005) *Learning the bash Shell,* published by O'Reilly.

```
#!/usr/bin/env bash
# Name    : irb-html2markdown
# Purpose : To run the html2markdown command with
# less chance of making a silly typing mistake
# (c) Ian Bruntlett
#
# Changelog removed for brevity. See
# [Bruntlett-1] for complete version...
function Usage
{
cat <<END-OF-USAGE-MESSAGE
Usage: $0 name-of-html-file-1 name-of-html-file-2
name-of-html-file-etc

For each given html filename, convert the HTML
file into markdown, writing the results to a file
with a similar name of the origin - the only
```

Listing 7

```
change being the results filename has .md at the
end and not .html
Note:
Because this utility removes the HTML suffix from
filenames, you can use globbing to specify input
files.

Return codes:
0      Success
1      Input file does not exist
10     Input filename not a .html or .HTML file
11     No parameters passed on command line
END-OF-USAGE-MESSAGE
}

# remove .HTML or .html from parameter 1 and then
# output/return that result.
# With thanks to accu-general posters: Mathias
# Gaunard and Hans Vredeveld and Sven
# See https://www.gnu.org/software/bash/manual/
# bash.html#Shell-Parameter-Expansion
# 10 - Input filename not an html or HTML file
# name
function strip_extension
{
  local filename stub
  filename=$1
  case "$filename" in
    *.html | *.HTML)
      stub="${filename%.*}"
   ;;
   *)
    echo "Not an HTML file name."
    return 10
   ;;
  esac
  echo "$stub"
}

function perform_html2markdown()
{
  local input_html_file output_markdown_file

  input_html_file="$1"

  if ! output_markdown_file=$(strip_extension \
    "$input_html_file").md ; then
  echo "Bad HTML filename: $input_html_file";
  return 10; # unsupported or bad filename
             # extension
  fi

  echo Translating  "$input_html_file" to \
    "$output_markdown_file"

  html2markdown --no-skip-internal-links < \
    "$input_html_file" > "$output_markdown_file"
}

# main code here

if [ $# -lt 1 ]; then
  Usage >&2
  exit 11 # error. need at least 1 parameter
fi

for f in "$@"; do
  perform_html2markdown "$f" || exit
done
```

Listing 7 (cont'd)

User-Defined Formatting in std::format

std::format allows us to format values quickly and safely. Spencer Collyer demonstrates how to provide formatting for a simple user-defined class.

In a previous article [Collyer21], [I gave an introduction to the **`std::format`** library, which brings modern text formatting capabilities to C++.

That article concentrated on the output functions in the library and how they could be used to write the fundamental types and the various string types that the standard provides.

Being a modern C++ library, **`std::format`** also makes it relatively easy to output user-defined types, and this series of articles will show you how to write the code that does this.

There are three articles in this series. This article describes the basics of setting up the formatting for a simple user-defined class. The second article will describe how this can be extended to classes that hold objects whose type is specified by the user of your class, such as containers. The third article will show you how to create format wrappers, special purpose classes that allow you to apply specific formatting to objects of existing classes.

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

Interface changes

Since my previous article was first published, based on the draft C++20 standard, the paper [P2216] was published which changes the interface of the **`format`**, **`format_to`**, **`format_to_n`**, and **`formatted_size`** functions. They no longer take a **`std::string_view`** as the format string, but instead a **`std::format_string`** (or, for the wide-character overloads **`std::wformat_string`**). This forces the format string to be a constant at compile time. This has the major advantage that compile time checks can be carried out to ensure it is valid.

The interfaces of the equivalent functions prefixed with **`v`** (e.g. **`vformat`**) has not changed and they can still take runtime-defined format specs.

One effect of this is that if you need to determine the format spec at runtime then you have to use the **`v`**-prefixed functions and pass the arguments as an argument pack created with **`make_format_args`** or **`make_wformat_args`**. This will impact you if, for instance, you want to make your program available in multiple languages, where you would read the format spec from some kind of localization database.

Another effect is on error reporting in the functions that parse the format spec. We will deal with this when describing the **`parse`** function of the **`formatter`** classes described in this article.

C++26 and runtime_format

Forcing the use of the **`v`**-prefixed functions for non-constant format specs is not ideal, and can introduce some problems. The original P2216 paper mentioned possible use of a **`runtime_format`** to allow non-constant format specs but did not add any changes to enable that. A new proposal [P2918] does add such a function, and once again allows non-constant format specs in the various **`format`** functions. This paper has been accepted into C++26, and the **`libstdc++`** library that comes with GCC should have it implemented by the time you read this article, if you want to try it out.

