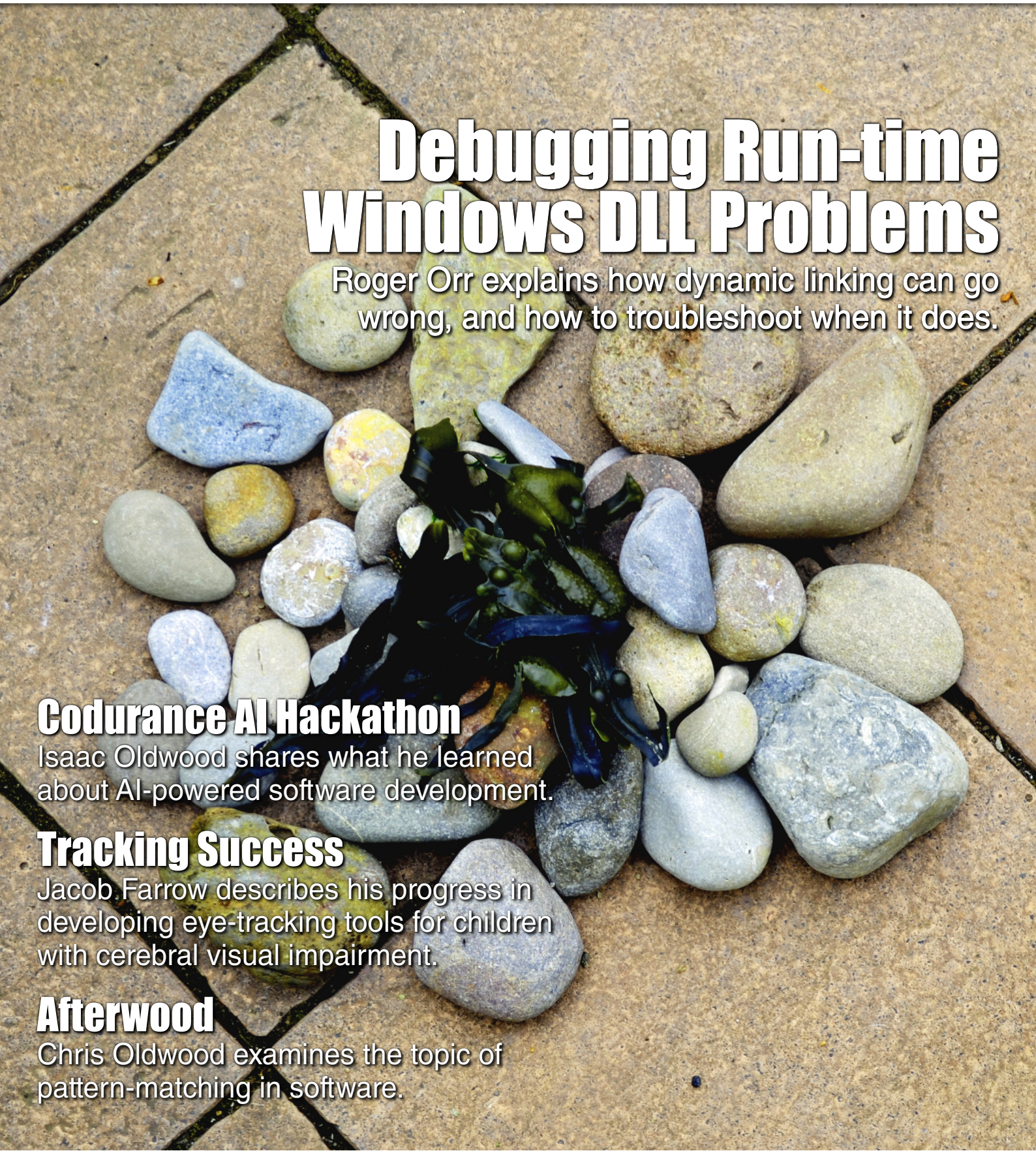
overload 187

JUNE 2025 £4.50



Debugging Run-time Windows DLL Problems

Roger Orr explains how dynamic linking can go wrong, and how to troubleshoot when it does.

Codurance AI Hackathon

Isaac Oldwood shares what he learned about AI-powered software development.

Tracking Success

Jacob Farrow describes his progress in developing eye-tracking tools for children with cerebral visual impairment.

Afterwood

Chris Oldwood examines the topic of pattern-matching in software.

A magazine of ACCU

ISSN: 1354-3172

To connect with like-minded people visit accu.org

accu

June 2025
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

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

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Kevlin Henny.

ACCU

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

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

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

Copy deadlines

All articles intended for publication in *Overload* 188 should be submitted by 1st July 2025 and those for *Overload* 189 by 1st September 2025.

Copyrights and trademarks

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

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

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

Eliminate the Impossible

Some things are – or seem to be – impossible. Frances Buontempo explores how to distinguish between the two.

I haven't manage to think of an editorial topic, so yet again, sorry. There are so many things I could write about, but I don't want to cover old ground and don't have the bandwidth to spend ages learning new topics at the moment. I am currently trying to rein in my commitments. I say "Yes" far too often, and am now starting to realise I can't do "all the things". Trying to limit the choice of what to do is difficult. I am tending to postpone some things, and they eventually fall off a TODO list. Not a great strategy, but a strategy nonetheless.

Trying to eliminate things is difficult. The "You ain't gonna need it"(YAGNI) mantra from Extreme Programming encourages us to avoid creating things we don't need now. Martin Fowler wrote about YAGNI [Fowler15], comparing the cost of building now versus building later. Sometimes delay has a cost, but doing things now costs, too. He says, YAGNI:

> doesn't mean to forego all abstractions, but it does mean any abstraction that makes it harder to understand the code for current requirements is presumed guilty.

For example, it's OK to build an abstraction, if that makes code easier to change. He points out:

> Yagni only applies to capabilities built into the software to support a presumptive feature, it does not apply to effort to make the software easier to modify.

Maybe the phrase "Never say never" is relevant? Trying to eliminate unneeded code, or anything unneeded, is sensible, as is avoiding wasting time on planning for something that won't happened. However, predicting the future is difficult. I prepared a workshop last year for a conference, but the conference got cancelled. That was frustrating, but I can use the materials for a different conference.

Now, consider the Sherlock Holmes quote, "Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth." This presupposes you have an exhaustive list that includes the truth. This is called a Holmesian fallacy [RationalWiki]: believing one explanation because the others you have thought of are impossible. The rational wiki (op cit) gives an example from Thales of Miletus: "The lodestone has a soul because it moves iron. This proves that all things are full of gods." That might not be the best example, since I suspect a non-corporeal substance like a soul cannot move something physical. A better example might be C++ programmers arguing over undefined behaviour (UB). You often see people asking questions about strange behaviour in code, for example getting the right numbers from code compiled with one compiler, but not from another. That code seems to work sometimes leads to the claim they can't have UB, otherwise why is it OK in some circumstances? Of course, that's not how UB works.

Furthermore, the history of science and mathematics is littered with examples of impossible things becoming possible. What's the square root of a negative number? Initially regarded as impossible, allowing the possibility opens up new mathematics. I have written about complex numbers before [Buontempo24]. Pythagoras believed all numbers were rational. A story goes that Hippasus of Metapontum, a member of Pythagoras' group, demonstrated that the length of the diagonal of a square of side length 1 is the square root of 2, which is not rational (the length, not the proof) [Cambridge]. He was kicked out for heresy. Pythagoras thought everything in nature must be based on whole numbers, so did not approve. Mind you, Pythagoras also held that 1 is not a number, because it represents a singularity rather than a plurality [Britannica], and believed you shouldn't eat beans because they have a soul. (You've heard of jumping beans, I presume? They move, so like the lodestone, must have a soul.)

Many things are now possible on computers that would have been unthinkable years ago. The rise of deep learning needed much faster processors and much more memory. The precise requirements vary, but for example consider a 50 layer network with about 26 million weight parameters and about 16 million activations in the forward pass. Using a 32-bit floating-point value to store each weight and activation gives a total storage requirement of 168 MB [Hanlon17]. Lots of research is focused on speeding up the calculations, or running algos on GPUs, or even building specialized hardware, but maybe we need to step back and find a completely different algorithm? The power requirements and excessive use of water for cooling in data centres worries me as well. Perhaps we should eliminate resource hungry methods? Doing so might also reduce costs. I realise I am in danger of expressing opinions now, which would take me dangerously near to an editorial! Which would, of course, be impossible. Let's eliminate that immediately.

Stepping back and thinking through why you believe something is impossible can be useful. You might not invent a new branch of mathematics, or find a new computing algorithm, but you might discover a different approach. Alternatively, you might find you can manage something you thought you couldn't do. This can happen when you try to learn something. We all have blind spots, or certain things we find difficult to get our heads round. Some people panic at the sight of numbers, but discovering how to deal with a small part of a big scary topic helps. A thousand mile journey begins with the first step, as they say. You might discover you can manage something, even if you are neither very good at it nor enjoy it. GUI work is my mental block. I can write a front end program, but I'd rather not. I'm also trying to learn German

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

on Duolingo. I didn't do very well at foreign languages at school, and struggle to spell English words. In fact I just typed 'sturrgl eot'. I suspect I have dyslexia, which doesn't help. I used to think I would never be able to learn a different language or spell properly. I now realise I can try a different ways to phrase something if I get stuck. In school exercises, you often aren't allowed to do that. Finding a Plan B offers an alternative if Plan A is impossible. Eliminate the impossible, and what's left might work, you never know.

I gave a talk at *C++Online* called 'Don't be negative' [Buontempo25]. Why might you want to eliminate negative elements from a container or range? Well, maybe a negative price is implausible. Go give it a try in your favourite language. I used C++. The `std::remove_if` algorithm used to be a common interview question. As you probably know, this doesn't remove elements – the container stays the same size, but appropriate elements are shuffled to the front. There are newer, better ways, like `std::erase_if`. You can also try a recursive approach and more besides. You had to be there. It looks like you can get exclusive access to content if you can't wait for YouTube [C++Online]. I believe this is a great example, with a simple problem statement, but many valid approaches, as well as somewhat silly methods. Being silly often gets your imagination going, and can provide great learning opportunities.

People often use silly analogies to make points. Sometimes these are intended to ridicule others' points of view. For example, Bertrand Russell discussed the idea of a celestial teapot, too small to be seen, orbiting the sun between Earth and Mars. Hard to argue with, right? Because whatever you say to suggest there is no such teapot can be countered by pointing out there can't be any evidence because it is unobservable. Bertrand Russell's point was "the philosophic burden of proof lies upon a person making empirically unfalsifiable claims, as opposed to shifting the burden of disproof to others." [Wikipedia-1] Russell was talking about religion, but the logic applies more generally. When you eliminate the impossible, if what's left is unfalsifiable, Russell would say the person making the suggestion still has to prove it's true. Sherlock Holmes was wrong. Not everyone agrees with Russell's thought experiment. For example, the philosopher Paul Chamberlain countered, "every truth claim, whether positive or negative, has a burden of proof." Again, this would mean Sherlock Holmes is wrong.

Now, Sherlock Holmes is a fictional character, so shouldn't be taken as a source of authority. To be honest, many non-fictional characters shouldn't be taken as a source of authority either. Fiction can be useful though. Russell's teapot is one of many thought experiments. The dining philosophers problem [Wikipedia-2] is a good story for thinking through concurrency and deadlock problems. Five philosophers sit at a table, with a plate each. There is a fork between each plate, but eating from a pile of spaghetti requires two forks. The problem is to allow the philosophers to eat or think, while ensuring none starve. It's easy to end up with a deadlock, whereby philosophers starve. Setting the problem as a story makes it easier to visualize and discuss. I'm sure you can think of many other stories or thought experiments. Schrodinger's cat comes to mind too [Wikipedia-3]. Even if you don't understand the physics you have probably heard of the story. Is the cat both dead and alive until you look? Is that impossible? I'll leave that thought with you.

Stories can be a useful way of thinking about things. They can illustrate an abstract idea or help to compress a chain of thought. By 'compress' I mean pick out salient parts, rather than conveying everything. Maybe your CV is a work of fiction, to some extent? Not that you have made up roles, but have you tried to give it a narrative, emphasising relevant skills and experience for a specific roles? You eliminate the irrelevant, if you are as old as me. Fitting everything on two pages is difficult. If you don't have much experience, filling two pages is a different problem. Don't forget, if you write for *Overload* you can include that on your CV.

Some stories worry me, though. It's easy to come to unfounded conclusions if you follow Sherlock Holmes' statement. I notice myself thinking, 'Oh, perhaps they are annoyed because…' or 'That bug must be due to …' or similar. I suspect you do as well. If you think of something that's not impossible, that does not mean it is correct. I spent a long while working in finance. You saw reports called 'PnL Explain', which 'explained' the profit or loss on a balance sheet. Sometimes 'attribution' is used instead of explain. There is more than one way to calculate this, and you often end up with an 'unexplained' portion of profit or loss [Wikipedia-4]. These reports are useful for risk analysis, but the idea that an explanation might come with an unexplained part is of note. Another finance example involves validating financial models. You often value a complicated instrument based on something simple that you can find prices for in the markets. Your model should be able to reconstruct the values you get from the markets, but often doesn't do this precisely. On more than one occasion I have seen 'stories' told explaining why there are differences in the numbers, floating point inaccuracy being a common excuse. More than once, the team later found a bug in the code which more accurately explained the difference.

We all come to wrong conclusions from time to time. That's OK. Being humble enough to admit your mistakes and say sorry matters. Maybe going forward, let's try to notice if we have picked what's left when we eliminated the impossible, but may not have thought of everything possible. Or catch ourselves spotting a possible explanation: the first thing you think of to make sense of the world might not be correct. Being wrong is OK, but that's why we all need to bounce our ideas off people, get a code review, or sanity check with a review team.

References

[Britannica] 'Pythagoreanism', published by Britannica, available at: https://www.britannica.com/topic/number-symbolism/Pythagoreanism

[Buontempo24] Frances Buontemp 'Counting Quals' in *Overload* 184, published December 2024, available at: https://accu.org/journals/overload/32/184/buontempo/

[Buontempo25] Frances Buontempo 'Don't be negative', slides from a talk given at *C++Online* given on 27 February 2025 available from: https://cpponline.uk/session/2025/dont-be-negative/

[Cambridge] 'Death by number', published on *Underground Mathematics* by University of Cambridge, last updated 18 Jan 2016, available at: https://undergroundmathematics.org/thinking-about-numbers/death-by-number

[C++Online] Access to talks available from: https://cpponline.uk/on-demand-early-access-pass-now-available/

[Fowler15] Martin Fowler, 'Yagni', posted 26 May 2015 at https://martinfowler.com/bliki/Yagni.html

[Hanlon17] Jamie Hanlon 'Why is so much memory needed for deep neural networks?', published 31 January 2017 on *Graphcore*, available at: https://www.graphcore.ai/posts/why-is-so-much-memory-needed-for-deep-neural-networks

[RationalWiki] 'Homesian fallacy' at http://rationalwiki.org/wiki/Holmesian_fallacy

[Wikipedia-1] 'Russell's teapot', available at: https://en.wikipedia.org/wiki/Russell%27s_teapot

[Wikipedia-2] 'Dining philosphers problem', available at: https://en.wikipedia.org/wiki/Dining_philosophers_problem

[Wikipedia-3] 'Schrödinger's cat', available at: https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat

[Wikipedia-4] 'PnL explained', available at: https://en.wikipedia.org/wiki/PnL_explained

Debugging Run-time Windows DLL Problems

Dynamic linking can fail in various ways. Roger Orr explains what can go wrong and how to troubleshoot such problems.

Windows, in common with many other operating systems, supports 'late binding', where some or most of the symbols in an executable program are resolved at runtime from other files in the system. On Windows, these are known as 'Dynamic-Link Libraries'and a brief overview can be found, for example, in [Microsoft-1]. The Win32 system itself is accessed via entry points in (numerous) such dynamic link libraries (DLLs) and many applications are shipped as one or more executable programs (EXEs) and supporting DLLs.

While this approach provides a lot of benefits, likely already known to many of the readers of this article, it also adds an additional failure point in the running of the application.

With so-called 'static' linking all required symbols are located at link time and the actual code or data is bound into the executable file. The resultant binary is therefore complete in itself and all the code needed at execution time is guaranteed to be present (and the same as that at link time.) Dynamic linking in contrast may fail if one or more of the DLLs needed at runtime cannot be found, cannot be loaded, or do not contain the symbols that are expected. (Additionally, but not otherwise focussed on in this article, there are security issues arising from the way that the code to be executed is located and loaded at runtime – the code executed can be very different from the code the original program linked against.)

The design choice taken for Windows DLLs is that each late binding symbol is tied to a named dynamic library, and this name is in turn tied to the actual filename of the DLL that the loader finds on disk. Note that this is not the only design choice, and Linux for example made a different choice which has a slightly different set of benefits and issues.

What could possibly go wrong?

There are three broad categories of failure when resolving a late binding symbol:

1. The DLL cannot be located
2. The symbol cannot be resolved against the DLL found
3. There is a problem when loading the DLL into the process memory

Additionally, there are two contexts where late binding occurs: one is when the system loader *implicitly* resolves late binding symbols and the other is under program control when an application can request a DLL to be loaded into the running process and can attempt to resolve symbols in a loaded DLL. This is not a hard separation as these two contexts overlap, when for example an application requests a DLL that itself has late binding symbols or makes use of the Microsoft 'delay loading" mechanism [Microsoft-2].

There are two main differences between these contexts. Firstly, in the former case any failure is fatal whereas in the latter case the program will receive an error code from the failed call and so some sort of recovery or remediation can be attempted. Secondly, the former case is in principle discoverable statically from the information in the headers of the executable and the DLLs whereas the second case requires an actual program execution as the behaviour is only evidenced at runtime.

These differences also affect the diagnosis when problems occur, as we shall see later on.

Note that *this* article doesn't cover the process of *building* DLLs on Windows.

Roger Orr Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.co.uk

What does Windows usually report?

Often (depending on a variety of factors outside the scope of this article) Windows will produce a simple error dialog when there are problems with the implicit resolution for a late binding symbol.

The first example (Figure 1) is when `DllNotFound.exe` is executed and the dependent DLL `MissingDll.dll` cannot be located.

This dialog does helpfully tell us the name of the DLL that is not found, but I must confess I have very rarely found that this problem can be fixed by reinstalling the program. Your experience may be different!

When under programmatic control, using **`LoadLibrary`** or **`LoadLibraryEx`**, the failure is indicated by returning a **`NULL`** handle to the loaded module and the actual underlying error can be obtained from **`GetLastError()`**; it is usually 126 which is defined as **`ERROR_MOD_NOT_FOUND`**.

While often this is enough to identify the problem, we get no information that might help with identifying the DLL that could not be located in the case where it was not the actual library we were trying to load that could not be found, but one of *its* dependent libraries.

The second example is when the DLL is found, but the required export is not present. For this to occur, the DLL found at load time must be different from the DLL that is associated with the library (LIB) file used when the executable was created (see Figure 2, next page). This dialog gives us the so-called 'decorated name' (also known as the 'mangled name') of the symbol we are loading – see [Microsoft-3] for more details – but it does *not* show the name of the DLL in which this symbol was expected to be found. Undecorating (or demangling) the name is easy (although it is probably not actually necessary in this case!) as fortunately we can

Figure 1

Looking for a missing symbol requires three things: the symbol being requested, the name of the DLL expected to provide this symbol, and the list of symbols actually exported from the target DLL

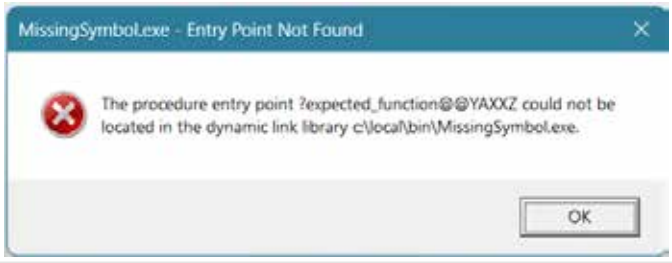


Figure 2

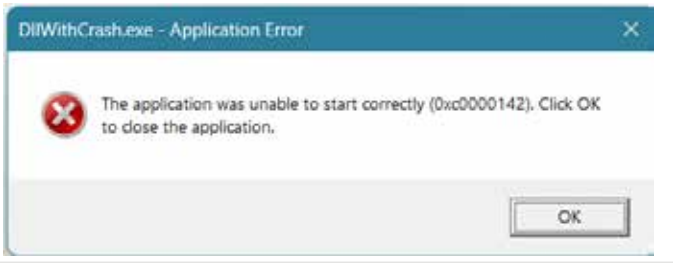


Figure 3

copy and paste from the dialog using Ctrl+C / Ctrl+V and then run the **undname** program provided with Visual Studio to turn this symbol into the C++ symbol we are looking for:

```
---------------------------
MissingSymbol.exe - Entry Point Not Found
---------------------------
The procedure entry point
?expected_function@@YAXXZ could not be located
in the dynamic link library
c:\local\bin\MissingSymbol.exe.
---------------------------
OK
---------------------------
```

Then **undname ?expected_function@@YAXXZ** produces:

```
Undecoration of :- "?expected_function@@YAXXZ"
is :- "void __cdecl expected_function(void)"
```

You can also call the function **UnDecorateSymbolName** from the header `DbgHelp.h` to undecorate names under program control.