Creating a formatter for a user-defined type

To enable formatting for a user-defined type, you need to create a specialization of the struct template **`formatter`**. The standard defines this as:

```
template<class T, class charT = char>
  struct formatter;
```

where **`T`** is the type you are defining formatting for, and **`charT`** is the character type your formatter will be writing.

Each **`formatter`** needs to declare two functions, **`parse`** and **`format`**, that are called by the formatting functions in **`std::format`**. The purpose and design of each function is described briefly in the following sections.

Inheriting existing behaviour

Before we dive into the details of the **`parse`** and **`format`** functions, it is worth noting that in many cases you can get away with re-using existing formatters by inheriting from them. Normally, you would do this if the standard format spec does everything you want, so you can just use the inherited **`parse`** function and write your own **`format`** function that ultimately calls the one on the parent class to do the actual formatting.

For instance, you may have a class that wraps an **`int`** to provide some special facilities, like clamping the value to be between min and max values, but when outputting the value you are happy to have the standard formatting for **`int`**. In this case you can just inherit from **`std::formatter<int>`** and simply override the **`format`** function to call the one on that formatter, passing the appropriate values to it. An example of doing this is given in Listing 1 on the next page.

Or you may be happy for your formatter to produce a string representation of your class and use the standard string formatting to output that string. You would inherit from **`std::formatter<std::string>`** and just override the **`format`** function to generate your string representation and then call the parent **`format`** function to actually output the value.

The parse function

The **`parse`** function does the work of reading the format specification (*format-spec*) for the type.

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

> The format-spec for your type is written in a mini-language which you design …there are no rules for the mini-language, as long as you can write a parse function that will process it

```
#include <format>
#include <iostream>
#include <type_traits>

class MyInt
{
public:
  MyInt(int i) : m_i(i) {};
  int value() const { return m_i; };
private:
  int m_i;
};
template<>
struct std::formatter<MyInt>
  : public std::formatter<int>
{
  using Parent = std::formatter<int>;
  auto format(const MyInt& mi,
    std::format_context& format_ctx) const
  {
    return Parent::format(mi.value(),
      format_ctx);
  }
};
int main()
{
  MyInt mi{1};
  std::cout << std::format(“{0} [{0}]\n”, mi);
}
```

Listing 1

It should store any formatting information from the *format-spec* in the `formatter` object itself[1].

As a reminder of what is actually being parsed, my previous article had the following for the general format of a replacement field:

‘`{`’ *[arg-id] [*‘`:`’ *format-spec]* ‘`}`’

so the *format-spec* is everything after the `:` character, up to but not including the terminating `}`.

Assume we have a typedef `PC` defined as follows:

```
using PC = basic_format_parse_context<charT>;
```

where `charT` is the template argument passed to the `formatter` template. Then the `parse` function prototype looks like the following:

```
constexpr PC::iterator parse(PC& pc);
```

The function is declared `constexpr` so it can be called at compile time.

The standard defines specialisations of the `basic_format_parse_context` template called `format_parse_context` and `wformat_parse_context`, with `charT` being `char` and `wchar_t` respectively.

On entry to the function, `pc.begin()` points to the start of the *format-spec* for the replacement field being formatted. The value of `pc.end()` is such as to allow the `parse` function to read the entire *format-spec*. Note that the standard specifies that an empty *format-spec* can be indicated by *either* `pc.begin() == pc.end()` *or* `*pc_begin() == '}'`, so your code needs to check for both conditions.

The `parse` function should process the whole *format-spec*. If it encounters a character it doesn't understand, other than the `}` character that indicates the *format-spec* is complete, it should report an error. The way to do this is complicated by the need to allow the function to be called at compile time. Before that change was made, it would be normal to throw a `std::format_error` exception. You can still do this, with the proviso that the compiler will report an error, as `throw` cannot be used when evaluating the function at compile time. Until such time as a workaround has been found for this problem, it is probably best to just throw the exception and allow the compiler to complain. That is the solution used in the code presented in this article.

If the whole *format-spec* is processed with no errors, the function should return an iterator pointing to the terminating `}` character. This is an important point – the `}` is not part of the *format-spec* and should not be consumed, otherwise the formatting functions themselves will throw an error.