Note that the decoration is MSVC specific; other implementors' C and C++ compilers may or may not use the same scheme. Additionally, note that the decoration scheme used by MSVC compilers does gradually change over time to support new features in the language (although we've had a long period of relative stability with no significant changes since VS 2015 and the last (minor) change being in VS 2019 version 16.10.)

The equivalent programmatic mechanism is to call **GetProcAddress**; this takes a module handle from a DLL previously loaded into the process address space and the name of the symbol to be loaded. The failure is indicated by returning **NULL** as the address of the symbol and the actual underlying error can be obtained from **GetLastError()** (as above); it is usually 127 which is defined as **ERROR_PROC_NOT_FOUND**.

Finally, if the DLL fails to *load* then you are likely to see something like Figure 3 produced.

This dialog is less useful than the first two since it gives no indication of *which* initialization routine failed.

The equivalent programmatic error returned from **LoadLibrary** or **LoadLibraryEx** is error code 1114, which is defined as **ERROR_DLL_INIT_FAILED**.

Note that the error code in the dialog box, 0xc0000142, is defined as **STATUS_DLL_INIT_FAILED** in `NtStatus.h` and can be mapped to **ERROR_DLL_INIT_FAILED** via the **RtlNtStatusToDosError** function, defined in the header `winternl.h`.

'Manual' detective work

When one of the three errors occurs, we can check things by hand to try and identify the root cause of the failure.

We can look for a missing DLL by searching for the corresponding DLL filename and making sure that it can be found by the loader. However, working out exactly where the loader is going to look can be complex as there are numerous flags and options that change the actual path used by the system to locate DLLs. The full details are listed in [Microsoft-4], which is not an easy read – there are lots of factors to consider.

Fortunately, for many common cases it is enough if the target DLL is in the `system32` (64-bit programs) or `syswow64` (32-bit programs) directory underneath **%SystemRoot%** (typically `C:\Windows`), in the directory of the application executable, or somewhere along the **%PATH%**. (Note: the counterintuitive 64-bit directory name `system32` was retained for backwards compatibility with the original 32-bit Windows NT, even though it now contains 64-bit DLLs. In addition, for extra fun, Windows provides transparent file redirection from `system32` to `syswow64` for 32-bit programs: see [Microsoft-5] for more on this. If you find this confusing you are not alone.)

Looking for a missing symbol requires three things: the symbol being *requested*, the *name* of the DLL expected to provide this symbol, and the list of symbols actually *exported* from the target DLL. One way to obtain this information is by using the **dumpbin** program that comes with Visual Studio twice, once on the requesting binary and once on the target DLL.

For the example above of a missing symbol, this gives the information in Listing 1 (on next page).

We then need to locate the actual `ChangedExports.dll` that the application tried to load and then run **dumpbin** again, this time with the **/exports** switch to see what symbols the library *offers* (see Listing 2, also on next page).

```
c:> dumpbin /imports C:\local\bin\MissingSymbol.
exe
...

  Section contains the following imports:

  ChangedExports.dll
     14000E000 Import Address Table
     14000E458 Import Name Table
               0 time date stamp
               0 Index of first forwarder reference
                     0 ?expected_function@@YAXXZ
...
```

Listing 1

```
c:> dumpbin /exports C:\local\bin\ChangedExports.
dll
...
   ordinal hint RVA      name

         1    0 000010B9 ?renamed_function@@YAXXZ
= @ILT+180(?renamed_function@@YAXXZ)

Summary
...
```

Listing 2

Since the list of exported symbols only contains **renamed_function** and not **expected_function**, we can immediately see what the problem is.

Finally, the case where the DLL fails to initialize. This can be very hard to identify as there's little that can be done statically to identify *which* of the potentially large number of dependent DLLs was the one with the failing initialization routine; if you are fortunate, the problem is an access violation or an exception that you can find relatively easily using a debugger.

But surely there must be some better ways to do this than this sort of manual investigation?

Viewing the dependencies

What NOT to use

Many years ago Microsoft used to ship a GUI tool for viewing dependencies, `depends.exe`. This tool was later made freely available from its own website [DepWalker]. Unfortunately, the website stopped updating at version 2.2 which reports under 'What is New in Version 2.2' that it covers "... Updated internal information about known OS versions, build numbers, and flags up to the Vista RC1 build." (The tool's 'About' page proudly reports that it was built on 29 Aug 2006!) Microsoft's web site recommends using this tool only on Windows 8 or before. Many people do still try to use this tool, but it has not stood the test of time very well. In particular, recent versions of Windows have added 'API Set' [Microsoft-6] functionality, which is effectively a way to provide a platform-dependent virtual alias for a real DLL. These 'virtual DLLs' cause various problems for older tools that are completely unaware of their existence, like **depends**, and attempt to process them like normal DLL names.

For example, if I try to run depends.exe 2.2.6000 on Windows 11 to look at the dependencies for `DllNotFound.exe`, there are two problems. First, it takes over eight minutes to run on my computer (with one thread being 100% busy) and secondly, while it does successfully report that the **MissingDll** is missing, it *also* reports numerous false positives. See Figure 4.

These two problems make it of rather limited use on current versions of Windows especially for programs more complex than this almost trivial example program.

A potentially better tool

One recommended tool with similar functionality is 'Dependencies' available on GitHub [GitHub]. However, the last commit was back in Nov 2021 so it does not appear to be being maintained any longer.

It does at least *partly* understand the newer API Set entries in the module headers, and so if I run this tool against `MissingSymbol.exe`, it does

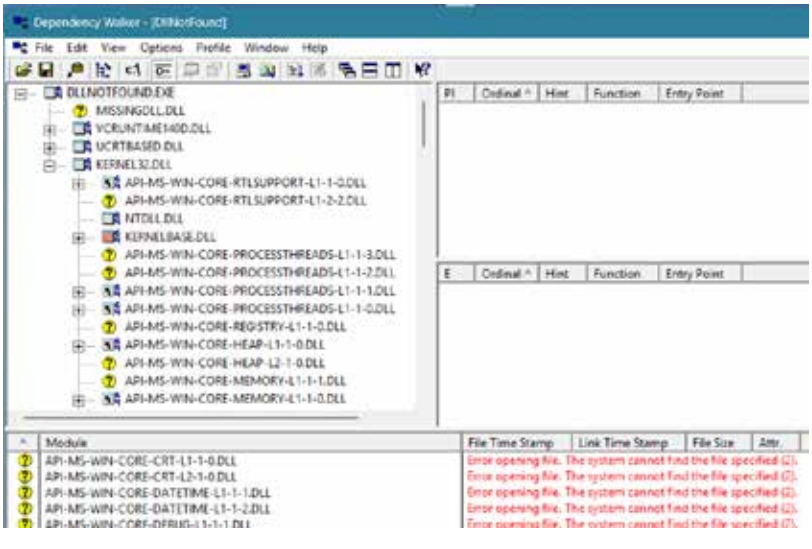


Figure 4

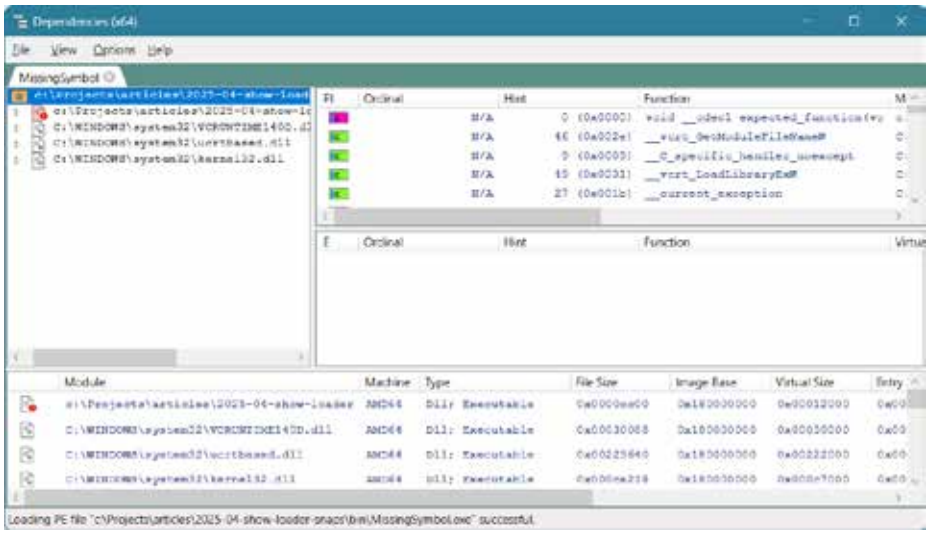


Figure 5

quickly identify the missing symbol and the DLL containing it (see Figure 5).

This works well for 'top level' problems that are visible in the default view, which lists direct child DLLs. For more complex applications with many DLLs and deep dependency trees it seems hard to use this view to find the actual problems as you need to manually expand each node in the tree in turn until you find a node showing a problem. This can take quite a while and is a very manual process.

There are two other options listed in the Tree Build behaviour dialog accessed from ***Options*** > ***Properties***: ***RecursiveOnlyOnDirectImports*** and ***Recursive***. The first option may help with locating the problem as it does provide a more complete list of DLLs in the lower pane. However, there does not appear to be a way to display errors so

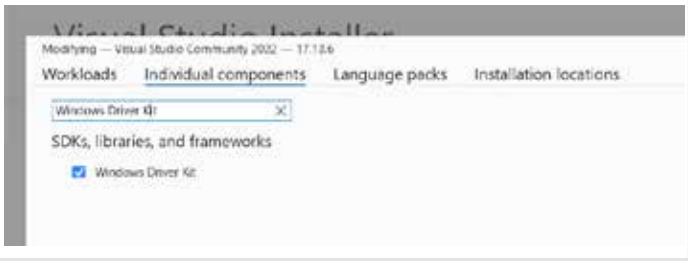


Figure 6

you have to scroll through the (potentially rather long) list of DLLs and API Sets looking for a warning icon. The second option does appear to do more work but it consumed **9 GB** of RAM during the process and, like Dependency Walker, consumed **lots** of CPU doing so (it actually used *all* my cores for about 20 minutes).