Format specification mini-language

The *format-spec* for your type is written in a mini-language which you design. It does not have to look like the one for the standard *format-specs* defined by `std::format`. There are no rules for the mini-language, as long as you can write a `parse` function that will process it.

An example of a specialist mini-language is that defined by `std::chrono` or its formatters, given for instance at [CppRef]. Further examples are given in the code samples that make up the bulk of this series of articles. There are some simple guidelines to creating a mini-language in the appendix at the end of this article: ‘Simple Mini-Language Guidelines’.

The format function

The `format` function does the work of actually outputting the value of the argument for the replacement field, taking account of the *format-spec* that the `parse` function has processed.

Assume we have a typedef `FC` defined as follows:

```
using FC = basic_format_context<Out, charT>;
```

where `Out` is an output iterator and `charT` is the template argument passed to the `formatter` template. Then the `format` function prototype looks like the following:

```
FC::iterator format(const T& t, FC& fc) const;
```

where `T` is the template argument passed to the `formatter` template.

Note that the `format` function should be `const`-qualified. This is because the standard specifies that it can be called on a `const` object.

---

[1] There is nothing stopping you storing the formatting information in a class variable or even a global variable, but the standard specifies that the output of the `format` function in the `formatter` should only depend on the input value, the locale, and the format-spec as parsed by the last call to `parse`. Given these constraints, it is simpler to just store the formatting information in the `formatter` object itself.

> If you need more complex formatting than just writing one or two characters, the easiest way to create the output is to use the formatting functions already defined by std::format

The standard defines specialisations of the `basic_format_context` template called `format_context` and `wformat_context`, with `charT` being `char` and `wchar_t` respectively.

The function should format the value `t` passed to it, using the formatting information stored by `parse`, and the locale returned by `fc.locale()` if it is locale-dependent. The output should be written starting at `fc.out()`, and on return the function should return the iterator just past the last output character.

If you just want to output a single character, the easiest way is to write something like the following, assuming `iter` is the output iterator and `c` is the character you want to write:

```
*iter++ = c;
```

If you need more complex formatting than just writing one or two characters, the easiest way to create the output is to use the formatting functions already defined by `std::format`, as they correctly maintain the output iterator.

The most useful function to use is `std::format_to`, as that takes the iterator returned by `fc.out()` and writes directly to it, returning the updated iterator as its result. Or if you want to limit the amount of data written, you can use `std::format_to_n`.

Using the `std::format` function itself has a couple of disadvantages. Firstly it returns a string which you would then have to send to the output. And secondly, because it has the same name as the function in `formatter`, you have to use a `std` namespace qualifier on it, even if you have a `using namespace std;` line in your code, as otherwise function name resolution will pick up the `format` function from the formatter rather than the `std::format` one.

Formatting a simple object

For our first example we are going to create a `formatter` for a simple `Point` class, defined in Listing 2.

Default formatting

Listing 3 shows the first iteration of the `formatter` for `Point`. This just allows default formatting of the object.

```
class Point
{
public:
  Point() {}
  Point(int x, int y) : m_x(x), m_y(y) {}

  const int x() const { return m_x; }
  const int y() const { return m_y; }

private:
  int m_x = 0;
  int m_y = 0;
};
```

Listing 2

In the `parse` function, the lambda `get_char` defined in line 1 acts as a convenience function for getting either the next character from the *format-spec*, or else indicating the *format-spec* has no more characters by returning zero. It is not strictly necessary in this function as it is only called once, but will be useful as we extend the *format-spec* later.

The `if`-statement in line 2 checks that we have no *format-spec* defined. The value 0 will be returned from the call to `get_char` if the `begin` and `end` calls on `parse_ctx` return the same value.