So, while this tool can be of some help in examining the dependency tree of an application, it does not appear to offer a simple solution to finding problems with the dependencies.

Other problems with dependency viewing tools

Since both tools are performing external analysis of dependencies in the target binary, they suffer from some inevitable issues with analysing problems occurring at runtime and also with problems that are related to the precise path being used to search for dependent DLLs. The older tool, `depends.exe`, did actually offer a way to attempt runtime diagnosis and this could be successfully used back in the pre-Windows 10 days, with some restrictions. The newer tool lists under 'Limitations' that: "Dynamic loading via LoadLibrary are not supported (and probably won't ever be)."

Using the loader itself

Fortunately there is a better way to debug loader problems than by analysing the program from the outside. The Microsoft loader itself contains diagnostic code that can be configured to print out information about the loading process *as it occurs*.

This setting goes by the name of ***Show Loader Snaps***. (I believe the 'Snaps' in this phrase refers to the short status messages it produces.) When this setting is enabled for a process the loader will provide extra diagnostic information *to an attached debugger* for the actions it takes while loading DLLs and resolving symbols; whether implicitly or by calling functions like **`LoadLibrary`**. The output appears in the debugger in the same way that output from calls to **`OutputDebugString`** does. However, the actual mechanism used in the loader is subtly different from an actual call to **`OutputDebugString`** and unfortunately, *unlike* output from **`OutputDebugString`**, there does not appear to be any way to view the loader snap information using other tools, such as DebugView from SysInternals.

Enabling 'Show Loader Snaps' for a process

The official way to use this is to use the `GFlags.exe` program that is part of Debugging Tools For Windows [Microsoft-7] – I chose the route of simply asking for ***Windows Driver Kit*** as an Individual Component as part of my VS 2022 installation (see Figure 6).

You then run `gflags.exe` – which requires Admin rights – and go to the ***Image File*** tab. Type the base name of the executable into the entry field and press TAB. You can then enable the flag, and click ***OK*** (or ***Apply***) – see Figure 7.

What this actually does is to write a value to the registry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options\
DllNotFound.exe
  GlobalFlag    REG_DWORD    0x2
```

You can of course write the same value using any other tool of your choice, you are not required to use GFlags.

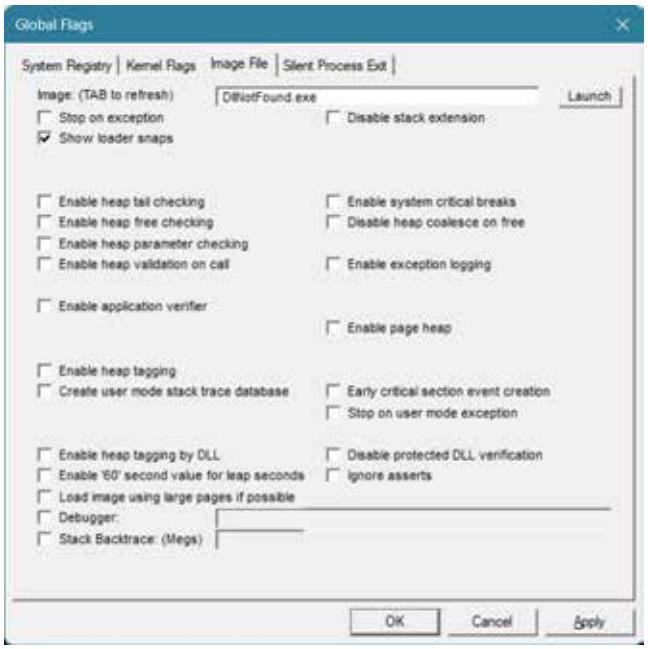


Figure 7

Viewing Loader Snaps

Now whenever a program named `DllNotFound.exe` is executed under a debugger such as Visual Studio or WinDbg all the loader diagnostic information will appear in the debugger's output window. For example, Figure 8 is the result of running `DllNotFound.exe` using WinDbg.

This gives us the information we had in the dialog box we obtained by default at the beginning of this article, and also additional debugging output that may help us with diagnosing more complicated issues. You can, for example, see in this screenshot the tail end of the complete list of places the loader searched when trying to locate `MissingDll.dll`.

The same is true for the `MissingSymbol.exe` case: the output in the debugger contains:

```
4a58:3e34 @ 836003546 - LdrpNameToOrdinal
- WARNING: Procedure "?expected_function@@
YAXXZ" could not be located in DLL at base
0x00007FFDC3310000.
4a58:3e34 @ 836003546 - LdrpReportError - ERROR:
Locating export "?expected_function@@YAXXZ" for
DLL "c:\Projects\articles\2025-04-show-loader-
snaps\bin\MissingSymbol.exe" failed with status:
0xc0000139.
```

However the output in this case is slightly less immediately readable as there are fifty or so *additional* **`INFO`** lines logged after these two, making it a little more difficult to identify the relevant output.

The 'Show Loader Snaps' approach *also* provides useful diagnostic information when using the **`LoadLibrary`** or **`GetProcAddress`** API. For example:

An error 126 is reported from **`LoadLibrary`** as the Dll is not found:

```
390c:2b7c @ 841275531 - LdrpProcessWork - ERROR:
Unable to load DLL: "MissingDll.DLL", Parent
Module: "(null)", Status: 0xc0000135
```

An error 1114 is reported from **`LoadLibrary`** when it loads a Dll that crashes during initialization:

```
316c:3540 @ 841347500 - LdrpInitializeNode -
INFO: Calling init routine 00007FFDC331100A for
DLL "c:\local\bin\CrashingDll.DLL"
```

An error 127 is reported from **`GetProcAddress`** for a symbol that is not found in the target DLL:

```
26c0:311c @ 841153484 - LdrpNameToOrdinal -
WARNING: Procedure "expected_function" could not
be located in DLL at base 0x00007FFE004D0000.
```

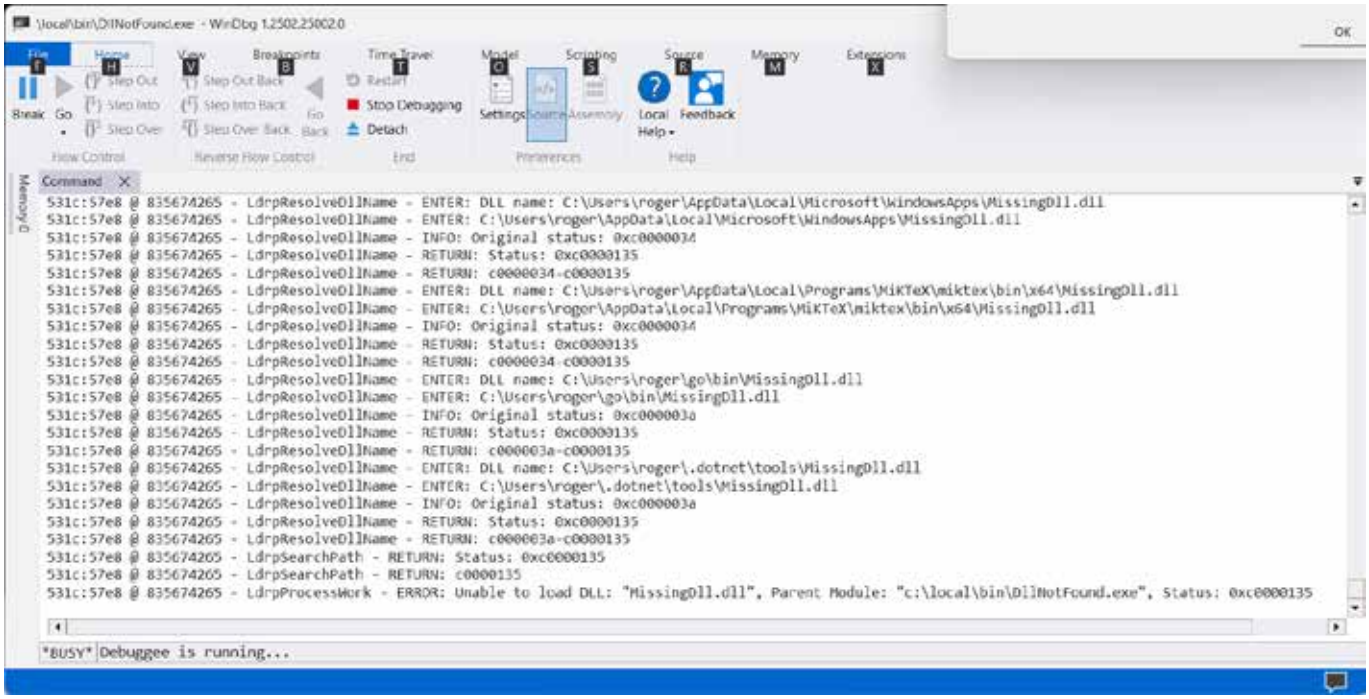


Figure 8

```
26c0:311c @ 841153484 - LdrpReportError -
WARNING: Locating export "expected_function" for
DLL "Unknown" failed with status: 0xc0000139.
```

Let's write a tool

The loader snap output goes to *any* debugger so let us write one that is designed specifically for this task.

We can make use of the debugger logic from 'Using the Windows Debugging API' published in *CVu* March 2011 and also available on GitHub [Orr-1].

The two basic parts to writing a simple Windows debugger are:

- Passing the flag **`DEBUG_PROCESS`** to the call to **`CreateProcess`**
- Repeatedly calling the pair of functions **`WaitForDebugEvent`** and **`ContinueDebugEvent`** to obtain and handle successive debug events from the target process.

For the purposes of this debugger, the only event we are interested in is **`OUTPUT_DEBUG_STRING_EVENT`** that contains the loader snap output; we don't need to handle any of the other event notifications here.

I've wrapped the basic debugger loop inside a helper class, **`DebugAdapter`**, and the user of this class simply overrides the methods of interest. In this case the only method we are interested in overriding is **`OnOutputDebugString`** (see Listing 3).

```
void ShowLoaderSnaps::OnOutputDebugString(
    DWORD /*processId*/, DWORD /*threadId*/,
    HANDLE hProcess, OUTPUT_DEBUG_STRING_INFO
    const &DebugString) {
  const auto message =
    ReadString(hProcess,
               DebugString.lpDebugStringData,
               DebugString.fUnicode,
               DebugString.nDebugStringLength);
  // Filter out unwanted messages
  for (const auto &filter : filters_) {
    if (message.find(filter) !=
      std::string::npos)
    {
      return;
    }
  }
  os_ << message << std::flush;
}
```

Listing 3

For the purposes of this article, we provide a simple list of filters as member data to reduce the number of messages we are not interested in. Of course, this logic could easily be expanded further to provide more targetted information for specific use cases.

At first start, when we enable 'quiet' mode, we should filter out messages containing **`ENTER:`**, **`RETURN:`**, and **`INFO:`**. This usually leaves us with warnings and errors, which for most DLL failures is often enough to solve the issue.

For example, Listing 4 is the complete output when running this program targetting `MissingSymbol.exe` (with 'Show Loader Snaps' enabled):

The filtering enabled by using the **`-q`** option has removed the extra 'noise', allowing us to see just the warnings and errors. If this is not quite enough to enable us to diagnose the root cause of the problem, we can, of course, re-run with full output to see the additional informational messages.

The full source code for `ShowLoaderSnaps` is available on GitHub [Orr-2].

```
C:> ShowLoaderSnaps -q c:\local\bin\
MissingSymbol.exe
1558:3b34 @ 842130000 - LdrpNameToOrdinal
- WARNING: Procedure "?expected_function@@
YAXXZ" could not be located in DLL at base
0x00007FFDD8260000.
1558:3b34 @ 842130000 - LdrpReportError - ERROR:
Locating export "?expected_function@@YAXXZ" for
DLL "c:\local\bin\MissingSymbol.exe" failed with
status: 0xc0000139.
1558:3b34 @ 842141015 - LdrpGenericExceptionFilter
- ERROR: Function LdrpSnapModule raised exception
0xc0000139
        Exception record: .exr 00000032FB9FEBF0
        Context record: .cxr 00000032FB9FE700
1558:3c98 @ 842141015 - LdrpInitializeProcess
- ERROR: Walking the import tables of the
executable and its static imports failed with
status 0xc0000139
1558:3c98 @ 842141031 - _LdrpInitialize - ERROR:
Process initialization failed with status
0xc0000139
1558:3c98 @ 842141031 - LdrpInitializationFailure
- ERROR: Process initialization failed with
status 0xc0000139
```

Listing 4

Removing the need for Admin rights

The solution so far has two problems, the worst of which is that writing to the **HKEY_LOCAL_MACHINE** area of the registry requires Admin rights. In quite a few of the places where I have worked, it is a challenge for developers to get Admin rights because of the obvious security issues that this causes. We ideally want a *non-admin* way to set the show loader snaps flag in the target process that doesn't require writing to the system part of the registry.

Using the Loader Config

One of the lesser-known parts of the PE header is the **LoadConfig** directory item (internally identified by the index **IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG**). The data structure this points to, **IMAGE_LOAD_CONFIG_DIRECTORY**, contains a field **GlobalFlagsSet** and values in this field are OR'd into the existing GlobalFlags settings for the process when this entity is processed by the system loader. You can examine the settings using **dumpbin /LOADCONFIG**.

If we can set the appropriate option in this header then our program will show loader snaps, and writing to the executable program file does not require admin rights *per se*.

While you *can* provide the complete data structure yourself at link time, replacing the default one placed in the binary by a combination of the linker and the MSVC runtime support library, this is quite hard to get right as some of the fields in the structure contain values necessary to support other features, such as structured exception handling, that your program probably also needs.

A simpler solution is to write a program that sets the correct value into the **GlobalFlagsSet** field of an already linked executable file. We know the value to set is 2 from what `GFlags.exe` writes into the registry – see earlier.

The ImageHlp header contains functions to help us do this; we can open the binary file using **MapAndLoad** and edit the data.

First (Listing 5), I'm using a simple helper class to provide a very simple RAII wrapper to the underlying C style API.

We can then set the flag appropriately for 32-bit programs (see Listing 6) using two helper functions:

- **GetImageConfigInformation**
- **SetImageConfigInformation**

While in theory the same code should work in 64-bit mode… it doesn't. There appears to be a problem in v64-bit mode with **GetImageConfigInformation**. I have raised a ticket with Microsoft to see if they could fix this issue [Orr-3].

However, we can still do the same thing ourselves, by manually walking the data structures.

- Starting with the **FileHeader** in the loaded image we cast it to the correct 32-bit or 64-bit **IMAGE_NT_HEADERS** structure.
- From the **IMAGE_NT_HEADERS** we read the relative address of the load config using:

  OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG]. VirtualAddress
- We convert this relative address in the header to a virtual address in our own address space using **ImageRvaToVa**
- Then we simply OR in the correct value:

  pLoadConfig->GlobalFlagsSet |= SHOW_LOADER_SNAPS

(The complete code is available in `SetLoaderSnaps.cpp`.)

```
struct LoadedImage : public LOADED_IMAGE {
  LoadedImage(const std::string& filename) {
    if (!MapAndLoad(filename.c_str(), nullptr,
        this, false, false)) {
      throw std::runtime_error("MapAndLoad(" +
        filename + ") failed: " +
        std::to_string(GetLastError()));
    }
  }
  ~LoadedImage() {
    if (!UnMapAndLoad(this)) {
      std::cerr << "UnMapAndLoad failed: "
                << GetLastError() << '\n';
    }
  }
};
```

Listing 5

Can we do all this at runtime?

The second problem with the approach taken so far is that it makes *persistent* changes to either the registry (with `GFlags.exe`) or the executable file (with `SetLoaderSnaps.exe`.)

We usually only want to set the loader snaps flag temporarily while we are investigating a problem; in the normal case where it 'all just works', we don't want to have any additional overhead (when there is no debugger attached) or extraneous debug output (when a debugger is attached.)

We can resolve this easily, subject to using a couple of non, or partially, documented features, inside our **ShowLoaderSnaps** program itself.

Firstly we need to use an undocumented field, **NtGlobalFlag**. The Show Loader Snaps flag ends up in the process' memory in the **NtGlobalFlag** structure which is in turn inside the **PEB** (Process Environment Block). While the Windows SDK does include *a* definition for the **PEB** structure in `winternl.h`, it is a simplified one with only a subset of the data available. See the official documentation at [Microsoft-8] and see that there are a dozen sections of the structure covered by various **Reserved** fields. We can use the PDB symbols for `NtDll.dll` (available from the Microsoft Symbol Servers) to get the offset in the process environment

```
static const int SHOW_LOADER_SNAPS = 2;

void UpdateImageConfigInformation(const
    std::string &filename) {
  LoadedImage loadedImage{filename};
  std::cout << "Mapped: "
            << loadedImage.ModuleName << '\n';

  IMAGE_LOAD_CONFIG_DIRECTORY imageConfig =
    {sizeof(imageConfig)};
  if (!GetImageConfigInformation(&loadedImage,
      &imageConfig)) {
    throw std::runtime_error(
      "GetImageConfigInformation("
      + std::to_string(sizeof(imageConfig))
      + ") failed: "
      + std::to_string(GetLastError()));
  }
  if (imageConfig.GlobalFlagsSet
      & SHOW_LOADER_SNAPS) {
    std::cout << "Show Loader Snaps flag "
      "already set in image\n";
  } else {
    imageConfig.GlobalFlagsSet |=
      SHOW_LOADER_SNAPS;
    if (!SetImageConfigInformation(&loadedImage,
        &imageConfig)) {
      throw std::runtime_error(
        "SetImageConfigInformation failed: "
        + std::to_string(GetLastError()));
    }
    std::cout << "Set Show Loader Snaps flag\n";
  }
}
```

Listing 6

```
void SetShowLoaderSnaps(HANDLE hProcess) {
  PROCESS_BASIC_INFORMATION pbi = {};
  if (0 == NtQueryInformationProcess(hProcess,
      ProcessBasicInformation, &pbi, sizeof(pbi),
      0)) {
#ifdef _WIN64
  // GlobalFlag is not officially documented
  // Offsets obtained from PDB file for ntdll.dll
  PVOID pGlobalFlag =
    ((char *)pbi.PebBaseAddress) + 188;
#else
  PVOID pGlobalFlag =
    ((char *)pbi.PebBaseAddress) + 104;
#endif // _WIN64
  ULONG GlobalFlag{0};
  const ULONG SHOW_LDR_SNAPS = 2;
  ReadProcessMemory(hProcess, pGlobalFlag,
    &GlobalFlag, sizeof(GlobalFlag), 0);
  GlobalFlag |= SHOW_LDR_SNAPS;
  WriteProcessMemory(hProcess, pGlobalFlag,
    &GlobalFlag, sizeof(GlobalFlag), 0);
  }
}
```

Listing 7

block of the global flag, which lies inside one of these reserved sections. For example, inside WinDbg:

```
0:000> dt ntdll!_PEB NtGlobalFlag
   +0x0bc NtGlobalFlag : Uint4B
```

(and correspondingly the 32-bit offset of 0x068 is obtained from the same command with a 32-bit target)

Secondly we have to get the PEB address in the process being debugged. The address of the PEB can be obtained using the `NtQueryInformationProcess` API.

However, note the cautionary message in the official documentation for this item:

> `NtQueryInformationProcess` may be altered or unavailable in future versions of Windows. Applications should use the alternate functions listed in this topic.

Finally, once armed with the address of the PEB in the target process and the offset of the `NtGlobalFlag` we can easily read/modify/write the value to set the loader snaps flag (Listing 7).

Now we have achieved the ability to write out the loader snap information on demand, without requiring administrator rights nor making persistent changes to either the registry or the binary file.

The `ShowLoaderSnaps` source code contains this additional piece of functionality. Of course, if one of the previous methods has been used the value in `NtGlobalFlag` will *already* contain a '2' and so we will simply re-write the same value back into the `NtGlobalFlag` field, which is benign.

What about Linux?

As mentioned above, Linux also has shared libraries but it uses a different design; executables and shared libraries contain two *unrelated* sets of data, one listing the shared libraries that are needed and the other listing the unresolved symbols that need resolving.

The failure to load a shared library is similar to the Windows case, except that the way the search path is supplied is different: Linux uses the `$LD_LIBRARY_PATH` and any `RPATH` or `RUNPATH` settings embedded in each executable. Additionally, the complete path to the dependent shared library can be embedded in the binary, which can obviate the need for a path search.

The failure to locate a *symbol* in a dependent library is harder to resolve than it is on Windows since there is no indication at all of *which* shared library was expected to provide the missing symbol.

Like the loader snaps, Linux provides ways to produce debug output from the system loader. The environment variable `LD_DEBUG` can be used to enable various categories of additional debugging output from the loader and `LD_DEBUG_OUTPUT` used to control where this output is written. See [man7] for more details.

Conclusion

It can be quite hard to diagnose problems with loading DLLs and the well-known standard debugging tools used for routine debugging tasks do not provide as much help as we might wish. I hope that some of the techniques shown here will help to reduce the pain of diagnosing and fixing such problems! ■

References

[DepWalker] 'Dependency Walker': https://www.dependencywalker.com/

[Github] 'Dependencies': https://github.com/lucasg/Dependencies

[man7] 'ld.so(8)': https://man7.org/linux/man-pages/man8/ld.so.8.html

[Microsoft-1] 'Dynamic Link Libraries': https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries

[Microsoft-2] 'Linker support for delay loaded DLLs': https://learn.microsoft.com/en-us/cpp/build/reference/linker-support-for-delay-loaded-dlls?view=msvc-170

[Microsoft-3] 'Decorated names': https://learn.microsoft.com/en-us/cpp/build/reference/decorated-names?view=msvc-170

[Microsoft-4] 'Dynamic-link library search order': https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order

[Microsoft-5] 'File System Redirector': https://learn.microsoft.com/en-us/windows/win32/winprog64/file-system-redirector

[Microsoft-6] 'Windows API Sets': https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-apisets

[Microsoft-7] 'Debugging Tools For Windows': https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

[Microsoft-8] 'PEB structure': https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb

[Orr-1] 'Simple Debugger', available at: https://github.com/rogerorr/articles/tree/main/Simple_Debugger

[Orr-2] Source code: https://github.com/AccuPublications/overload-listings/tree/feature/DebuggingWindowsDllProblems

[Orr-3] Developer Support Ticket: https://developercommunity.visualstudio.com/t/GetImageConfigInformation-fails-on-x64-w/10890839

Codurance AI Hackathon

This hackathon explored AI-powered software development. Isaac Oldwood shares what he learned from the event.

On Saturday 26th April, I attended an invite-only AI Hackathon at the Codurance headquarters in London. I give a run down of the motivations behind the event, what happened and what we learned. At the end, I discuss some limitations of the findings as well as further questions to be considered.

Codurance

Codurance is a global software consultancy that helps businesses build a better sustainable technical capability to support growth. The software craftsmanship ethos shaped the company. The goal of Software Craftsmanship is clear: raise the bar in the software industry through professionalism and technical excellence.

I do not work with or for Codurance and have no official affiliation with the company. I was on the invite list as I co-organise the Software Crafters Cambridge monthly tech meetup with their Head of Marketing, Natalie Gray.

At one of our planning sessions, Natalie mentioned the event was happening and asked if it was something I would be interested in. I was keen to attend as there is currently so much hype around LLMs and AI integrated tools. I wanted to see if we could cut through all the hype and noise and really learn some valuable real-world lessons.

Glossary

AI Tools	Any tools powered by AI that developers can use in their job, eg Cursor, ChatGPT and many more.
ChatGPT	A LLM created by OpenAI, widely regarded as the first 'mainstream AI'.
Co-pilot	An AI/LLM tool that is integrated directly into VSCode editor.
Cursor	A new code editor with AI built-in.
GitHub	An online website to store code.
LLM	Large language models are a type of artificial intelligence (AI) program that can recognize and generate text (and code).
VSCode	A code editor.

I use AI and LLM interchangeably throughout this piece as AI used in the context of the hackathon exclusively refers to varying integrations of LLMs.

The aim

The event was advertised as:

> AI is transforming software development, but how effective is AI-powered coding in real-world scenarios? Join Codurance's [...] AI Hackathon to put AI-assisted development to the test!

Online, I see lots of examples of people building web apps 'entirely' using AI, but on closer inspection these projects are not generally up to standard. They are usually not well structured, tested or maintainable. The aim of the hackathon was to see how good some of the AI tools available are at writing real-world production-standard code that developers would be proud of.

Timetable	
09:30	Arrival and registration
10:30	Kick-off and Challenge 1
12:45	Lunch
13:45	Challenge 2
16:00	Playback and discussion
16:45	Event close
17:00	Pub

Format

I arrived at the Codurance office a bit early due to train times (about 9:15am). Once I was buzzed in, I was met by Matt Belcher and Rowan Lea. Matt is 'Head of Emerging Technology' and Rowan is a 'Software Craftsperson', both working at Codurance. I was warmly welcomed and given a brief tour of the office as I was the first to arrive.

Over the next 45 minutes, other developers filtered in. It was great to meet everyone! There was a wide range of experience in both the software development industry and using AI tools – I think this played into the whole day very well. Some people were veterans of the industry with 30+ years experience but only had briefly used ChatGPT. At the other end, there was a developer who was still quite early in their career but has been following and using AI tools extensively since they first arrived on the scene. This was great as it meant that everyone had something to contribute but also something to learn and improve on!

After some chatting and fuelling (coffee drinking), Matt and Rowan invited everyone into the space we would be working in. They then explained that everyone was to be split into group A and group B. Group A would use AI tools for the first exercise whilst group B would use traditional non-AI methods. In the afternoon this would be swapped around so everyone gets a go! I was assigned to group A so I got to dive right into the AI tools. It was explained that we should get into pairs or threes within our group to tackle the two exercises.

The brief for both exercises can be found on Matt Belcher's GitHub. You can find each group's output in the forks of each repository.

Exercise 1

Exercise 1 was revealed as 'StyleDen' and asked you to 'build a minimal viable product (MVP) for their e-commerce website'.

For the first exercise, I paired with a C# developer who had been exploring and learning Python. As I was most comfortable with Python, we decided to work together and knowledge share along the way. Since we were assigned to the AI first group, we had a discussion about the best way to use it, and more importantly the best way to put it through its paces. We decided to try and use it to its full potential and avoid writing a single line of code if possible, i.e. just prompting and guiding it.

Isaac Oldwood is a Software Engineer working in the Insurance Industry. He taught himself Python at university to (unsuccessfully) purchase a pair of limited edition shoes. He organises Software Crafters Cambridge, a monthly tech meetup. In his spare time, he enjoys reading, rugby and running. He can be contacted at Isaac.Oldwood@gmail.com.

At the beginning of the task, all we had was a README which contained all the requirements. The first thing we needed was a plan. As previously mentioned, we wanted to fully utilise AI so we passed the entire README to ChatGPT and asked it to produce a solution to complete the exercise.

The first section it produced was titled 'Overview' and it was essentially the parts of the app we would need and suggested technologies for them. It mentioned a frontend built in React, a backend built using Python's FastAPI and a SQLite Database.

It then laid out a file directory structure to help us visualise how to split out the app. It listed some key API endpoints which we reviewed to make sure all the requirements were met. It was good to see these aligned with how we would have designed them ourselves.

One part of ChatGPT's output that I was really interested in was a section titled 'Tech Stack (Quick Justification)'. This section outlined WHY it chose to use the technologies described above. For me this is a really key aspect of using AI. In most of the uses I see of AI, we ask it to complete some task or ask it a question; we very rarely ask the AI to explain (this is a key point I raise later in the day).

The last part it produced was a 'Plan of Attack (MVP Steps)'. This was really useful as it gave us smaller bite size chunks to iterate on as we created our MVP. My only issue with the plan of attack was 'Write some unit tests (especially backend)' was at the bottom of the list. This highlights an issue I have seen repeatedly with AI- (and human-) developed code. Testing is not considered; or if it is, only as an afterthought. As an advocate of Test Driven Development (TDD), this is a real issue for me. I want tests to be written first based on the requirements, then code to be written to pass those tests. Just to reiterate, this is for production code as was the aim of the day. I understand that usually for a 'Hackathon', you are building some form of prototype and it may not be the time or place for TDD.

As we wanted to fully embrace AI, we concluded to use the technologies suggested by ChatGPT. This was partially a decision due to us having some experience with the technologies, but also because these are technologies that are widely used. This means in theory the LLMs will have plenty of training data and should produce decent code. That was the theory at least…

To actually start writing the code I used GitHub Co-pilot built into VSCode – with this you can use 'agent' mode. This allows you to prompt an 'AI Agent' which will then make edits directly in your files. We started at the first step of the ChatGPT plan of attack and asked it to create a SQLite database along with a seed script (to load the CSV into the database). This worked first time and created a file that worked successfully without any tweaks. However, it did not create any tests.To rectify this we discarded the changes and added 'Using TDD…' at the start of the prompt. The second attempt created a very similar script whilst also writing some tests.

As a side note, now reflecting on the day, it has been pointed out to me that it is possible that this isn't really proper TDD. An LLM writing code and tests in one loop/prompt does not force the tests to be written first and then code to be written to satisfy those tests. It is certainly possible that the production code is written first and then the tests are written. It is not clear to the prompter. Perhaps a better process would be using the LLM to write the unit tests first in one prompt, verifying the tests, and then using another prompt to write the production code to satisfy those tests.

The second step was creating a boilerplate FastAPI app. I used some prompts such as 'Create a boilerplate FastAPI app using TDD', this created a very basic app as well as using the FastAPI framework.

Another thing that we explored is documentation writing. If we were writing real production code, this app would need to be worked on by other developers that may not have experience with writing/running these APIs. So after getting some working code we asked Co-pilot to 'Add local setup and running steps to the README'. The documentation produced was easy to follow and contained all the necessary steps to get the app up and running locally.