The `format` function has very little to do – it just returns the result of calling `format_to` with the appropriate output iterator, format string, and details from the `Point` object. The only notable thing to point out is that we wrap the `format_ctx.out()` call which gets the output iterator

```
#include "Point.hpp"
#include <format>
#include <iostream>
#include <type_traits>

template<>
struct std::formatter<Point>
{
  constexpr auto parse(
    std::format_parse_context& parse_ctx)
  {
    auto iter = parse_ctx.begin();
    auto get_char = [&]() { return iter
      != parse_ctx.end() ? *iter : 0; };  // 1
    char c = get_char();
    if (c != 0 && c != '}') // 2
    {
      throw std::format_error(
        "Point only allows default formatting");
    }
    return iter;
  }
  auto format(const Point& p,
    std::format_context& format_ctx) const
  {
    return std::format_to(std::move(
      format_ctx.out()), "{},{}", p.x(), p.y());
  }
};
int main()
{
  Point p;
  std::cout << std::format("{0} [{0}]\n", p);
  try
  {
    std::cout << std::vformat("{0:s}\n",
      std::make_format_args(p));
  }
  catch (std::format_error& fe)
  {
    std::cout << "Caught format_error : "
      << fe.what() << "\n";
  }
}
```

Listing 3

we now have to store information derived from the format-spec by the parse function so the format function can do its job

in **`std::move`**. This is in case the user is using an output that has move-only iterators.

Adding a separator character and width specification

Now we have seen how easy it is to add default formatting for a class, let's extend the format specification to allow some customisation of the output.

The format specification we will use has the following form:

[sep] [width]

where *sep* is a single character to be used as the separator between the two values in the **`Point`** output, and *width* is the minimum width of each of the two values. Both elements are optional. The *sep* character can be any character other than **`}`** or a decimal digit.

The code for this example is in Listing 4.

Member variables

The first point to note is that we now have to store information derived from the *format-spec* by the **`parse`** function so the **`format`** function can do its job. So we have a set of member variables in the **`formatter`** defined from line `10` onwards.

The default values of these member variables are set so that if no *format-spec* is given, a valid default output will still be generated. It is a good idea to follow the same principle when defining your own **`formatter`**s.

The parse function

The **`parse`** function has expanded somewhat to allow parsing of the new *format-spec*. Line `1` gives a short-circuit if there is no *format-spec* defined, leaving the formatting as the default.

```
#include "Point.hpp"
#include <format>
#include <iostream>

using namespace std;

template<>
struct std::formatter<Point>
{
  constexpr auto parse(
    format_parse_context& parse_ctx)
  {
    auto iter = parse_ctx.begin();
    auto get_char = [&]() { return iter
      != parse_ctx.end() ? *iter : 0; };
    char c = get_char();
    if (c == 0 || c == '}') // 1
    {
      return iter;
    }
    auto IsDigit = [](unsigned char uc) { return
      isdigit(uc); }; // 2
    if (!IsDigit(c)) // 3
    {
      m_sep = c;
      ++iter;
      if ((c = get_char()) == 0 || c == '}') //4
      {
        return iter;
      }
    }
    auto get_int = [&]() {  // 5
      int val = 0;
      char c;
      while (IsDigit(c = get_char())) // 6
      {
        val = val*10 + c-'0';
        ++iter;
      }
      return val;
    };
```

Listing 4

```
    if (!IsDigit(c)) // 7
    {
      throw format_error("Invalid format "
        "specification for Point");
    }
    m_width = get_int(); // 8
    m_width_type = WidthType::Literal;
    if ((c = get_char()) != '}')  // 9
    {
      throw format_error("Invalid format "
        "specification for Point");
    }
      return iter;
  }
  auto format(const Point& p,
    format_context& format_ctx) const
  {
    if (m_width_type == WidthType::None)
    {
      return
        format_to(std::move(format_ctx.out()),
        "{0}{2}{1}", p.x(), p.y(), m_sep);
    }
    return format_to(std::move(format_ctx.out()),
      "{0:{2}}{3}{1:{2}}", p.x(), p.y(), m_width,
      m_sep);
  }
private:
  char m_sep = ‘,’;   // 10
  enum WidthType { None, Literal };
  WidthType m_width_type = WidthType::None;
  int m_width = 0;
};
int main()
{
  Point p1(1, 2);
  cout << format("[{0}] [{0:/}] [{0:4}]"
    "[{0:/4}]\n", p1);
}
```

Listing 4 (cont'd)

> **Avoid having complicated constructions or interactions between different elements in your mini-language … it should be possible to parse it in a single pass**

In the code following the check above we need to check if the character we have is a decimal digit. The normal way to do this is to use `std::isdigit`, but because this function has undefined behaviour if the value passed cannot be represented as an `unsigned char`, we define lambda `IsDigit` at line 2 as a wrapper which ensures the value passed to `isdigit` is an `unsigned char`.

As mentioned above, any character that is not `}` or a decimal digit is taken as being the separator. The case of `}` has been dealt with by line 1 already. The `if`-statement at line 3 checks for the second case. If we don't have a decimal digit character, the value in `c` is stored in the member variable. We need to increment `iter` before calling `get_char` in line 4 because `get_char` itself doesn't touch the value of `iter`.

Line 4 checks to see if we have reached the end of the *format-spec* after reading the separator character. Note that we check for the case where `get_char` returns 0, which indicates we have reached the end of the format string, as well as the `}` character that indicates the end of the *format-spec*. This copes with any problems where the user forgets to terminate the replacement field correctly. The `std::format` functions will detect such an invalid condition and throw a `std::format_error` exception.

The `get_int` lambda function defined starting at line 5 attempts to read a decimal number from the *format-spec*. On entry `iter` should be pointing to the start of the number. The `while`-loop controlled by line 6 keeps reading characters until a non-decimal digit is found. In the normal case this would be the `}` that terminates the *format-spec*. We don't check in this function for which character it was, as that is done later. Note that as written, the `get_int` function has undefined behaviour if a user uses a value that overflows an `int` – a more robust version could be written if you want to check against users trying to define width values greater than the maximum value of an `int`.

The check in line 7 ensures we have a width value. Note that the checks in lines 3 and 4 will have caused the function to return if we just have a *sep* element.

The width is read and stored in line 8, with the following line indicating we have a width given.

Finally, line 9 checks that we have correctly read all the *format-spec*. This is not strictly necessary, as the `std::format` functions will detect any failure to do so and throw a `std::format_error` exception, but doing it here allows us to provide a more informative error message.

The format function

The `format` function has changed to use the *sep* and *width* elements specified. It should be obvious what is going on, so we won't go into it in any detail.

Specifying width at runtime

In this final example we will allow the *width* element to be specified at runtime. We do this by allowing a nested replacement field to be used, specified as in the standard format specification with either `{}` or `{`*n*`}`, where *n* is an argument index.

The format specification for this example is identical to the one above, with the addition of allowing the width to be specified at runtime.

The code for this example is in Listing 5. When labelling the lines in this listing, corresponding lines in Listing 4 and Listing 5 have had the same labels applied. This does mean that some labels are not used in Listing 5 if there is nothing additional to say about those lines compared to Listing 4. We use uppercase letters for new labels introduced in Listing 5.

```
#include "Point.hpp"
#include <format>
#include <iostream>
using namespace std;
template<>

struct std::formatter<Point>
{
  constexpr auto
    parse(format_parse_context& parse_ctx)
  {
    auto iter = parse_ctx.begin();
    auto get_char = [&]() { return iter
      != parse_ctx.end() ? *iter : 0; };
    char c = get_char();
    if (c == 0 || c == '}')
    {
      return iter;
    }
    auto IsDigit = [](unsigned char uc)
      { return isdigit(uc); };
    if (c != '{' && !IsDigit(c))    // 3
    {
      m_sep = c;
      ++iter;
      if ((c = get_char()) == 0 || c == '}')
      {
        return iter;
      }
    }
    auto get_int = [&]() {
      int val = 0;
      char c;
      while (IsDigit(c = get_char()))
      {
        val = val*10 + c-'0';
        ++iter;
      }
      return val
    };
    if (!IsDigit(c) && c != '{')  // 7
    {
      throw format_error("Invalid format "
        "specification for Point");
    }
```

Listing 5

```
    if (c == '{')  // A
    {
      m_width_type = WidthType::Arg; // B
      ++iter;
      if ((c = get_char()) == '}')  // C
      {
        m_width = parse_ctx.next_arg_id();
      }
      else  // D
      {
        m_width = get_int();
        parse_ctx.check_arg_id(m_width);
      }
      ++iter;
    }
    else  // E
    {
      m_width = get_int();    // 8
      m_width_type = WidthType::Literal;
    }
    if ((c = get_char()) != '}')
    {
      throw format_error("Invalid format "
        "specification for Point");
    }
    return iter;
  }
  auto format(const Point& p,
    format_context& format_ctx) const
  {
    if (m_width_type == WidthType::None)
    {
      return
        format_to(std::move(format_ctx.out()),
        "{0}{2}{1}", p.x(), p.y(), m_sep);
    }
    if (m_width_type == WidthType::Arg) // F
    {
      m_width = get_arg_value(format_ctx,
        m_width);
    }
    return format_to(std::move(format_ctx.out()),
      "{0:{2}}{3}{1:{2}}", p.x(), p.y(), m_width,
      m_sep);
  }
private:
  int get_arg_value(format_context& format_ctx,
    int arg_num) const // G
  {
    auto arg = format_ctx.arg(arg_num); // H
    if (!arg)
    {
      string err;
      back_insert_iterator<string> out(err);
      format_to(out, "Argument with id {} not "
        "found for Point", arg_num);
      throw format_error(err);
    }
    int width = visit_format_arg([]
        (auto value) -> int { // I
      if constexpr (
        !is_integral_v<decltype(value)>)
      {
        throw format_error("Width is not "
          "integral for Point”);
      }
      else if (value < 0
        || value > numeric_limits<int>::max())
      {
        throw format_error("Invalid width for "
          Point");
      }
      else
      {
        return value;
      }
    }, arg);
    return width;
  }
```