The rest of the first session followed in this flow. After a basic API was created we moved onto the frontend. Neither myself or my partner have extensive experience with React (though I am trying to learn a bit more). The first thing Co-pilot did was ask to run `create-react-app`. I was surprised that it was capable of using the terminal directly.To clarify, it does ask your permission before running every command with a simple 'Continue' button. I do worry that people may just click 'Continue' without fully understanding the commands being run, which could become a security concern.

My part of the exercise was to create the cart page. I prompted Co-pilot to create a new cart page with tests. I asked it to add some basic functionality; for example, allow the user to increase/decrease the item count in the cart. As well as, if the item count reached zero then remove it from the cart. After some manual testing of the app, I discovered that once I removed the last item from the cart the table still showed but just empty. This was bad UX in my opinion. I was happily surprised with how easy this was to improve by prompting Co-pilot 'Currently when no items are left in the cart nothing happens, update this code and tests to display a message such as "No items in cart"' It updated the code and tests in a straightforward way and in very little time.

By this point, we were running out of time. I wanted to add a couple of finishing touches and asked the AI to add some images and a dynamic total at the bottom of the table. You can see the code's final state on my GitHub along with all the local running instructions. All the code and documentation has been entirely written by AI tools. My partner and I edited no code manually. To summarise, I was very impressed with how quickly we got a working app up and running with very little intervention from us humans.

Lunch

Lunch was provided by Codurance and gave us all some well-earned time to reflect. Of course, Exercise 1 dominated the topic of discussion. There was lots of chatting between pairs within group A about what tools were used, what prompts worked well and other tips and tricks. There were also lots of discussions between group A and B about varying aspects of the task. The key takeaways were:

- Group A got further in the exercise (a more complete solution with more features) than Group B

  Clearly due to using AI tools, it allowed them to work faster

- Co-pilot and ChatGPT were widely chosen AI tools

  It seemed like this is due to familiarity and being built into VSCode, most of the developers' editor of choice

- The AIs did not write unit tests unless specifically asked, but when prompted it did write them mostly to an acceptable standard

Exercise 2

The second exercise was revealed as 'StreamStack'. Essentially, build a movie reviewing website. For this task we decided to mix up the pairs, which allowed new ideas and networking. I ended up forming a three with two other developers who were happy to use Python. We knew we would have no AI help for this exercise so we needed to stick to tools and technology we had experience with.

We started off by working out that we would need a backend and frontend. We wrote down some questions and design decisions on post-it notes and created a rough architecture/design diagram. One of the team had experience with React and so offered to handle the frontend part. This left me and the other team member to create the backend.

As the functionality was on the simpler side, I suggested using FastAPI. It is my preferred technology for creating APIs as it is simple, integrates with Pydantic for validation and has a great testing framework. My backend partner had not used FastAPI before and preferred Flask, it didn't take me long to persuade them to give it a try!

We continued much as you'd expect at a hackathon: we used TDD to put together the backend API and start integrating it with the UI. It was noticeably slower this time round compared to using the AI tools (especially without the auto-complete/in-line suggestions). Although this time round I personally felt I understood every line of code and was happy that it would pass a code review. I also spent next to no time at all reviewing the code as we actually wrote it ourselves.

An example of being slower was right at the start. We needed to create the FastAPI app, first of all just with a "Hello world" endpoint to make sure we had set it up right. Previously, I would have asked Co-pilot or ChatGPT to write a very brief boilerplate file for a FastAPI app. This time we had to google the FastAPI docs, navigate to the quick start guide and copy the code from there. As I had used this many times before I knew where to look, which sped things up somewhat. However, this process would have certainly been faster with the use of an AI tool.

By the end of the exercise we had a slightly crude web app with a UI and a backend. It had some basic filtering and sorting functionality but we did not have time to complete all of the requested features in the given time. It did have a full test suite though!

End of day discussion

This was the part of the day that was the most insightful to me. A pair from group B kicked off the 'show and tell' by showing their 'StreamStack' app. They had used Cursor and it was immediately very impressive. They had a complete application that had every functionality asked for, looked nice and they even had time to add bonus things like images. One of the members of the pair said something that really stuck with me, though. They explained that the application was practically a black box as they had only given it a few prompts and just asked it to create the application. After the AI had finished, they had tried to use images on a different page and were unable to get it working; this should have been trivial. They said, "This application was written two hours ago and I already feel like I'm working with legacy code." They believed that if they had written it all then adding these images would be trivial but because it was a black box they would take much longer to understand and make these changes.

I feel like my first pair had a similar problem with the AI code being a bit of a black box. This prompted me to ask the question, "There has been lots of talk about black box code and not easily understanding the AI changes – did anyone ask the AI to explain the code?". There was a long pause as it was clear no one had done this, including myself! It seems all groups had spent the day asking AI to write/change code and not once asked it to explain code. This is a feature that has been advertised, particularly with Co-pilot's chat feature. I have used this a few times at work when moving into a new project. I think that was a large unexplored part and a use that we should have tested more during the hackathon.

Another group spoke about abstraction and refactoring. They said that the AI tools heavily favour 'copying and pasting' similar code instead of extracting and refactoring into its own function for reusing elsewhere. They had similar functionality in three places in their app and the AI re-created the logic every time. If they wanted to tweak it they would have to change it in multiple places. It seems AI does not follow DRY. They did explain that with some guidance and prompting the AI tools could refactor and extract logic, but it wasn't natural and had to be requested specifically.

A pair of developers followed on from this point. They asked the AI tools to refactor some code in a specific file; it did manage this but along the way would update and change unrelated code in other files. Another person raised their hand and agreed with this point. They vented some frustration with this in their day job. They told us the following anecdote; they were working on a large codebase with many files and wanted to update/refactor a specific file. By default, Co-pilot will take your whole workspace as 'context' to make these changes. Unfortunately, that also means it can access and make changes to every file in your workspace. They suggested a good improvement to the tool would be to tell the AI to 'read' these files for context but only allow 'write' changes in file X,Y and Z.

Lastly, a member of my three for Exercise 2 said, "I have achieved a lot less in this problem compared with using AI tools; however, I can say for sure, I am more proud of the code I have written." I think this is a key point because, as developers, all code we commit has our name on it. We should be proud of the code we write. This perpetuates ownership and in my opinion results in better code being written.

Post-event

After the event we headed to the pub. There was still a bit of chatting about AI but we mostly were all done with discussing AI for the day. It was nice to chat about other non-AI stuff over a beer. We all agreed we would love to attend a similar event in the future!

Limitations

If we were to do this again there are some things I would like to test. I think we gave the AI tools the best possible chance by picking problems that are widely solved with lots of examples on the internet. Having said that, there are some questions raised:

- How well does it perform when writing code for something other than a web app e.g. embedded systems?
- How well does it perform in a different problem domain?

  What about in a domain where there is lots of context required that may not be widely documented in the training data?
- How well does it perform in an existing code base?

  Both of these exercises were building something new. How well does it work when asked to change/write new code in an existing project?
- Would developers with more AI experience do better?

  Some of the developers had little experience with AI tools. Are there ways of working that unlock better output? Had we known these, would we have done better?
- How good and useful is asking AI to summarise/explain code?
- Most of these tools allow you to change the LLM being used. Would different LLM choices have produced better results?
- As previously mentioned, does adding "Use TDD…" to the prompt actually use TDD within one prompt or does it require a two step process?
- How safe is allowing the LLMs to directly run commands in the terminal?

  Is a 'Continue' button enough to prompt the user to verify the code vs copying and pasting commands from the internet?

TLDR

My key takeaways are:

- The AI tools are great for writing boilerplate/setup code.
- AI tools avoid DRY.
- The AI tools did not write unit tests unless specifically asked, but when prompted it did write them to an acceptable standard.
- The AI tools did better when asked to work in smaller steps.
- Developers are more proud of their work when using less AI.
- Some tools are better than others, with the tools that can edit directly in the IDE saving more time.
- The 'auto-complete'/in-line functionality is the way most developers use the AI tools.

Ultimately, it is clear to me: developers can already move faster and be more productive with AI tools and these effects are only increasing. ■

This article was first published on 2 May 2025 on Isaac's blog: https://isaacoldwood.com/blog#codurance-ai-hackathon

Tracking Success

Developing adaptive eye-tracking tools for children with cerebral visual impairment has specific challenges. Jacob Farrow describes his progress so far.

Earlier this year, I presented a poster at ACCU 2025 [ACCU] titled 'Tracking Success: Enhancing Visual Tracking Skills in Children with Cerebral Visual Impairment (CVI) through Interactive Digital Tools'. The project explores whether gaze-tracking technology can be meaningfully adapted for children with CVI – an often-overlooked neurological condition that affects the brain's ability to process visual information.

I was thrilled to see people stop by and engage with the poster. Some had experience with assistive tech, others wanted to know how eye-tracking can be made more inclusive. We discussed head pose, side-eyeing, glare from glasses, and real-time feedback loops. It was an encouraging reminder that sometimes niche research can strike a chord with a wide audience.

Problem statement

CVI is now the leading cause of visual impairment in children in the UK. Unlike traditional eye problems, it affects how the brain interprets visual input – even if the eyes themselves are healthy. CVI manifests in many (often contradicting) ways. Children with CVI may use peripheral vision instead of central, avoid eye contact, or struggle to recognise moving/static objects. This makes it difficult for traditional educational tools – and standard eye-tracking systems – to interpret what these children are seeing or focusing on.

So, full of the hubris of an engineering student, I created my final-year project and set out to change that. I wanted to build an eye-tracking system that could cope with diverse gaze behaviours, and provide real-time feedback to help practitioners understand how children with CVI engage with visual stimuli.

The research had both academic and real-world legs. Academically, it formed the core of my Software Engineering degree project at The University of Bradford [Bradford]. Professionally, I developed it as Lead Software Engineer at SpaceKraft Ltd [SpaceKraft] – a company that researches and develops sensory solutions for children with disabilities. I saw a chance to make a practical tool that could be deployed in classrooms and therapy spaces, not just written about in reports.

What I built

The core of the system is an interactive game that asks users to follow moving objects across a screen. A camera tracks the user's eye movements, estimating gaze position in real-time. The game then uses this data to adjust the size and speed of the object based on user accuracy in order to give performance feedback.

Jacob Farrow is the Lead Software Engineer at SpaceKraft Ltd and a final-year Software Engineering student at the University of Bradford. He leads the development of sensory solutions used in special education around the globe, specializing in computer vision and real-time interaction. An Engineering Leaders Scholar with RAENG, he contributes to inclusive design frameworks and mentors young engineers. Contact him at Jacob-M-Farrow@outlook.com

But building a working prototype meant figuratively wrestling with a long list of edge cases. Many standard eye-tracking libraries assume a clear, frontfacing gaze. Children with CVI often present anything but. They may 'side-eye', tilt their heads, look 'through' objects, or glance briefly before disengaging.

Here's where the system had to adapt:

- **Face & Eye Detection**: I used dlib [dlib] for facial landmark detection, reinforced by CLAHE (Contrast Limited Adaptive Histogram Equalisation) [Wikipedia] to enhance image clarity under varied lighting.
- **Glare Reduction**: Glasses introduced major glare issues, especially with sensory room lighting. I applied inpainting and thresholding to mask bright regions, along with techniques inspired by polarization filtering.
- **Calibration & Gaze Mapping**: I stored pupil and eye corner data, along with head pose matrices, during calibration. This was mapped to screen coordinates using a combination of linear regression and data-driven mapping.
- **Feedback & Logging**: Engagement data (accuracy and session metrics) was logged securely for practitioner review – while respecting strict privacy standards.

The whole system runs on a streamlined Linux build on a 32″ touchscreen with a USB camera, booting directly into the app for plug-and-play simplicity. It's developed in C++ with OpenCV, OpenGL, and Dear ImGui, compiled using Ninja and CMake.

What I learned

Accuracy isn't everything. Most eye-tracking systems measure fixations, saccades, and dwell time to infer engagement. But children with CVI don't necessarily exhibit those behaviours in expected ways. Instead of focusing purely on metrics, my system focuses on responsiveness. If the child interacts – however briefly or obliquely – that counts as meaningful engagement.

Practitioner input is vital. This wasn't a solo coding exercise. I collaborated closely with educators and specialists, who gave continual feedback during development. They helped me understand not just how the system *works*, but how it might actually be *used* in a therapeutic setting.

The main menu UI rendered using Dear ImGUI.

Figure 1

Gaze point being rendered to screen while tracking the rocket [Art].

Figure 2

Adaptive design beats one-size-fits-all. Customisation was key. Children needed different contrast levels, movement speeds, and calibration sensitivities. This led to a settings system that could be tuned per user – an arena I'd like to expand further.

Feedback from ACCU

People at ACCU had great questions – some of which caught me off guard in the best way. One asked whether the system could learn from individual users over time. Another wondered about the potential of integrating into VR environments. A few developers had worked on gaze estimation themselves and were curious about how I approached noisy data, partial occlusion, and hardware constraints.

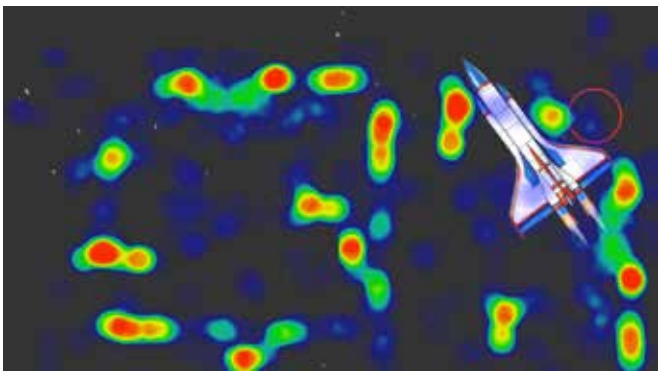
It was validating to hear how many people saw potential of this kind of tech beyond high-end labs or gaming setups. One even said, "I've never seen eye-tracking used for kids before – especially not like this."

Next steps

The current prototype has laid the groundwork, but there's a long way to go. Planned improvements include:

- **Dynamic calibration** that adjusts on-the-fly during gameplay, reducing setup time and improving accuracy without user effort.
- **Multiple game modes**, including shifting gaze tasks and noisy backgrounds to test visual attention more thoroughly.
- **Gaze heatmap visualisation**, offering real-time and session-based insight for practitioners to understand focus zones and avoidances.
- **Deeper analytics**, including attention duration, latency, and object tracking success over time.

The project will be entering a new phase of weekly testing with a cohort of children with CVI at a partner school. The feedback will guide further iteration and help define the long-term viability of the tool in classroom environments.



Session-based heatmaps could offer practitioners valuable insights.

Figure 3

Final thoughts

Software isn't just about solving problems – it's about solving the right problems. This project gave me the opportunity to design something that may help children who are often underserved by mainstream tech. It challenged me technically, but also reminded me why I got into this field in the first place.

The real test will be whether children engage with it, learn from it, and enjoy using it. If they do – even just one of them – then this project has already been a success. ■

Acknowledgements

I wish to express my profound gratitude to John Kopelciw and Chris Morton of SpaceKraft for their invaluable guidance, encouragement, and professional insights throughout the course of this project. Their dedication to creating innovative and impactful solutions has been both inspiring and pivotal to the development of this work.

I am equally indebted to Dr. Rachel Pilling of the University of Bradford, whose expertise and thoughtful advice have been instrumental in ensuring the relevance and effectiveness of this project in addressing the needs of children with Cerebral Visual Impairment (CVI). Her support has been critical in shaping the academic and practical contributions of this research.

Finally, I extend my heartfelt thanks to Dr. Ci Lei of the University of Bradford, who always encouraged me to look at things from a different angle. His perspective has profoundly influenced the innovative aspects of this project and has inspired me to think more critically and creatively.

The completion of this project would not have been possible without their collective expertise and generosity in sharing their time and knowledge, for which I am deeply thankful.

Glossary

CLAHE	Contrast Limited Adaptive Histogram Equalisation – used to enhance image contrast in low-light or uneven lighting conditions.
CVI	Cerebral Visual Impairment – a condition where the brain struggles to process visual information.
Dear ImGUI	An immediate-mode GUI library used for rendering fast, dynamic user interfaces in graphical applications.
Dlib	An open-source machine learning and computer vision library used for facial landmark detection.
Fixation	When the eyes are stationary and focused on a single visual point.
Gaze Heatmap	A visual representation of where the user looked most frequently or for the longest duration.
Inpainting	An image-processing method that fills in missing or obscured parts of an image.
Saccades	Rapid, ballistic eye movements between fixation points.
OpenCV	Open Source Computer Vision Library: A widely-used library for real-time computer vision.
OpenGL	Open Graphics Library: A graphics API used to render interactive elements on the screen in real time.

References

[ACCU] ACCU 2025 Conference: https://accuconference.org/.

[Art] Artist of the Portal Illustrations kasej.portalillustrations@gmail.com.

[Bradford] The University of Bradford: https://www.bradford.ac.uk/external/.

[dlib] dlib C++ Library: https://dlib.net/.

[SpaceKraft] SpaceKraft Ltd: http://www.spacekraft.co.uk.

[Wikipedia] CLAHE: https://en.wikipedia.org/wiki/Adaptive_histogram_equalization.

Afterwood

Human brains are wired for pattern recognition. Chris Oldwood explores some patterns he's seen over the years.

Pattern matching is a concept traditionally associated with functional programming but the more I think about my day-to-day job in the world of software development, the more I realise that pattern matching in general is something which pervades everything from working with the codebase, to processes, and ultimately the people in the organisation.

I was reminded again recently that we don't all see the same patterns in code. (Before going on we need to stop and remind ourselves of Ralph Waldo Emerson's famous quote "A foolish consistency is the hobgoblin of little minds" but this not about being right or wrong, just a reflection on the disparity.)

I discovered that someone had inserted 7DY (a financial period, aka 'tenor', representing 7 days) between 0DY and 0YR instead of between 6DY and 1WK in this sequence below, and that threw me.

0DY, 0YR, 4DY, 5DY, 6DY, 1WK, 30DY, 1MO, …

Now, I should point out that each entry was on a separate line as they were the keys of a dictionary, but they were still on consecutive lines. In the past, even alphabetised lists have not been immune to seemingly random insertions, and they have slipped through the review process, too, because diff tools only show a few lines of context, which incentivises you to largely ignore the wider context unless you go out of your way. (If it really matters, enforce it in code or with a test.)

In essence you could consider the first choice of insertion point a local maximum (between days and years, albeit both zero) whereas the second one might be more like a global maximum (between six days and one week – yes, finance is weird). And that, I think, is one difference that distinguishes programmers along their journey to mastery – they typically go looking for the global maxima rather than settling for the first local maxima they find.

As we start to zoom out of the codebase, we see patterns at different levels – statements, functions, classes, components, systems, etc. For some reason, we refer to small-scale patterns as mere 'idioms', whereas once they get large enough, they get promoted to Design Pattern™ status. (Although ironically the original design patterns made famous by the Gang of Four were relegated to 'idiom' by many commentators.) In the past, I've quipped that many interfaces I see are more *adhesive* than cohesive, as people have a tendency to just stick a new method on the end instead of looking for a 'more logical' place to insert it.

On the subject of terminology, a favoured pattern-oriented approach to software development goes by the moniker 'Convention over Configuration'. The idea is that it should be easier to follow an existing pattern and have the right thing magically happen, than be given free rein and then need to explicitly link the artefacts together. Some conventions, e.g. putting all source files under a `src` folder to avoid you needing to add each filename to a project/makefile/build script, span technologies and helps you fall into the Pit of Success. Other conventions, which typically involve using reflection and adhering to seemingly arbitrary naming rules, are less obvious. In the past, I've discovered tests that weren't run because they only started with 'test' and not 'test_'. (The former style is the convention in more than one popular test framework, but not the one we were using.) Likewise, I've discovered entire test assemblies being missed out of the CI pipeline due to being *un*conventional. Typically, this then begs the question about how someone could write tests and not notice that they weren't being run.

I think this is another area where those of us further along their programming journey begin to explore patterns outside the codebase and architecture – people and their behaviours. For example, I once noticed that a certain member of the team would consistently submit merge requests with a large commit history with tiny changes and typically the word 'fixed' in the message. When reviewing, it's common to only consider the final outcome of the entire changeset and not the journey, so you probably miss the signs buried in the history. In this instance what was missing was taking the time to review their code properly and run the entire test suite before pushing the branch thereby creating an excessive amount of context switching.

A few decades earlier, I remember joining a team to help improve the scalability and reliability of a distributed system. One of my first jobs was to deal with a serious memory leak, which actually turned out to be a considerable number of smaller leaks – forgetting to mark the destructor of a base class as virtual. While I could have just fixed the leaks and moved on, I wondered if there was a pattern there. It turned out they were all written by the same person, and although no longer in the team, they were still at the company and only a few rows away. They were very appreciative of my discovery and for taking the time to talk to them about the mistake. (Sadly, not everyone there was quite so happy with my ability to spot suspect code patterns and use the version control tool to work backwards to identity the author.)

Monitoring systems is another area where our pattern recognition abilities get put to the test as we scour log files and performance graphs, looking for the clues that caused the system to behave in an odd way. The trick here, like elsewhere, is not to fall foul of the old adage about correlation being mistaken for causation. Of course, before you can spot a system that's behaving weirdly, you have to know what it looks like when it's behaving normally, which likely entails spotting a different kind of pattern. This is why I like to write-up postmortems, it's hard to spot a pattern if you don't have the history to draw upon.

Cohesion, whether it be in the code, design, system, processes, or organisation, is an enabler for successful delivery, but it requires us to stand back to see the bigger picture and course-correct when things have gone wayward. Inside the chaos, there may well be order fighting to get out, but only if you go looking for it. ■

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

accu

professionalism in programming

Monthly journals, available printed and online

Discounted rate for the ACCU Conference

Email discussion lists

Technical book reviews

Local groups run by ACCU members

Visit www.ACCU.org to find out more

accu

Professionalism in Programming

Professional development
World-class conference

Printed journals
Email discussion groups

Individual membership
Corporate membership

Visit accu.org for details