Listing 5 (cont'd)

```
private:
  mutable char m_sep = ',';
  enum WidthType { None, Literal, Arg };
  mutable WidthType m_width_type
    = WidthType::None;
  mutable int m_width = 0;
};
int main()
{
  Point p1(1, 2);
  cout << format(
    "[{0}] [{0:-}] [{0:4}] [{0:{1}}]\n", p1, 4);
  cout << format(
    "With automatic indexing: [{:{}}]\n", p1, 4);
  try
  {
    cout << vformat("[{0:{2}}]\n",
      std::make_format_args(p1, 4));
  }
  catch (format_error& fe)
  {
    cout << format("Caught exception: {}\n",
      fe.what());
  }
}
```

Listing 5 (cont'd)

Nested replacement fields

The standard *format-spec* allows you to use nested replacement fields for the*width* and *prec* fields. If your *format-spec* also allows nested replacement fields, the **`basic_format_parse_context`** class has a couple of functions to support their use: **`next_arg_id`** and **`check_arg_id`**. They are used in the **`parse`** function for Listing 5, and a description of what they do will be given in that section.

The parse function

The first change in the **`parse`** function is on line `3`. As can be seen, in the new version, it has to check for the **`{`** character as well as for a digit when checking if a width has been specified. This is because the dynamic width is specified using a nested replacement field, which starts with a **`{`** character.

The next difference is in line `7`, where we again need to check for a **`{`** character as well as a digit to make sure we have a width specified.

The major change to this function starts at line `A`. This **`if`**-statement checks if the next character is a **`{`**, which indicates we have a nested replacement field. If the test passes, line `B` marks that we need to read the width from an argument, and then we proceed to work out what the argument index is.

The **`if`**-statement in line `C` checks if the next character is a **`}`**, which means we are using automatic indexing mode. If the test passes, we call the **`next_arg_id`** function on **`parse_ctx`** to get the argument number. That function first checks if manual indexing mode is in effect, and if it is it throws a **`format_error`** exception, as you cannot mix manual and automatic indexing. Otherwise, it enters automatic indexing mode and returns the next argument index, which in this case is assigned to the **`m_width`** variable.

If the check in line `C` fails, we enter the **`else`**-block at line `D` to do manual indexing. We get the argument number by calling **`get_int`**, and then we call the **`check_arg_id`** function on **`parse_ctx`**. The function checks if automatic indexing mode is in effect, and if so it throws a **`format_error`** exception. If automatic indexing mode is not in effect then **`check_arg_id`** enters manual indexing mode.

The **`else`**-block starting at line `E` just handles the case where we have literal width specified in the *format-spec*, and is identical to the code starting at line `8` in Listing 4.

Note that when used at compile time, **`next_arg_id`** or **`check_arg_id`** check that the argument id returned (for **`next_arg_id`**) or supplied (for

check_arg_id) is within the range of the arguments, and if not will fail to compile. However, this is not done when called at runtime.

The format function

The changes to the **format** function are just the addition of the **if**-statement starting at line `F`. This checks if we need to read the width value from an argument, and if so it calls the **get_arg_value** function to get the value and assign it to the **m_width** variable, so the **format_to** call following can use it.

The get_arg_value function

The **get_arg_value** function, defined starting at line `G`, does the work of actually fetching the width value from the argument list.

Line `H` tries to fetch the argument from the argument list. If the argument number does not represent an argument in the list, it returns a default constructed value. The following **if**-statement checks for this, and reports the error if required. Note that in your own code you might want to disable or remove any such checks from production builds, but have them in debug/testing builds.

If the argument is picked up correctly, line `I` uses the function **visit_format_arg** to apply the lambda function to the argument value picked up in line `H`. The **visit_format_arg** function is part of the **std::format** API. The lambda function checks that the value passed is of the correct type – in this case, an integral type – and that its value is in the allowed range. Failure in either case results in a **format_error** exception. Otherwise, the lambda returns the value passed in, which is used as the width.

Summary

We have seen how to add a **formatter** for a user-defined class, and gone as far as allowing the user to specify certain behaviour (in our case the width) at runtime. We will stop at this point as we've demonstrated what is required, but there is no reason why a real-life *Point* class couldn't have further formatting abilities added.

In the next article in the series, we will explain how you can write a formatter for a container class, or any other class where the types of some elements of the class can be specified by the user. ■

Appendix: Simple mini-language guidelines

As noted when initially describing the **parse** function of the formatters, the *format-spec* you parse is created using a mini-language, the design of which you have full control over. This appendix offers some simple guidelines to the design of your mini-language.

Before giving the guidelines, I'd like to introduce some terminology. These are not 'official' terms but hopefully will make sense.

- An *element* of a mini-language is a self-contained set of characters that perform a single function. In the standard *format-spec* most elements are single characters, except for the *width* and *prec* values, and the combination of *fill* and *align*.
- An *introducer* is a character that says the following characters make up a particular element. In the standard *format-spec* the '**.**' at the start of the *prec* element is an introducer.

Remember, the following are guidelines, not rules. Feel free to bend or break them if you think you have a good reason for doing so.

Enable a sensible default

It should be possible to use an empty *format-spec* and obtain sensible output for your type. Then the user can just write **{}** in the format string and get valid output. Effectively this means that every element of your mini-language should be optional, and have a sensible default.

Shorter is better

Your users are going to be using the mini-language each time they want to do non-default outputting of your type. Using single characters for the elements of the language is going to be a lot easier to use than having to type whole words.

Keep it simple

Similar to the above, avoid having complicated constructions or interactions between different elements in your mini-language. A simple interaction, like in the standard *format-spec* where giving an *align* element causes any subsequent '**0**' to be ignored, is fine, but having multiple elements interacting or controlling others is going to lead to confusion.

Make it single pass

It should be possible to parse the mini-language in a single pass. Don't have any constructions which necessitate going over the *format-spec* more than once. This should be helped by following the guideline above to '**Keep it simple**'. This is as much for ease of programming the **parse** function as it is for ease of writing *format-specs*.

Avoid ambiguity

If it is possible for two elements in your mini-language to look alike then you have an ambiguity. If you cannot avoid this, you need a way to make the second element distinguishable from the first.

For instance, in the standard *format-spec*, the *width* and *prec* elements are both integer numbers, but the *prec* element has '**.**' as an introducer so you can always tell what it is, even if no *width* is specified.

Use nested-replacement fields like the standard ones

If it makes sense to allow some elements (or parts of elements) to be specified at run-time, use nested replacement fields that look like the ones in the standard *format-spec* to specify them, i.e. **{** and **}** around an optional number.

Avoid braces

Other than in nested replacement fields, avoid using braces (`{` and `}`) in your mini-language, except in special circumstances.

References

[Collyer21] Spencer Collyer (2021) 'C++20 Text Formatting – An Introduction' in *Overload* 166, December 2021, available at: https://accu.org/journals/overload/29/166/collyer/

[CppRef] **std::formatter<std::chrono::systime>**: https://en.cppreference.com/w/cpp/chrono/system_clock/formatter

[P2216] P2216R3 – std::format improvements, Victor Zverovich, 5 Feb 2021, https://wg21.link/P2216

[P2918] P2918R2 – Runtime format strings II, Victor Zverovich, 7 Nov 2023, https://wg21.link/P2918

This article was previously published in *Overload* 180, in April 2024.

Afterwood

The plethora of names for our profession is an occupational hazard. Chris Oldwood considers those he's encountered.

A recurring (though some may say repetitive) theme of this column, and the software industry in general, is that of naming, and nowhere is this more apparent than in how we refer to ourselves on a professional level. For the most part it doesn't cross my mind, but every ten years in the UK we have a census and so are faced with struggling to fill in the box marked 'occupation'. It also used to be a perennial question from family and friends, but I think they all settled on 'something to do with computers' and just moved on. This latest test comes from looking for a new contract and being faced with the onslaught of questions from LinkedIn as I update my profile once again. It seems that despite the industry's newfound ability to create electricity guzzling machines which can write poetry, it's also lost the ability to do simple pattern recognition on free text and now I need to manually select any (and all) relevant matching job titles from a humungous list of seemingly similar entries. Do I really need to add both 'programmer' *and* 'computer programmer'? Have we also lost the ability to do fuzzy matching too?

Before I went to university back in the late 80's, I didn't really know that the messing around on my home computer I had done in my teens was even a career option I could explore. (I only found out after getting my first programming job that my late uncle had been a programmer in the 70's.) It felt natural at that time that someone who programs a computer would be known as a 'programmer', and I was happy to go with that until I met my (now) father-in-law. He informed me that a programmer was a lowly job and that I should be looking at the more superior position of *analyst/programmer* (which was the level he had obtained). In *his* experience (at a huge aerospace company) a programmer just typed in code provided by someone else, whereas he *also* did the thinking *and* design too! Luckily, he had someone else to do the testing though…

During that conversation, I began to see why the title of Software Engineer had probably come about and why I might need to be more careful about how I referred to a role on my CV. At that time, the list of responsibilities for a Software Engineer appeared to be more encompassing than for the traditional programmer roles, though in retrospect it was likely more due to the differences in environment – shrink-wrapped desktop applications versus in-house mainframe programs. Either way, being a Software Engineer got my father-in-law off my back by appearing more honourable.

In some countries you can only call yourself an engineer if you have an official qualification or recognition from a formal body. I do have a degree in an engineering subject, so I felt a little justified in adopting that title, but I can also see why Software Developer might have sprung up because of the backlash against the use of engineer for what is an incredibly young industry by comparison, despite it being used at the NATO conference back in 1968.

Once I became established as a freelancer and was freed from the shackles of HR I went with the flow and adopted the developer moniker because I liked the terseness of 'dev' in direct contrast to the rise of 'title inflation' that seemed to be taking hold, at least in financial circles. I once remember interviewing a candidate who was Vice President of Some American Bank. In my head, a vice-president was second only to a president, and, like the Highlander, there can be only one. Apparently not: the bank had a gazillion vice-presidents, directors, chiefs, etc. This departure from traditional titles means a Property Developer now sounds like someone who only writes getters-and-setters.

In the intervening years the variation in job titles has increased, which shouldn't be surprising as the nature of the job itself has changed substantially over time. We write code for so many different reasons now, which are not necessary part of any product or service delivered to an end user – programming is quite often a means to another end. I recently spent a couple of years largely working on the build pipeline for an in-house programming language and quipped on my socials that I should change my job title to 'Software Plumber' on account of the time I was spending fixing pipelines of various kinds. I also suggested growing a moustache and wearing dungarees into the office. I was quickly informed that 'Build Engineer' was the correct term for that particular role.

Some jobs might have *had* engineer or developer in the job advert but when you get there you can find that it's largely about shifting data from one place to another. I was tempted to use Data Hygienist to describe one role as I spent so much time trying to clean up the data from a bunch of legacy back-end systems to avoid polluting the internals of our codebase with special case logic. It was on the same project that our illustrious *CVu* editor coined the term Data Marshaller to reflect his role of shepherding data to/from the database. I suggested he probably needed to wear a high-viz jacket and hard hat too, though falling asleep at the keyboard was the only real hazard.

I've always liked Chris O'Dell's notion of a Software Gardener as it felt like it really captured the organic nature of modern systems development. When you get in early and can let the design emerge and blossom, it's incredibly satisfying, plus there is always some pruning to be done. Gardener would also be a suitable name for those people who just seem to spread crap all over the codebase.

And that metaphor provides the perfect segue into the world of AI-based tooling. Programming was never just about writing code – despite where the job market might once have positioned it – it was always about solving problems, typically by producing software, and that includes a whole bunch of other skills. Although I might have to jump on the bandwagon at some point to remain competitive, I never want to lose the ability to handcraft code the old-fashioned way. I look forward to the day when I can update my LinkedIn profile to Artisan Programmer.

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
professionalism in programming
Monthly journals
Annual conference
Discussion lists
To find out more, visit accu.org

“The magazines”

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

“The conferences”

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

“The community”

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

“The online forums”

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe