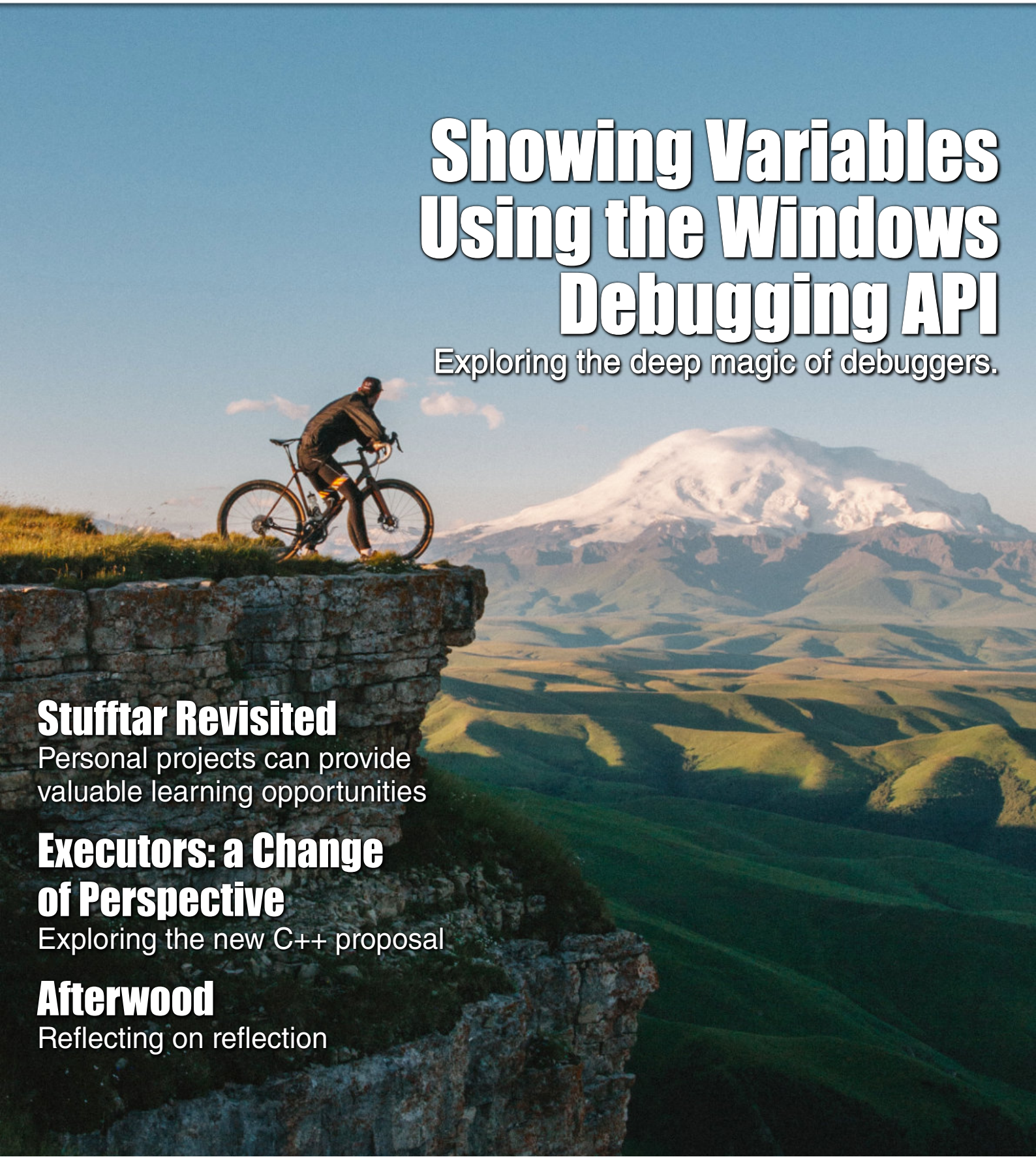
overload 165
OCTOBER 2021 £4.50



Showing Variables Using the Windows Debugging API

Exploring the deep magic of debuggers.

Stufftar Revisited

Personal projects can provide valuable learning opportunities

Executors: a Change of Perspective

Exploring the new C++ proposal

Afterwood

Reflecting on reflection

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 165

October 2021

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Dmitrii Vaccinium on Unsplash.

Copy deadlines

All articles intended for publication in Overload 166 should be submitted by 1st November 2021 and those for Overload 167 by 1st January 2022.

The ACCU

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

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and 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 Showing Variables Using the Windows Debugging API
Roger Orr explores the deep magic of debuggers.

13 Stufftar Revisited
Ian Bruntlett shares a system call surprise he discovered while extending stufftar.

15 Executors: a Change of Perspective
Lucian Radu Teodorescu explains the new C++ proposal.

20 Afterwood
Chris Oldwood reflects on reflection.

Copyrights and Trade Marks

The Right Tool for the Job

Fads and frameworks come and go. Frances Buontempo encourages us to find what works rather than follow fashions.

Having been on gardening leave recently, I have in fact been gardening. As you can imagine, this hasn't left enough time to think of an editorial, so I apologise. I used some loppers to cut down a climbing hydrangea, which is making its way up a wall and over a roof. This seemed like a good idea, but there are a couple of cables going over the roof and into the wall. Fortunately, the one I cut through by mistake is for a television point we don't use. A close call, since the other cable connects us to the internet. A near disaster I am not proud of. With hindsight, using something smaller, like secateurs, would have enabled more precise chopping. Though they wouldn't have got through the thicker branches, I would have been able to see what I was doing better. I possibly also need to go to the opticians, but that's another story.

They say a bad worker blames their tools, but if you use a hammer to put a screw in place, things will go wrong. Perhaps someone blaming a hammer for its failure to screw a nail in place is a sign of a bad worker. The tool used does impact your work, though the effects can be subtle. When I write my excuses for a lack of editorial straight into a word processor, my writing style appears to change slightly. I'm not sure why and find it hard to vocalise how, but I seem to want to type paragraphs in order and end up with slightly tortured links between ideas. If I scribble notes on paper first, I can draw arrows to remind me to move stuff around. Obviously, you can move paragraphs around in a word processor too, but I then forget what I was in the middle of. A line on a bit of paper seems less disruptive. In contrast, trying to make notes on code, outside an editor, means I have piles of paper dotted around, none of which make any sense afterwards. Leaving a TODO comment, or better, something which won't compile works better for me. This doesn't mean there is one true way for any creative process, but there are options and some work better for some people than others. Context is everything.

I have seen a recent trend claiming that agile is the only way to succeed with software projects. This usually specifically means a very rigid scrum process. Though this can work, it can also devolve into what might be termed 'dark scrum'. Ron Jeffries coined this term, saying, "Too often, at least in software, Scrum seems to oppress people. Too often, Scrum does not deliver as rapidly, as reliably, as steadily as it should. As a result, everyone suffers. Most often, the developers suffer more than anyone." [Jeffries16] I have seen estimates for work in story points, which are not supposed to reflect time required to complete, held as promises. If a developer takes longer than a manager expects the number of story points to need, trouble ensues. Stop using story points if you want to estimate how many hours' work something might take [Cohn14]. Scrum is not the only project management approach and is sometimes not the right tool for the job. Kevlin Henney recently republished a blog post [Henney21] about the development process. He suggests that many agile shops are in fact using "waterfall projects rebadged with new terminology, more urgent time scales and increased micromanagement." Micromanagement is almost always mismanagement; however, claiming to be agile because you hold all the right ceremonies is not agile. Furthermore, as Kevlin suggests in his blog, waterfall might be appropriate sometimes. The right process for a job depends on the context.

As opinions on processes change, tooling changes over time too. When did you last use a fax machine? Faxes were the right choice, before scanning and emails became common. They now seem like a slightly pointless historical curio. How many scart cables do you own? Do you still have any 'off the shelf' software on a CD or DVD, but a computer without a cd reader? Or, even more redundant, a floppy disc or two in a drawer somewhere? I have a nagging feeling I have a slide rule somewhere. Unlike the digital tools, a slide rule would still work, though I would need to spend a moment re-learning how to use one. Much of technology becomes obsolete, and some of it really rather quickly. I tend to think it takes a generation or two for tech to fade away, but fax machines prove me wrong. I do wonder if some new tech under active research might turn out to be a short lived fad if it ever becomes a reality, self-driving cars being one thing I have in my sights. I've said it before, but I want a transporter, and a replicator while you're on the case, not a self-driving car. A functional public transport system, more people eating locally grown (or replicated) produce, thereby taking a few lorries off the road, and other ways to reduce volume of traffic can only be a good thing. Since the UK seems to be suffering a lorry driver issue at present, this is now a pressing need. "Tea, Earl Grey. Hot." as Picard says.

To continue on the theme of fads, I shall now turn my attention to AI. For a long while, many new products proclaimed they used machine learning. More recently there seems to have been a move to claiming things are powered by AI. Much of this is an outright lie; however, some systems, such as chat bots, smart speakers and recommender systems are genuinely using elements of AI. Interestingly, anecdotal evidence suggests an increased use of deep learning recently. Specifically, many winners of Kaggle competitions [Kaggle], a website offering prizes for analysis of many disparate datasets, have used deep learning, particularly for 'unstructured' problems – vision, text and sound [KDnuggets]. For those unfamiliar with this tech, deep learning is a type of neural network, with many 'hidden layers of neurons' giving it many more sums to do than a traditional feedforward neural network, which tends to only have one hidden layer. Inputs go to the first layer, magic (maths) happens in the hidden layers, then predictions come out at the final layer. Back to this deep learning fad. Why is this happening? I suspect a spot of TDD – tool driven development – here. There are many free cloud-based frameworks that support deep learning, so it is easy to chuck some data into a network and see what happens without any local setup. You can even invoke the

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. 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.

power of many GPUs without having them locally. This doesn't make it the right tool for the job. I am not disparaging deep learning completely. It does manage some near-magic in disparate domains. I'm not convinced anyone really understands precisely how it works, though. Perhaps whatever works is the right tool for the job.

I gave a talk at an online conference earlier this year, The Machine Learning BASH [BASH], in which I explored the term 'regression'. I was asked about which machine learning frameworks to use. Between ourselves, I panicked at this question. I do tend to zone in on Scikit Learn [scikit-learn], a python framework providing curve fitting, classification and clustering tools, and more besides. However, I have also used JCLEC, a Java framework for evolutionary computation [Ramirez17], and played around with tensor flow, keras, Google's colab and many other frameworks. To be frank, I can't keep up. As I start to get the hang of one thing, a new kid turns up on the block and I feel left behind. I have decided to stick with understanding the maths behind the algorithms and learning how to use the latest tool when I need to, with the full knowledge my ability to drive the tool will become obsolete very quickly. Reading docs on the latest version when required is better than memorising APIs etc which are bound to change, at least to my mind.

I acknowledge some people like to be at the bleeding edge, so make an effort to always upgrade to the latest version or try the newest framework or language. I do feel like I'm missing out when I haven't tried Rust or whichever new language, tool or tech everyone is talking about, but I have learnt to just watch some things from the side lines. There isn't enough time to learn everything, and your brain is a finite resource so I'm sure learning some new things makes you forget other things. Sometimes if you need to analyse data, keep track of something or create software, the best tool for the job is one you know. It's OK to use your favourite IDE, even if people around you are being snobby and trying to make you use Vim or another editor. It's acceptable to draw an architecture diagram with a pencil on a piece of paper, rather than spending an hour trying to work out how to draw a text box on Lucid charts or Visio or the like. It's fine to use a spreadsheet if you want some basic adding up or plots and don't know R or Python very well. I might raise an eyebrow if you tried to do this in C++, but if that's what you want to do and can get results quickly enough that way, do it. Your experience counts for something. Unfortunately, though, even if you are familiar with one toolchain, you find yourself in a position where you cannot use what you know. I recall having to use very old versions of compilers and similar and being stumped when things went wrong. ProTip – don't read the latest docs for gcc or python if you are using a much earlier version. Don't expect things that compile under Microsoft to compile with gcc – though to be fair, the gap is narrowing compared to twenty years ago. Sometimes you have to use what's to hand, even if you suspect there's a better way.

While some businesses are either stuck on old versions of tools and others are on the cutting edge, many will have coding standards, dictating how to do almost anything. This 'One True Way' may be enforced automatically, or via gate-keeping code reviews. It is what it is; however, notice that these guidelines all tend to vary. They are written by people, and everyone has a different history. We've all been burnt by slightly different problems in the past, or been taught one way to arrange braces and whitespaces. Aside from the layout of the code, many guidelines stray into diktats on testing, telling you to always/never use mocks, achieve 100% coverage with end to end tests, ensure the unit tests run quickly. Always use parameterised tests. Never use parameterised tests. And so on. Perhaps the variety of guidelines means we're all still trying to figure out what works. Time will tell.

Here's the thing: many tools, processes and guidelines make sense when you look at the world one way, but if you change your perspective different things come into focus. I've recently been reading a handful of physics books I've found on our bookshelves. The tensions and contradictions between quantum (small subatomic scale) and classical (larger people and planet type scale) models left us searching for a grand unified theory. We do not seem to have found this yet. Classical models and relativity see the world as smooth and predictable. Quantum models have packets or quanta and are probabilistic. Both models make accurate predictions, even though they seem to make conflicting assumptions about the fundamental nature of the universe. The trick is to use the right equations for the scale at which you need results.

It seems there can be such a thing as the right tool for the job, even though opinions can be divided. Maybe the best thing to do is stand firm, ensuring you are on a stable footing. I have been told your stance can make a huge difference in snooker. You might think it's all about maths models, and angles and trig, but it turns out you need to be able to stand firmly and look where you're aiming. Don't get distracted by what's going on around you. Keep your eyes on the balls. Don't be shy about using something you are familiar with if it gets the job done, but be willing to try out new things once in a while.

References

[BASH] Machine Learning BASH: https://www.youtube.com/watch?v=CFwlCCM8ZnI

[Cohn14] Mike Cohn, 'Don't Equate Story Points to Hours', posted 16 Sept 2014 on https://www.mountaingoatsoftware.com/blog/dont-equate-story-points-to-hours

[Henney21] Kevlin Henney, 'Getting over the Waterfall', posted 30 Aug 2021 on https://kevlinhenney.medium.com/getting-over-the-waterfall-c090c6228ca9

[Jeffries16] Ron Jeffries, 'Dark Scrum', posted 8 Sept 2016 on https://ronjeffries.com/articles/016-09ff/defense/

[Kaggle] Competitions: https://www.kaggle.com/competitions

[KDnuggets] 'Lessons from 2 Million Machine Learning Models on Kaggle' at https://www.kdnuggets.com/2015/12/harasymiv-lessons-kaggle-machine-learning.html

[Ramirez17] Aurora Ramirez and Chris Simons (2017) 'Evolutionary Computing Frameworks for Optimisation' in *Overload* 142, published December 2017 and available from https://accu.org/journals/overload/25/142/ramirez_2444

[scikit-learn] skikit-learn: Machine Learning in Python at https://scikit-learn.org/stable/

Showing Variables Using the Windows Debugging API

Debuggers use deep magic to help us out. Roger Orr explores how this magic is performed.

In previous articles [Orr11, Orr12], I demonstrated the basic principles of using the Windows Debugging API to manage a program being debugged and to produce a simple stack trace.

This article looks in more detail about what is needed to access variables in the program using the debugging interface and additionally discusses some of the issues with optimised code that make debugging it a challenge. While the techniques may be useful in their owfn right, they are described principally to try and help explain what interactive debuggers, such as Visual Studio or WinDbg, are doing for us 'under the hood' to achieve some of their functionality.

While the article is written explicitly using the Windows Debug API targeting x64 programs, many of the principles apply to other environments even though the precise details will differ.

Presenting the example code

The code in this article works through varying levels of detail in viewing the local variables in the following, deliberately fairly simple, piece of code. For this example I am using an 'invasive' explicit call to **`stackWalk`**, which I will gradually expand to obtain information about the local variables (Listing 1).

```
void process(Source &source) {
  int local_i = printf("This ");
  int local_j = printf("is ");
  int local_k = printf("a test\n");
  int local_l = source();

  printStack(); // << Here is our 'invasive'
                // function call
  if (local_i != 5 || local_j != 3 ||
      local_k != 7) {
    std::cerr << "Something odd happened\n";
  }
}
int test() {
  Source source;
  int return_value = source();
  process(source);
  return return_value;
}
int main() { return test(); }
```

Listing 1

The **`printStack`** function simply creates a separate thread to perform the actual stack trace, and then joins this thread. This technique allows the program to print its own stack trace; in the previous articles cited in the introduction I used a separate debugging process to control the target process. Both techniques have their uses!

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

We would like to *programmatically* obtain the values of the local variables and the return value of the calling function. Most of us will have done this sort of thing before, but using an interactive debugger.

We will start out compiling the example code without optimisation, and then later on look at the issues that result from turning on optimisation.

The first step in our quest is to walk the call stack. This basic code was described in the earlier articles, and is also relatively well known, so I provide a quick summary of the principles and the sample stack walking code.

Quick summary of stack tracing with the Win32 debugging API

The mechanism used by DbgHelp for Win32 stack tracing revolves around the function **`StackWalk64`**. The programmer sets up the **`stackFrame`** and **`context`** data for the start point on the stack and then calls **`StackWalk64`** in a loop until either it returns **`false`** or the frames of interest have all been processed.

The reason for there being *two* structures involved is that the **`stackFrame`** structure is portable and is passed as a pointer to a **`STACKFRAME64`**, but the **`context`** structure contains environment-specific values – this argument is passed by **`void*`** and it is up to the programmer to provide a pointer to the correct structure for the environment being debugged.

While the basic operation is the same for each platform supported by Win32, there are slight differences. For the purposes of simplifying this article, I am only supporting the x64 platform. In this scenario the Windows headers set up the **`CONTEXT`** typedef to refer to the default, x64, context record and we must pass the **`MachineType`** of **`IMAGE_FILE_MACHINE_AMD64`** as the first argument to **`StackWalk64`**. Other use cases, such as debugging an x86 process, would need to populate the appropriate actual context structure name and set the corresponding value for **`MachineType`**.

The code for walking the stack starting from the 'current location' of the target thread looks like Listing 2 (overleaf).

The **`addressToString`** function is unchanged from the earlier articles cited above and, as its implementation is not relevant to this article, will therefore not be explained further here.

Printing the basic call stack

If we compile the example program with no optimising and with debug symbols ("/Zi") from the command line then, with the **`stackTrace`** function shown above, **`printStack`** produces output like Listing 3 (also overleaf).

(Note: to make the output easier to read I've shortened long paths by replacing the middle of the path with `...`)

While printing a call stack like is extremely useful for debugging problems and getting clearer ideas of the flow of the program, it is possible to enrich the information provided.

If we delete the PDB file and re-run the program, it is easy to see what information from the PDB file is used by the stack trace code

```
void SimpleStackWalker::stackTrace(
    HANDLE hThread, std::ostream &os) {
  CONTEXT context = {0};
  STACKFRAME64 stackFrame = {0};
  context.ContextFlags = CONTEXT_FULL;
  GetThreadContext(hThread, &context);
  stackFrame.AddrPC.Offset = context.Rip;
  stackFrame.AddrPC.Mode = AddrModeFlat;
  stackFrame.AddrFrame.Offset = context.Rbp;
  stackFrame.AddrFrame.Mode = AddrModeFlat;
  stackFrame.AddrStack.Offset = context.Rsp;
  stackFrame.AddrStack.Mode = AddrModeFlat;
  os << "Frame               Code "
        "address\n";
  while (::StackWalk64(
      IMAGE_FILE_MACHINE_AMD64, hProcess,
      hThread, &stackFrame, &context, nullptr,
      ::SymFunctionTableAccess64,
      ::SymGetModuleBase64, nullptr)) {
    DWORD64 pc = stackFrame.AddrPC.Offset;
    DWORD64 frame =
        stackFrame.AddrFrame.Offset;
    if (pc == 0) {
      os << "Null address\n";
      break;
    }
    os << "0x" << (PVOID)frame << "  "
       << addressToString(pc) << "\n";
  }
}
```

Listing 2

But how does it *work*?

In the x64 world, stack walking uses the same logic that is used to support exception handling. The compiler generates some metadata (tagged as "xdata" and "pdata") which is bound into the executable image and is available at runtime.

You can dump out this data with the **`dumpbin`** program supplied with Visual Studio (see, for example, Listing 4).

The function **`SymFunctionTableAccess64`** is the one used by the stack walker to obtain the address of the function table entry metadata for the various code addresses found during stack unwinding. This data provides, among other things, information about the size of the current stack frame and the offset of the return address. The stack walking logic uses this data on each iteration to work up the stack to the calling frame and to update the **`stackFrame`** and **`context`** data to reflect this new frame.

This logic does *not* require any additional debug information that might be present in the PDB file, it simply uses the read-only tables in the binary.

What is in the PDB file?

If we delete the PDB file and re-run the program, it is easy to see what information from the PDB file is used by the stack trace code: the same **number** of stack frames is produced with the same **addresses** (subject to any relocations performed by Address Storage Layout Randomisation) but the **names** for the functions inside the executable and the **source file** information are no longer printed; these are being obtained from the PDB file.

The Microsoft PDB files contain a lot of data. For the example program, I have a PDB file that is over 17 times larger than the executable! All we have used so far is a small part of the overall data – to map **`addresses`** to **`function names`** and **`source file`** information.

However, there is also a huge amount of detail available for the types and variables inside the program. There is enough detail that we can, for instance, generate the full definition of the data members and class hierarchy for the C++ classes used by the program or, as we do next, to introspect on the variables within the program. Note that the full type information is not *always* available – for example Microsoft's public symbol files for the Windows binaries normally only expose function names.

The PDB file format is not, to the best of my knowledge, publicly documented but there are various public APIs to read the data. However, I have found that the documentation is often quite thin on detail and this can make it quite slow to successfully make use of the API in your own programs. See, for example, the [dbghelp.h] documentation.

Getting the names of local variables at each point in the call stack

The first step we take towards our goal is to use the DbgHelp function **`SymEnumSymbols`** to enumerate the local variables at each point in the call stack and then simply printing the names of these variables to demonstrate the enumeration works.

We add this functionality by writing a new function, **`showVariablesAt`**, which is called on each iteration of the main loop in the **`stackTrace`** function. This function first calls **`SymSetContext`** (which requires populating a slightly *different* stack frame structure: **`IMAGEHLP_STACK_FRAME`**) to ensure the subsequent call to **`SymEnumSymbols`** will search at the location of the call site. For each variable found, a callback function we provide is called by the symbol engine, passing us a pointer to the symbol information and a user-supplied value.

(Note that the callback function is also passed a **`SymbolSize`**, which we ignore here because the information is also available in the **`Size`** field of the **`SYMBOL_INFO`** structure.)

The **`SymEnumSymbols`** function operates in a variety of modes – you can, for instance, use it to enumerate *all* symbols within a binary file matching a specified filter string. The callback function invoked for each symbol found allows the option of terminating early if the item sought has been found. In our use case, we want to enumerate all the local symbols in scope

The SYMBOL_INFO structure that we are passed in the callback contains a number of fields obtained from the PDB information for the module being examined

```
Frame               Code address
0x0000007CF88FE0A0  0x00007FFD554CCEA4 NtWaitForSingleObject + 20
0x0000007CF88FE140  0x00007FFD530B19CE WaitForSingleObjectEx + 142
0x0000007CF88FE180  0x00007FFD38672E24 Thrd_join + 36
0x0000007CF88FE1E0  0x00007FF67DB85C2F std::thread::join + 95   C:\Program Files
(x86)\...\include\thread(130) + 32 bytes
0x0000007CF88FE380  0x00007FF67DB81C60 printStack + 112   c:\article\TestStackWalker.cpp(65)
0x0000007CF88FE3C0  0x00007FF67DB81D2C process + 76   c:\article\TestStackWalker.cpp(76)
0x0000007CF88FF7A0  0x00007FF67DB81DB1 test + 65   c:\article\TestStackWalker.cpp(87)
0x0000007CF88FF7D0  0x00007FF67DB81DE9 main + 9   c:\article\TestStackWalker.cpp(90) + 9 bytes
0x0000007CF88FF820  0x00007FF67DB8AEA9 invoke_main + 57
d:\agent\_work\4\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(79)
0x0000007CF88FF890  0x00007FF67DB8AD4E __scrt_common_main_seh + 302
d:\agent\_work\4\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288) + 5 bytes
0x0000007CF88FF8C0  0x00007FF67DB8AC0E __scrt_common_main + 14
d:\agent\_work\4\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(331)
0x0000007CF88FF8F0  0x00007FF67DB8AF3E mainCRTStartup + 14
d:\agent\_work\4\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp(17)
0x0000007CF88FF920  0x00007FFD53B07034 BaseThreadInitThunk + 20
0x0000007CF88FF9A0  0x00007FFD55482651 RtlUserThreadStart + 33
```

Listing 3

at a given call site, so we provide `"*"` as the filter and always return `TRUE` from our callback function to ensure we continue to enumerate.

This user-supplied value can be used to pass arbitrary data to the callback; here we use the common technique when calling such a C API from C++ and pass a pointer to an instance of a user defined structure as the user defined value, dereference this in the callback function, and finally call its `operator()` (see Listing 5, on facing page).

The `SYMBOL_INFO` structure that we are passed in the callback contains a number of fields obtained from the PDB information for the module being examined. We will need several of these fields in order to successfully decode the values of the variables we are interested in. The first pair we are interested in are `Name` and `NameLen`, as these provide us with the name of each variable found.

However, we should also consider the `Flags` field. We are only interested in symbols where `SYMFLAG_LOCAL` is set and we should also exclude any symbols marked with `SYMFLAG_NULL`. We will be using *other* flags in the `Flags` field in due course.

We will enrich the contents of the `EnumLocalCallBack` function call operator as we proceed, but the first implementation is quite simple (see Listing 6).

The `showVariablesAt` function (Listing 7) populates the structures and invokes `SymEnumSymbols`.

Finally we must add a call to this function to the end of the main loop in `stackTrace`:

```
showVariablesAt(os, stackFame, context);
```

```
C:> dumpbin /unwindinfo TestStackWalker.exe
...
          Begin    End      Info      Function Name
...
  0000000C 00001EF0 00001F69 000188D4  ?process@@YAXAEAVSource@@@Z (void __cdecl process(class Source &))
    Unwind version: 1
    Unwind flags: None
    Size of prologue: 0x09
    Count of codes: 1
    Unwind codes:
      09: ALLOC_SMALL, size=0x3
...
```

Listing 4

```
struct EnumLocalCallBack {
  // Called by the Symbol Engine
  static BOOL CALLBACK
  enumSymbolsProc(PSYMBOL_INFO pSymInfo,
                  ULONG /*SymbolSize*/,
                  PVOID UserContext) {
    auto &self =
        *(EnumLocalCallBack *)UserContext;
    self(*pSymInfo);
    return TRUE;
  }
  EnumLocalCallBack(
      const SimpleStackWalker &eng,
      std::ostream &os,
      const STACKFRAME64 &stackFrame,
      const CONTEXT &context)
      : eng(eng), os(os),
        frameOffset(frameOffset),
        context(context) {}
  void operator()(
      const SYMBOL_INFO &symInfo) const;
private:
  const SimpleStackWalker &eng;
  std::ostream &os;
  const STACKFRAME64& stackFrame;
  const CONTEXT &context;
};
```

Listing 5

```
// Simplest useful implementation
void EnumLocalCallBack::
operator()(const SYMBOL_INFO &symInfo) const {
  if (!(symInfo.Flags & SYMFLAG_LOCAL)) {
    // Ignore anything not a local variable
    return;
  }
  if (symInfo.Flags & SYMFLAG_NULL) {
    // Ignore 'NULL' objects
    return;
  }
  std::string name(symInfo.Name, symInfo.NameLen);
  os << "  " << name << '\n';
}
```

Listing 6

```
void SimpleStackWalker::showVariablesAt(
    std::ostream &os,
    const STACKFRAME64 &stackFrame,
    const CONTEXT &context) const {
  EnumLocalCallBack callback(
      *this, os, stackFrame, context);
  IMAGEHLP_STACK_FRAME imghlp_frame = {0};
  imghlp_frame.InstructionOffset =
      stackFrame.AddrPC.Offset;
  SymSetContext(hProcess, &imghlp_frame, nullptr);
  SymEnumSymbols(
      hProcess, 0, "*",
      EnumLocalCallBack::enumSymbolsProc,
      &callback);
}
```

Listing 7

With all this in place we now get a list (Listing 8) of the local variables at each function in the stack trace (or at least, those for which the debug information is available).

Note that the order in which the local variables are enumerated does not match declaration order in the source code.

How to identify the types of these variables

The PDB information for each symbol also includes type information. This information is held by index (since multiple variables can have the same type) and the index to this information is provided in the **`TypeIndex`** field of the **`SYMBOL_INFO`** structure.

We add a new function to the **`SimpleStackWalker`** to encapsulate adding the type to the variable name:

```
void decorateName(std::string &name,
                  DWORD64 ModBase,
                  DWORD TypeIndex) const;
```

We call this with the **`name`** to add type information before printing it – note that the function needs to *modify* the name because the declaration rules in C and C++ may result in embedding the variable name in the complete declaration (for example **`void (*func)()`**).

The **`decorateName`** function makes use of another function in the symbol library, **`SymGetTypeInfo`**. This function provides access to various attributes of the type, selected by the **`IMAGEHLP_SYMBOL_TYPE_INFO`** enumeration type passed as the fourth argument. The actual data is returned in the final argument, where the format of the data depends on the value of the **`GetType`** parameter.

We provide a member function, **`GetTypeInfo()`**, that adds **`hProcess`** as the first argument to avoid having to specify this everywhere.

```
0x000000BBDEAFE0E0  0x00007FFD554CCEA4 NtWaitForSingleObject + 20
0x000000BBDEAFE180  0x00007FFD530B19CE WaitForSingleObjectEx + 142
0x000000BBDEAFE1C0  0x00007FFD38672E24 Thrd_join + 36
0x000000BBDEAFE220  0x00007FF7CF505C2F std::thread::join + 95   C:\Program Files
(x86)\...\include\thread(130) + 32 bytes
  this
0x000000BBDEAFE3C0  0x00007FF7CF501C60 printStack + 112   c:\article\TestStackWalker.cpp(65)
  ss
  hThread
  thr
0x000000BBDEAFE400  0x00007FF7CF501D2C process + 76   c:\article\TestStackWalker.cpp(76)
  source
  local_k
  local_i
  local_l
  local_j
0x000000BBDEAFF7E0  0x00007FF7CF501DB1 test + 65   c:\projects\articles\2021-09-
...
```

Listing 8

There are many different classes of symbol information, all accessed using this method and the type index. We pass **TI_GET_SYMTAG** to **SymGetTypeInfo** to provide the *tag* type of the corresponding symbol information. These tag values are defined in the enumeration **SymTagEnum** in `cvconst.h` (found in the `DIA SDK\include` subdirectory of Visual Studio, which is not by default in the include path) or alternatively from `DbgHelp.h`, if you define the symbol **_NO_CVCONST_H**.

Since each tag holds different information the tag is used as the condition for a **switch** statement. For the purposes of this article, only four types are of interest and I describe each in turn and show the code for that **case** statement.

1. Built-in data types

The value **SymTagBaseType** is used for 'built-in' data types, such as **int** and **double**. The **TI_GET_BASETYPE** and **TI_GET_LENGTH** arguments to **SymGetTypeInfo** provide the underlying type (taken from the **BasicType** enumeration, for example **btUInt**) and the data length (for example, 4).

The code uses a helper function, **std::string getBaseType(DWORD baseType, ULONG64 length)**, to convert the data to C++ data types such as **unsigned short**.

The **getBaseType** function uses a data structure holding type, length, and corresponding C++ type name, for example:

```
...
{btUInt, sizeof(unsigned short),
  "unsigned short"},
{btUInt, sizeof(unsigned int), "unsigned int"}
...
```

In action, **getBaseType** just returns the name found in the matching element of this structure. The complete **case** statement is then this:

```
case SymTagBaseType: {
  DWORD baseType{};
  ULONG64 length{};
  getTypeInfo(modBase, typeIndex,
              TI_GET_BASETYPE, &baseType);
  getTypeInfo(modBase, typeIndex,
              TI_GET_LENGTH, &length);
  name.insert(0, " ");
  name.insert(
      0, getBaseType(baseType, length));
  break;
}
```

2. User defined types

The value **SymTagUDT** is used for user defined types, such as **Source** in our example code.

The first call we make in the function uses **TI_GET_SYMNAME** value, which retrieves the full type name as a wide character string, where **strFromWchar** simply creates an **std::string** from a W**CHAR***:

```
case SymTagUDT: {
  WCHAR *typeName{};
  if (getTypeInfo(modBase, typeIndex,
                  TI_GET_SYMNAME,
                  &typeName)) {
    name.insert(0, " ");
    name.insert(0, strFromWchar(typeName));
    // We must free typeName
    LocalFree(typeName);
  }
  break;
}
```

3. Pointers and arrays

Pointers and arrays are identified by the **SymTagPointerType** and **SymTagArrayType**, respectively. In both cases the dependent type is obtained using **TI_GET_TYPEID** and we recursively call **decorateName** on this type index. (See Listing 9.)

```
case SymTagPointerType: {
  name.insert(0, "*");
  recurse = true;
  break;
}
case SymTagArrayType: {
  if (name[0] == '*') {
    name.insert(0, "(");
    name += ")";
  }
  DWORD Count{};
  getTypeInfo(modBase, typeIndex,
              TI_GET_COUNT, &Count);
  name += "[";
  if (Count) {
    name += std::to_string(Count);
  }
  name += "]";
  recurse = true;
  break;
}
```

Listing 9

The recurse logic is common and is at the end of the **decorateName** function:

```
if (recurse) {
  DWORD ti{};
  if (getTypeInfo(modBase, typeIndex,
                  TI_GET_TYPEID, &ti)) {
    decorateName(name, modBase, ti);
  }
}
```

Listing 10 (overleaf) is a stack trace with the names and types of local variables.

Note that the type of **source** in the **process** function is shown as **Source *** although the argument is actually passed by reference. In the PDB file, the distinction in the C++ code between pointers and references is lost.

Showing the actual *values* for local variables

We now know the name and the type of our local variables, what about their **value**?

In an unoptimised program, local variables are held in the stack frame; if we look at the assembly output from compiling the program we can see this (produced when we add **/Fasc** to the command line):

```
local_i$ = 32
...
?process@@YAXAEAVSource@@@Z PROC; process
...
  0001589 44 24 20 mov DWORD PTR local_i$[rsp], eax
```

The compiler output uses a symbolic name for the variable and uses this value as an offset from the stack pointer (rsp).

In the symbol engine, this is indicated in the **SYMBOL_INFO** by a flag value of **SYMFLAG_REGREL**. The base register is provided in the **Register** field and the offset (32 for **local_i**, in this example) is supplied in the **Address** field.

There is a large enumeration in `cvconst.h` listing all the various register values – the one we want here for **local_i** is **CV_AMD64_RSP** (which is 335).

We can encapsulate the access to the register value by creating a **struct** and a helper function:

```
Frame                Code address
0x000000CD0F6FE110   0x00007FFD554CCEA4 NtWaitForSingleObject + 20
0x000000CD0F6FE1B0   0x00007FFD530B19CE WaitForSingleObjectEx + 142
0x000000CD0F6FE1F0   0x00007FFD36972E24 Thrd_join + 36
0x000000CD0F6FE250   0x00007FF61B674F4F std::thread::join + 95   C:\Program Files
(x86)\...\include\thread(130) + 32 bytes
  std::thread *this
0x000000CD0F6FE3F0   0x00007FF61B671E40 printStack + 112   c:\article\TestStackWalker.cpp(64)
  std::basic_stringstream<char,std::char_traits<char>,std::allocator<char> > ss
  void *hThread
  std::thread thr
0x000000CD0F6FE430   0x00007FF61B671F0C process + 76   c:\article\TestStackWalker.cpp(75)
  Source *source
  int local_k
  int local_i
  int local_l
  int local_j
0x000000CD0F6FF810   0x00007FF61B671F81 test + 49   c:\article\TestStackWalker.cpp(85)
...
```

Listing 10

```
struct RegInfo {
  RegInfo(std::string name, DWORD64 value)
      : name(std::move(name)), value(value) {}
  std::string name;
  DWORD64 value;
};
RegInfo getRegInfo(ULONG reg,
                   const CONTEXT &context);
```

This function returns the correct name and value for the supplied **reg**; at this point the 'Minimal viable product' is:

```
RegInfo getRegInfo(ULONG reg,
                   const CONTEXT &context) {
  switch (reg) {
  case  CV_AMD64_RSP:
    return RegInfo("rsp", context.Rsp);
  }
  return RegInfo("", 0);
}
```

We will come back to this function before we have finished....

So the steps we need to obtain the value of the variable are:

- detect it is a register relative value
- add the offset to the corresponding register value
- read **Size** bytes from the resulting address.

In code this looks like Listing 11, where **eng.readMemory** is a simple wrapper for **ReadProcessMemory** that adds the current **hProcess**. Listing 12 (overleaf) shows the stack trace with names, types, and values of local variables.

Hurrah! We have successfully walked the stack and printed the values of the (simple) local variables we find. We could, if we wished, expand the code further to print out the contents of C++ classes by reflecting on the fields and their offsets.

However, the code so far has been demonstrated against an *unoptimised* program.

What happens when we start to optimise the code?

As many readers are likely to be already aware, it is usually harder to debug optimised code because of the changes made to the executable code during optimisation.

Here are a few of the troublesome optimisations:

- code movement, so things no longer occur in the order of the source code syntax
- heavy use of registers rather than storing values on the stack
- elimination of 'dead stores' (values stored but not subsequently loaded)
- inline function calls.

We can some see these at work in the example program if we enable **/O1** – the local variables displayed in the stack trace for the **process** function are shown as:

```
Source *source
int local_k
int local_i
int local_j
```

The compiler has eliminated **local_l** which you might have noticed was written to but not read. The *compiler* noticed too – a debug build gives a warning:

```
C4189: 'local_l': local variable is initialized
but not referenced
```

The optimised build elides storing the return value into **local_l**, and doesn't even write any information for the variable into the pdb.

Secondly, the *values* are no longer shown. This is because the optimiser is now using **registers** to store the values – they do not need to be stored on the stack in the function.

```
if (symInfo.Flags & SYMFLAG_REGREL) {
  const RegInfo reg_info =
      getRegInfo(symInfo.Register, context);
  if (reg_info.name.empty()) {
    opf << " [register '"
        << symInfo.Register << "']";
  } else {
    opf << std::hex
        << " [" << reg_info.name << " + "
        << symInfo.Address << "]";
    if (symInfo.Size != 0 &&
        symInfo.Size <= 8) {
      DWORD64 data{};
      eng.readMemory(
          (PVOID)(reg_info.value +
                  symInfo.Address),
        &data, symInfo.Size);
      opf << " = 0x" << data;
    }
    opf << std::dec;
  }
}
```

Listing 11

```
Frame                Code address
0x00000097900FE570  0x00007FFD554CCEA4 NtWaitForSingleObject + 20
0x00000097900FE610  0x00007FFD530B19CE WaitForSingleObjectEx + 142
0x00000097900FE650  0x00007FFD2E512E24 Thrd_join + 36
0x00000097900FE6B0  0x00007FF7196A62CF std::thread::join + 95   C:\Program Files
(x86)\...\include\thread(130) + 32 bytes
  std::thread *this [rsp+60] = 0x97900fe6f8
0x00000097900FE850  0x00007FF7196A1E70 printStack + 112   c:\article\TestStackWalker.cpp(64)
  std::basic_stringstream<char,std::char_traits<char>,std::allocator<char> > ss [rsp+60]
  void *hThread [rsp+20] = 0xc4
  std::thread thr [rsp+38]
0x00000097900FE890  0x00007FF7196A1F3C process + 76   c:\article\TestStackWalker.cpp(75)
  Source *source [rsp+40] = 0x97900fe8d0
  int local_k [rsp+28] = 0x7
  int local_i [rsp+20] = 0x5
  int local_l [rsp+2c] = 0xfda93c3e
  int local_j [rsp+24] = 0x3
0x00000097900FFC70  0x00007FF7196A1FB1 test + 65   c:\article\TestStackWalker.cpp(85)
  int return_value [rsp+20] = 0x799c244e
  Source source [rsp+30]
...
```

Listing 12

If we examine the assembly output from the compiler we see:

```
  lea     rcx, OFFSET FLAT:??_C@_05PFHNGCBD@This?5@
  call    printf

; 69   :   int local_j = printf("is ");

  lea     rcx, OFFSET FLAT:??_C@_03FLKGGKMB@is?5@
  mov     ebx, eax
```

The complier is using the 32bit register **ebx** to hold the value of **local_i**.

On the x86 instruction set, a general purpose register can be treated as a 64-bit, a 32-bit, a 16-bit, or an 8-bit value. Writing to the 32-bit value, for instance, modifies only the lower 32 bits of the full 64-bit register value. Hence, the **context** at this point will have the **ebx** value as the low 32 bits of the value in `context.Rbx`.

To see this information in the symbol engine we check another flag in the **Flags** field: **SYMFLAG_REGISTER**.This flag indicates that the value of the variable is held in a register (and the field **Register** holds the register used – in this case **CV_AMD64_EBX**).

The first thing we need to do is to implement a fuller version of the **getRegInfo()** function we used to decode the values of the stack based variables in the *unoptimised* case.

There are two things we need to do to this function; the first one is to add the other general purpose registers to the **switch** statement and the other thing we need to do is to mask the values for the registers which are using only *part* of the 64bit general purpose register.

So, when processing **local_i**, the line in the **switch** statement that will be executed is:

```
  case CV_AMD64_EBX:
    return RegInfo("ebx", context.Rbx & ~0u);
```

We then add handling for the **SYMFLAG_REGISTER** flag to the function call operator of **EnumLocalCallBack**, just after the existing code for the **SYMFLAG_REGREL** flag, like Listing 13.

With these changes we now get values printed for the local variables in the optimised build too. Listing 14 shows the stack trace with the names, types, and values of local variables in an optimised build.

```
} else if (symInfo.Flags &
           SYMFLAG_REGISTER) {
  opf << "  " << name;
  const RegInfo reg_info =
      getRegInfo(symInfo.Register, context);
  if (reg_info.name.empty()) {
    opf << " (register '"
        << symInfo.Register << "\')";
  } else {
    opf << " (" << reg_info.name << ") = 0x"
        << std::hex << reg_info.value
        << std::dec;
  }
}
```

Listing 13

```
Frame                Code address
0x000000A0EA72E390  0x00007FFD554CCEA4 NtWaitForSingleObject + 20
0x000000A0EA72E430  0x00007FFD530B19CE WaitForSingleObjectEx + 142
0x000000A0EA72E460  0x00007FFD4B2E2DFF Thrd_join + 31
0x000000A0EA72E5F0  0x00007FF72B7A3Df2 printStack + 2f2   c:\article\TestStackWalker.cpp(62) + 47 bytes
  std::basic_stringstream<char,std::char_traits<char>,std::allocator<char> > ss [rsp + 50]
  void *hThread [rsp + 40]
  std::thread thr [rsp + 30]
0x000000A0EA72E620  0x00007FF72B7A3F4F process + 79   c:\article\TestStackWalker.cpp(75)
  Source *source (rdi) = 0xa0ea72e650
  int local_k (esi) = 0x7
  int local_i (ebx) = 0x5
  int local_j (ebp) = 0x3
0x000000A0EA72F9F0  0x00007FF72B7A440F test + 115   c:\article\TestStackWalker.cpp(85)
  int return_value (ebx) = 0x673ae2c3
  Source source [rsp + 20]
...
```

Listing 14

But ... what? Where does the *right* register value come from?

The code changes we had to make to display local variables in an optimised build were quite small. The 'magic' is that we were provided with the *correct* value of `Rbx` in the context record. When we read the initial thread context in the `stackTrace` method:

```
GetThreadContext(hThread, &context);
```

then I see the value of `Rbx` as **0**. Where, you might wonder, does the runtime find the value of **5** which `Rbx` had further up the call stack?

The 'Register usage' documented in [x64abi] states that the `Rbx` register value must be preserved when a function is called, and restored on return. The register set is basically divided into ones that must be restored – also called 'non-volatile' and ones that are 'scratch' – also called 'volatile'.

So, if the called function wants to use the register, it must save it somewhere. There is no point using another register to save it as the called function could simply use the other register itself so the value is saved on the stack, and restored when the function returns. This of course applies to the process function as well, so the function prolog contains the instruction:

```
mov     QWORD PTR [rsp+8], rbx
```

and the function epilog reverses this:

```
mov    rbx, QWORD PTR [rsp+48]
```

(The stack offsets are *different* because the function manipulates the stack pointer during its prolog and epilog, and the ordering in the epilog is not the reverse of that in the prolog.)

This works well during normal flow, but what if an exception is thrown somewhere? The runtime needs to ensure the register convention is maintained otherwise the code catching the exception might find a local variable, held in a register, had suddenly changed value!

However it would be expensive (and hard!) for the runtime to try and do this by disassembling the code in each function as it unwinds in order to work out which registers were saved and what frame offset should be used to restore them.

This information is also saved in the unwind meta-data I touched on briefly in the discussion of stack tracing ('But how does it *work*?')

We can examine this data for the `process` function as before using dumpbin, but this time on the optimised program (Listing 15).

The table contains the information for each non-volatile register and how it should be restored. At runtime the unwind logic uses this information as it works up the stack to restore the register values at each level. The symbol engine code does *exactly* the same thing when you produce a stack trace – it reads the unwind metadata to update the `context` with the register values that were currently at each call site.

More extreme optimisation

It's not possible to undo the effect on debugging of all optimisations. As we saw, even in our simple example, `local_1` is not saved. The return value from `source()` is returned in the `Eax` register, but this is a *volatile* register and here the fourth instruction in `printStack` overwrites the `Rax` register and this value is lost forever.

A particular problem is debugging inlined functions. Inlining not only removes the function prolog and epilog, but it also allows the compiler to further optimise the instructions in the called function as part of the body of the caller. This results in code where the assembly instructions executed may toggle back and forth between logically different functions.

Windows debuggers are able to make use of additional data in the PDB file which identifies where the various parts of the inlined functions end up in the binary. This allows the debugger in Visual Studio to make a reasonable stab at debugging even quite heavily optimised code.

I'm not going to attempt to do this here.

How can we debug and get performance?

As a developer there is a tension between performance and debuggability.

The ideal case is where the program behaves in a sufficiently similar way with and without optimisation, so you can run an interactive debugger against an unoptimised build and get the same behaviour as with the released product.

This is the well known pattern of having separate 'Release' and 'Debug' builds.

If this works in your case, it is likely to be the easiest way to resolve problems. Of course, this pattern only works if you are able to reproduce a problem originally occurring with a Release build when using a Debug one.

More nuanced control is possible, however, with care.

The naïve approach of mixing together object files from a Debug and Release build, unfortunately, very rarely works. This is because many compiler flags differ between the two projects, which means things like structure sizes and layouts may not match.

However, you can change *just* the optimisation setting for individual files in the Release build and for even finer control you can change the optimisation setting for individual functions.

This can be very useful when you know roughly which functions are involved in a failure case but cannot, for whatever reason, use the full Debug build.

Let us try this out in our simple example. If we compile with full optimisation (for example with the compiler option `/Ox`) our stack trace is now not very useful. Listing 16 (overleaf) contains output from a fully optimised program.

```
C:> dumpbin /unwindinfo TestStackWalker.exe
...
            Begin    End      Info      Function Name
...
  00000264 00003F00 00003F86 0001095C  ?process@@YAXAEAVSource@@@Z (void __cdecl process(class Source &))
    Unwind version: 1
    Unwind flags: None
    Size of prologue: 0x14
    Count of codes: 8
    Unwind codes:
      14: SAVE_NONVOL, register=rsi offset=0x40
      14: SAVE_NONVOL, register=rbp offset=0x38
      14: SAVE_NONVOL, register=rbx offset=0x30
      14: ALLOC_SMALL, size=0x20
      10: PUSH_NONVOL, register=rdi
```

Listing 15

```
This is a test
Frame               Code address
0x000000A72BAFE420  0x00007FFB259ACEA4 NtWaitForSingleObject + 20
0x000000A72BAFE4C0  0x00007FFB230619CE WaitForSingleObjectEx + 142
0x000000A72BAFE4F0  0x00007FFB20272DFF Thrd_join + 31
0x000000A72BAFE6B0  0x00007FF7602F1E6A printStack + 346   c:\article\TestStackWalker.cpp(62) + 49 bytes
  std::basic_stringstream<char,std::char_traits<char>,std::allocator<char> > ss [rsp+80]
  void *hThread [rsp+70]
  std::thread thr [rsp+40]
0x000000A72BAFFA80  0x00007FF7602F2302 main + 178   c:\article\TestStackWalker.cpp(88) + 178 bytes
0x000000A72BAFFAC0  0x00007FF7602FA830 __scrt_common_main_seh + 268
d:\a01\...\startup\exe_common.inl(288) + 34 bytes
  bool has_cctor [rsp+20]
0x000000A72BAFFAF0  0x00007FFB24A67034 BaseThreadInitThunk + 20
0x000000A72BAFFB70  0x00007FFB25962651 RtlUserThreadStart + 33
```

Listing 16

Here we see that the call stack in our program has collapsed – `main` calls `printStack` directly and the intervening calls to both `test` and `process` have been inlined.

If we wrap the `process` function in a `#pragma optimize("", off)` / `#pragma optimize("", on)` pair then this function will not be optimised and therefore easy to debug, without affecting the optimisation elsewhere in the program. Listing 17 shows output from a fully optimised program with an *unoptimised* function.

Conclusion

In this article, I have sketched some of the techniques used by an interactive debugger to provide values for local variables. I've also shown some of the ways in which more work is needed to do this when optimisations are applied.

The implementers of the various Windows debuggers have done a great job at providing a powerful environment which works amazingly well even on optimised programs.

However, there are times when if you wish to obtain debugging information at runtime you may need to compromise on the performance, at least for the parts of the program under investigation which you are focussing on. ■

References

[dbghelp.h] dbghelp.h header documentation: https://docs.microsoft.com/en-us/windows/win32/api/dbghelp/

[Orr11] Roger Orr, 'Using the Windows Debugging API', *C Vu* 23.1 https://accu.org/journals/cvu/23/1/cvu23-1.pdf

[Orr12] Roger Orr, 'Using the Windows Debugging API on Windows 64', *C Vu*, 23.6 https://accu.org/journals/cvu/23/6/cvu23-6.pdf

[x64abi] Microsoft x64 Software Conventions: https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions

Source code

The full source code for this article can be found at: https://github.com/rogerorr/articles/tree/main/Debugging_Optimised_Code

```
This is a test
Frame               Code address
0x0000006E236FE4F0  0x00007FFB259ACEA4 NtWaitForSingleObject + 20
0x0000006E236FE590  0x00007FFB230619CE WaitForSingleObjectEx + 142
0x0000006E236FE5C0  0x00007FFB20272DFF Thrd_join + 31
0x0000006E236FE780  0x00007FF7FA041E6A printStack + 346   c:\article\TestStackWalker.cpp(62) + 49 bytes
  std::basic_stringstream<char,std::char_traits<char>,std::allocator<char> > ss [rsp+80]
  void *hThread [rsp+70]
  std::thread thr [rsp+40]
0x0000006E236FE7C0  0x00007FF7FA0420FC process + 76   c:\article\TestStackWalker.cpp(75)
  Source *source [rsp+40] = 0x6e236fe7f0
  int local_k [rsp+28] = 0x7
  int local_i [rsp+20] = 0x5
  int local_l [rsp+2c] = 0xd9e328d8
  int local_j [rsp+24] = 0x3
0x0000006E236FFB90  0x00007FF7FA04224D main + 125   c:\article\TestStackWalker.cpp(89) + 125 bytes
0x0000006E236FFBD0  0x00007FF7FA04A830 __scrt_common_main_seh + 268
d:\a01\...\startup\exe_common.inl(288) + 34 bytes
  bool has_cctor [rsp+20]
0x0000006E236FFC00  0x00007FFB24A67034 BaseThreadInitThunk + 20
0x0000006E236FFC80  0x00007FFB25962651 RtlUserThreadStart + 33
```

Listing 17

Stufftar Revisited

Personal projects can provide valuable learning opportunities. Ian Bruntlett shares a system call surprise he discovered while extending stufftar.

In *Overload* 132, I presented stufftar, a program used by me to backup key parts of my filesystem [Bruntlett16]. Since then, I have occasionally had the need to find out: what has changed in my filesystem since a particular backup file was created? It remained a 'would like to have' option until I was experimenting with the GNU **find** command line utility. It has a **-newer** option which, while searching a sub-directory, causes it to list any files newer than a reference file. So this command will list any files in my `TECH-Manuals` folder newer than the file `TECH-Manuals.tar.gz`:

```
find ~/TECH-Manuals/ -newer ~/TECH-Manuals.tar.gz
```

Because I wasn't sure I could remember the name of the **-newer** option, I wrote a bash shell script inspired by the above command. And wrote it so that I could pass additional parameters to find. The script is in Listing 1 and is called like this:

```
newtar ~/TECH-Manuals ~/TECH-Manuals.tar.gz
```

The script requires two parameters – the path to the backed up directory and the name of the backup file. For example, given the directory `TECH-Manuals` and a backup of it, `TECH-Manuals.tar.gz`, running:

```
find ~/TECH-Manuals/ -newer ~/TECH-Manuals.tar.gz
```

currently gives this output:

```
TECH-Manuals
TECH-Manuals/tm-processes.html
TECH-Manuals/tm-index.html
TECH-Manuals/tm-processes.html~
TECH-Manuals/tm-index.html~
TECH-Manuals/tm-platforms.html
TECH-Manuals/tm-platforms.html~
```

That is useful – but what if you want to know more about those files? You can pass in **printf** options. This example lists when the new files were last updated:

```
newtar TECH-Manuals TECH-Manuals.tar.gz -printf
"%t %p\n"
```

which gives this output:

```
Sun Jul 18 21:16:08.7651258860 2021 TECH-Manuals
Wed Jul  7 18:35:57.6009844760 2021 TECH-Manuals/
tm-processes.html
Sun Jul 18 21:16:08.7651258860 2021 TECH-Manuals/
tm-index.html
Wed Jul  7 18:06:16.9915321630 2021 TECH-Manuals/
tm-processes.html~
Fri Jul  9 16:30:47.2928147850 2021 TECH-Manuals/
tm-index.html~
Wed Jul  7 18:34:51.5270892980 2021 TECH-Manuals/
tm-platforms.html
Wed Jul  7 18:05:39.4933397510 2021 TECH-Manuals/
tm-platforms.html~
```

Since I wrote **stufftar**, I have learned about the shellcheck program that critiques shell programming style. This new script is quietly accepted by shellcheck.

```
#!/usr/bin/env bash
# Ian Bruntlett, 20th April 2021 - 7th July 2021
# newtar - file to list entries in a tar file
# newer than the date of a 2nd file
# Based on this command: find ~/TECH-Manuals/
# -newer ~/TECH-Manuals.tar.gz
# 2021-07-07 - updated using advice from
# shellcheck
function Usage
{
  echo Usage: "$0" dir-of-source-files tar-file-
that-backed-up-that-dir
  echo Example: "$0" Desktop
Desktop_02_April_2021.tar.gz
  echo Note: Extra parameters passed on command
line are passed on to find
}
if [ $# -lt 2 ]; then
  Usage >&2
  echo Incorrect number of parameters - at least 2
expected, received $# >&2
  exit 1
fi

if [ ! -d "$1" ]; then
  Usage >&2
  echo ERROR: "$1" is not a directory >&2
  exit 1
fi

if [ ! -f "$2" ]; then
  ERROR_CODE=$?
  Usage >&2
  echo ERROR: "$2" is not a file \($ERROR_CODE\)
>&2
  exit 1
fi

STARTING_POINT=$1
TIME_REFERENCE=$2
shift
shift
find "$STARTING_POINT" -newer "$TIME_REFERENCE"
"$@"
```

Listing 1

Ian Bruntlett Ian has been programming for some years. He also reads a lot to improve his skills. He volunteers for a mental health charity called Contact (www.contactmorpeth.org.uk) doing various things including running a Computer Wombling Project (refurbishing old computers with Linux for the members) and, perhaps, running Mongoose Traveller games there again one day.

A system call surprise

It seems that every Linux expert says "it's in the man files" but I was always curious to know: how do you find out about what you don't know exists? The **apropos** command (a synonym for **man -k**) can be used to find about things related to a topic. For example, to find out things related to the Hewlett Packard Printing and Scanning utilities, I use this command:

```
apropos hp-
```

which yields this (abbreviated) output::

```
hp-align (1)         - Printer Cartridge Alignment
Utility
hp-check (1)         - Dependency/Version Check
Utility
hp-check-plugin (1) - AutoConfig Utility for
Plug-in Installation
```

So I thought: what if I used a regular expression? I tried this command:

```
man -k "[a-z]"
```

and got a lot of output – 7,519 lines worth. And then I thought… what if I restricted the output to a particular section of the man pages using the **-s** option? To refresh the reader's memory, here is a quick summary of the section numbers:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions, e.g. /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

So running this command lists all the pages on system calls:

```
man -k "[a-z]"  -s 2
```

yielding 496 lines of output, abbreviated here:

```
_newselect (2) - synchronous I/O multiplexing
_Exit (2)      - terminate the calling process
__clone2 (2)   - create a child process
_exit (2)      - terminate the calling process
_llseek (2)    - reposition read/write file
offset
_syscall (2)   - invoking a system call without
library support (OBSOLETE)
_sysctl (2)    - read/write system parameters
accept (2)     - accept a connection on a socket
<snip!>
vserver (2)    - unimplemented system calls
wait (2)       - wait for process to change state
wait3 (2)      - wait for process to change
state, BSD style
wait4 (2)      - wait for process to change
state, BSD style
waitid (2)     - wait for process to change state
waitpid (2)    - wait for process to change state
write (2)      - write to a file descriptor
writev (2)     - read or write data into multiple
buffers
```

Again, I took this as an opportunity to write a supporting shell script so I wrote the one in Listing 2:

```
#!/usr/bin/env bash
# 2021-07-11 Ian Bruntlett
# 2021-07-14 Tidying up heredoc in "Usage"
# function.
# Name   : man-section
# Purpose: To dig around the man pages

function Usage
{
cat <<END-OF-USAGE-MESSAGE
Usage: $0 section-number (from 1 to 9, optional)
       1   Executable programs or shell commands
       2   System calls (functions provided by
the kernel)
       3   Library calls (functions within
program libraries)
       4   Special files (usually found in /dev)
       5   File formats and conventions, e.g. /
etc/passwd
       6   Games
       7   Miscellaneous (including macro
packages and conventions), e.g. man(7), groff(7)
       8   System administration commands
(usually only for root)
       9   Kernel routines [Non standard]
END-OF-USAGE-MESSAGE
}

if [ $# -eq 0 ]; then
  Usage >&2
  exit 0
fi

while [ "$1" != "" ]; do
  man -k "[a-z]" -s "$1"
  shift
done
```

Listing 2

The added bonus of the above bash script is that it can list the contents of multiple sections. So the command **man-section 1 8** will list executables for the user (section 1) and for administrators (section 8).

The command **man-section {1..9}** will list pages for all 9 sections. This gives a strange warning message on my system as there are no installed pages for section 9 on my system.

I know its a cliché but I am perpetually hopping inside a small collection of languages, learning new things and reinforcing existing knowledge. I have been advised that one way to achieve this is to have side projects to exercise knowledge to avoid losing it. I bought some revision flash cards from WH Smith's and am collecting bash questions for me to answer in the future. As time goes by, I'll collect questions for the other languages. I am aware that the shell scripts here are quite basic but I believe that developing them has helped improve my bash skills and possibly provided people with a couple of useful tools. ■

References

[Bruntlett16] Ian Bruntlett (2016) 'Stufftar' in *Overload* 132, published April 2016, available from https://accu.org/journals/overload/24/132/bruntlett_2226/

Executors: a Change of Perspective

Parallelism is powerful. Lucian Radu Teodorescu explains the new C++ proposal for managing asynchronous execution on generic execution contexts.

In software engineering, it's often the case that a change of perspective can dramatically modify the perceived usefulness and complexity of a given piece of code. Probably one of the most famous examples in the C++ world is Sean Parent's `rotate` algorithm [Parent13]; once you realise that a complex loop is essentially a rotate operation, you can simplify how you write the code a lot, but, more importantly, also how you reason about the code.

I believe that most of us have had several of these "Aha!" moments during our programming careers, which significantly improved our understanding of some programming problems. Some of us might remember the joy provoked by the understanding of pointers, backtracking, dynamic programming, graph traversal, etc. We probably often don't pay attention to these moments, and they are constantly happening. Sometimes the trigger is learning a new technology, sometimes it is learning a new trick, and sometimes it is just a clear explanation from a colleague.

But, regardless of the trigger for these moments, oftentimes the result is that the change of perspective brought a positive improvement in understanding. The problem does not change, but the way we approach it is (fundamentally) different. And sometimes the end results are remarkable.

This article is about such a perspective change. In fact, it's about two separate perspective changes: one coming from the C++ standard committee, as they revised and reworded the executors proposal, and one from my side, looking at the proposal in an entirely different way.

However, before diving into this perspective change, I owe the readers an apology.

Errata

In my last article, 'C++ Executors: the Good, the Bad, and Some Examples' [Teodorescu21], I made one major mistake. While trying to pick out some pros and cons of the proposal of the initial set of algorithms ([P1897R3]), I claimed that the proposed executors do not have a monadic bind operation defined. This is fundamentally wrong. I failed to realise that the proposal defines the `let_value` operation, which is precisely the monadic bind. I was confused about the description of the operation and failed to realise that this is the monadic bind I was looking for.

The `let_value` operation receives two parameters: a sender and an invocable, and returns another sender. When the whole computation runs, the value resulting from running the input sender is passed to the given invocable. This invocable is then returning a sender, possibly yielding a different value type. There you go: this is just the monadic bind.

I hope the readers will excuse me for this error.

A new proposal, a new perspective

This year, the C++ standard committee dropped [P0443R14] and related proposals, and combined all the important pieces into a new proposal: *P2300: std::execution* [P2300R1].

The core of the proposal is the same. The proposal envisions that senders and receivers are the basis for expressing asynchronous execution, concurrency, and parallelism in future C++ programs. While the core remains the same, there are some fundamental changes to the new [P2300R1] proposal.

I would categorise these changes into two groups: simplification of the concepts and simplified presentation. Let's analyse them individually.

Simplification of concepts

The first major simplification is the dropping of the concept of `executor` completely, along with any of the related functionalities. The new proposal argues that `executor` concepts should be replaced by schedulers (and senders). The core responsibility is the same for both, and it makes sense to drop one of them. And schedulers have the advantage over executors in that they also have completion notifications; one can get notified when a certain work is completed (whether successfully, with an error or cancelled) with schedulers, but with executors, the same job is much more complex. Thus, schedulers have the advantage here.

The idea of an executor is elementary: *an executor executes work*. Schedulers are not as easy to describe; they can be described by *schedulers represent execution contexts*, or *schedulers represent strategies for scheduling work on execution contexts*. This difference of perceived complexity is somewhat subjective, but I argue that the idea of *executing work* is easier to understand than *scheduling work for execution*.

However, this complexity is just at the surface. One can easily transform a scheduler into an executor. In fact, the paper proposes an algorithm `void execute(scheduler auto sched, invocable auto work)` that practically makes the scheduler behave like an executor. To make things clearer, here is a possible implementation (a bit simplistic, but it works):

```
void execute(scheduler auto sched,
    invocable auto work) {
  start_detached( schedule(sched) | then(work) );
}
```

This function calls three algorithms defined by [P2300R1]. Calling `schedule` on a scheduler object will return a sender that would simply send an impulse and no value to its receiver. The `then` algorithm takes the impulse coming from the scheduler and calls the work. This ensures that the work will be executed whenever the scheduler *schedules* the work, and on whatever execution context is defined in the scheduler. The `then` algorithm also returns a sender. The whole thing is started by `start_detached`, which ensures that the entire composition will start to be executed.

The second change that I would call major is the removal of the `submit` operation and, with that, the cleaning of possible operations on the main

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

the biggest perspective shift that reduces the complexity is the breakdown of the proposal design into user side and implementer side

flexible `connect()`/`start()` pair as shown in this (simplified) definition:

```
void submit(sender auto s, receiver auto recv) {
  start(connect(std::move(s), std::move(recv)));
}
```

This removal makes the important concepts and operations between them easier to understand:

- schedulers are strategies for scheduling work on the execution context
  - one can call `schedule()` on a scheduler to obtain a sender that just sends an impulse on the right execution context
- senders describe work to be done
  - one can call `connect()` on a sender and a corresponding receiver to obtain an operation state object
- receivers are completion notifications and serve as glue between senders
  - senders will call one of `set_value`, `set_done` or `set_error` on them
- operation states represent actual work to be done[1]
  - one can call `start()` on an operation state to actually start the work

The paper envisions that end users will only use schedulers and senders, while receivers and operation states are more of a concern to the library implementers. But, more on that later.

There are more changes that [P2300R1] makes over [P0443R14] and related papers. The new proposal adds more algorithms to make it simpler for users to use this proposal in various contexts, allows implementers to provide more specialised algorithms, removes the need for properties (replacing them with *scheduler queries*, which are much easier to understand for regular programmers), etc. Such changes make the whole proposal more approachable by typical C++ programmers. We won't be diving into these now. I highly encourage the reader to take a look at the proposal…

Simplified presentation

One pleasant surprise of the new proposal [P2300R1] is that it can be easily read, without being a C++ expert. The new proposal is organised in such a way that it can be read by both users and library creators; it's not quite for beginners, but still, it's accessible to a larger audience. Again, I urge the reader to take a look at the [P2300R1] proposal to check this.

One of the first things to notice is that the new proposal starts with clear motivation, a set of guiding priorities and some well-explained examples. These will allow the typical reader to understand much more easily the problem that the proposal tries to solve. Thus, by this new arrangement of the content, the authors make the whole proposal seem less complex.

But probably the biggest perspective shift that reduces the complexity is the breakdown of the proposal design into *user side* and *implementer side*. In other words, the authors clearly delimit between the things that most users will care about, and the things that mostly the implementers will care about. This translates into a great simplification for most users.

In terms of concepts, users will care about schedulers and senders. Schedulers, as mentioned above, describe execution contexts. The only thing that the user can do with them is to obtain a sender. The current proposal allows this only by using the `schedule()` algorithm; in the future, there might be other ways to create senders.

Senders, on the other hand, can be manipulated in multiple ways using sender algorithms. Here, the paper also makes it easier for users to understand the proposed functionality by breaking these sender algorithms into three categories:

- sender factories (`schedule()`, `just()` and `transfer_just()`)
- sender adaptors (`transfer()`, `then()`, `upon_*()`, `let_*()`, `on()`, `into_variant()`, `bulk()`, `split()`, `when_all()`, `transfer_when_all()` and `ensure_started()`)
- sender consumers (`start_detached()` and `sync_wait()`)

Now, even without knowing what these algorithms do, one can intuitively figure out the rough idea behind them. And this helps a lot in improving understandability of the whole proposal.

Arranging user-facing concepts like this is a change of perspective that makes the proposal seem less complex.

Only after presenting all the user-facing concepts does the paper go into describing the design of the concepts and algorithms needed for the implementer. The average user will not care about this. In this section, the paper describes the design of the receivers and of the operation states, how the senders can be customised, and the design decisions that make senders and receivers be performant in many different contexts.

What is commonly known as the proposal of senders and receivers, now splits the senders from receivers, making them part of different sets of concerns. It's just a change of perspective, but this change makes the proposal look simpler.

Key insight: it's all about computations

The second major change of perspective presented in this article is my own change of perspective. Since the last article [Teodorescu21], I have had an *Aha!* moment that made me realise that one can look at schedulers, senders and receivers in a simpler way.

Naming

The main problem that I had with the previous proposal was that the names *sender* and *receiver* are extremely abstract. In fact, one can apply the sender/receiver terminology to function composition, single-threaded

1. We said that senders *describe* work, and operation state objects *represent* actual work. While senders are just pieces of work that need to be glued with receivers, the operation states represent these glued pieces of work. Starting an operation state will execute work, whereas we cannot simply start directly one sender.

one can apply the sender/receiver terminology to function composition, single-threaded algorithms, processes, Unix pipes, architectural components, and a multitude of other concepts in software engineering

algorithms, processes, Unix pipes, architectural components, and a multitude of other concepts in software engineering. Moreover, in the concurrency world, there are many types of useful tasks in which we are not sending and receiving any important data; the sending and the receiving part is not essential, the execution of the task is.

The change of perspective that I'm proposing is the following:

- let's think of senders as *async computations*
- let's think of receivers as *async notification handlers*

Because asynchronicity is implied in this context, I'll drop the term *async* and keep just *computations* and *notification handlers*. I argue that it's easier to explain to users the concepts described in [P2300R1] by using *computations* and *notification handlers* instead of *senders* and *receivers*.

One can easily think of computations as just description of work to be executed in one or multiple execution contexts. Schedulers are then the objects that *schedule* computations: when and where these should start executing. Receivers, if one needs to use them, are the handlers that get notified about the completion of computations. Doesn't sound that complicated.

To make things clearer, I'll make the distinction between *computation* and *computation parts*. Let's look at one simple example:

```
auto c1 = schedule(my_scheduler);
auto c2 = then(c1, some_work_fun);
auto c3 = then(c2, some_other_work_fun);
start_detached(c3);
```

We define three computations: `c1`, `c2`, and `c3`. The first computation is a part of the second computation, which, in turn, is a part of the third computation. When we refer to a *computation*, we refer both to the algorithm that we are using to define it (along with its parameters) and to the base sender that the algorithm might take. Thus, the `c3` computation also includes `c2`. However, a `computation_part` is only the algorithm invocation, without including the base computation.

While a *computation* can be recursively defined, a *computation part* refers only to a narrow description of an operation that needs to be executed as a part of a chain.

Figure 1 tries to illustrate the recursion of computations. `c3` defines the entire computation, and it consists of 3 parts (colored differently). `c3` recursively contains `c2` which in turn contains `c1`.

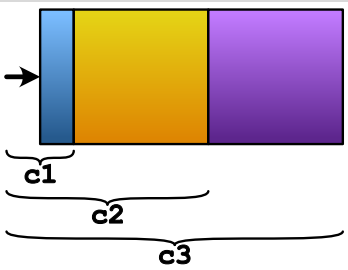


Figure 1

Computations and tasks

In [Teodorescu20], we defined a task to be an independent unit of work. That is, the minimal amount of work that can be executed in one thread of execution, in such a way that it doesn't conflict with other active tasks currently executing. Leaving the *independent* part aside for a moment, let's compare tasks and computations with from the amount of work perspective.

Having [P2300R1] in mind, computations can be created at different levels:

- sub-unit computations; where multiple computations are chained together to form a unit of work (typically run on a single thread)
- unit computations; where a computation is roughly equivalent to a task
- multiple unit computations; where the computation corresponds to more than one task, possible executed on different threads.

The code in the section above showed an example of sub-unit computations. We will show, however, an example containing the two other categories in Listing 1.

It's interesting to notice that, even in this example, sub-unit computations are defined. In fact, most of the time, the framework forces the user to employ sub-unit computations. Some of them are stored in named variables, and some of them are just temporaries. In this example, both `start` and `start2` variables represent sub-unit computations.

The variables `c1` and `c2` represent tasks, so we can say that they are units of work.

In the second part of the example, we've shown a computation that spawns two tasks and ensure they are both executed. In this sense, `c3` describes more work than a task can.

This example is illustrated in Figure 2. `c2` looks just like a task, while `c3` describes two independent tasks, plus the initial split operation and the final join operation. This suggests that computations can represent higher level patterns of concurrency.

```
// unit
sender auto start = schedule(my_scheduler);
sender auto c2 = start | then(save_log_to_disk);
start_detached(c2);
// more than one unit
sender auto start2 =
split(schedule(my_scheduler));
sender auto c3 = when_all(
  then(star2, [] { do_first_work(); }),
  then(star2, [] { do_second_work(); })
);
start_detached(c3);
```

Listing 1

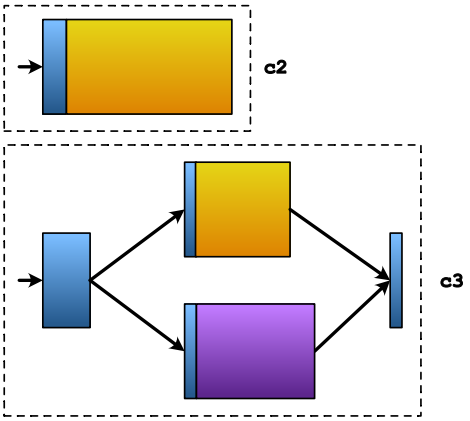


Figure 2

In order for a computation to become greater than a unit of work, one has to call some algorithms that can move work from one thread to another. Example of such algorithms are `split`, `transfer` or `bulk`. We can easily identify these cases just by code inspection.

On the other hand, the distinction between sub-unit and unit computation may not always be clear. The same computation can be both a sub-unit and a unit computation in different contexts; the user of the computation decides whether it is a part of a bigger unit computation or it is spawned into a task.

The bottom line is that these computations (again, they are called *senders* in [P2300R1]) can have different sizes, sometimes much bigger than tasks. This has two important consequences:

1. As computations can be equivalent with tasks, everything that can be modelled with tasks can also be modelled with computations.
2. Computations can be used to express large parts of the concurrent applications.

The first consequence is crucial. We proved in [Teodorescu20] that every concurrent system can be described with tasks, and we don't need mutexes or any other low-level synchronisation primitives (except in the task framework itself). With this consequence in mind, we can apply the same conclusion with a system based on computations (or senders): all concurrent problems can be expressed with just computations, without needing synchronisation primitives in user-code. We can use computations as a global solution to concurrency.

This is where the *independent* part in the definition of a task is important. We assume that the breakdown of work between computations is done in such a way that two computations that can be run in parallel will not produce unwanted race conditions.

If the first consequence allows us to prove the existence of a general algorithm for handling the concurrency algorithm (something that we won't do in this article), the second consequence allows us to easily build complex concurrent applications. Let's quickly dive into this.

Composability of computations

As we've seen above, computations are built on top of each other, starting from small primitives and building up. This is especially visible when we use the pipe operator (see above example). What is on the left of the pipe operator (i.e., `|`) are computations, but also the result of the pipe computations are computations. That is, smaller computations are composed into bigger computations. That is, computations are highly composable.

The examples in the proposal are, of course, small; the authors couldn't have spent most of the papers discussing a large example. We won't do that here either, but we will try to contemplate how such an example might look.

Let's assume that we want to build a web server, with functionality similar to Google Maps. One needs to implement operations for downloading map data, for searching the map and for routing through the map, plus other less-visible operations (e.g., getting data for points-of-interest, downloading pictures, searching similar points of interests, etc.). Some operations might be bigger (e.g., calculating routes) and some might be smaller (e.g., downloading data).

If we ignore the concerns related to network communication, security and data packaging, the application needs to have some *request handlers* that, given some inputs, will generate the right outputs to be sent to the user. Let's focus on this core part. Each type of request will have a corresponding handler. These handlers can vary in size for anything between small and big. But, we can express any of these handlers as one computation. Each of these computations can be then made by joining in multiple smaller computations.

A computation that will implement the routing algorithm will probably be broken down into computations that will perform a general analysis of the situation, read the corresponding map data into memory, perform a graph routing algorithm to find the best path, compute the properties of the best route found, gather all the data that needs to be sent down to the client, etc. All of these can be implemented again in terms of computation. Moreover, they can be executed on multiple threads of execution inside the same server, or can be distributed across multiple servers. We can represent all of this using computations.

For example, Listing 2 shows how one can encode the part that reads the data needed for such a request.

Some brief explanations for this example:

- starting from the data received as argument, `read_data_computation()` will first break the data into tiles and then attempt to load the tiles
- the loading of the tiles is done in parallel by using the `bulk` algorithm; we divide the work to be done into smaller chunks, one per `tile`, calling `do_read_tile` to read a `tile`'s worth of data;
- `load_tiles_async` returns a computation/sender; the way to integrate it into the upper level function is to use the monadic bind; this is done by using the `let_value` algorithm; the reader hopefully sees why I had the urge to show an example of `let_value`

Hopefully, you can understand the main idea behind our sketch example. We can compose computations into more and more complex computations, until we represent large chunks of the application as one bigger computation. This is extremely powerful.

Now, the point of this exercise was to show you that we can approach complex concurrency problems from a top-down perspective, and we've encapsulated the concurrency concerns into computations.

Applying this technique, we can have a structured approached to concurrency. We can easily define the concurrency structure of our programs, and we can make it as visible as any other major architectural structure. Here, when I say *structured concurrency*, I'm only referring to the high-level concerns of what it means to have a structured approach; please see Eric Niebler's blog post on Asynchronous Stacks and Scopes [Niebler21] for a discussion of structured concurrency focusing on lexical scopes (what I would call the low-levels of structured programming).

```
sender auto read_data_computation(in_data_t data)
{
  return just(data) | then(break_into_tiles)
    | let_value(load_tiles_async);
}
sender auto load_tiles_async(tiles_t& tiles) {
  size_t num_tiles = tiles.count();
  return bulk(num_tiles, do_read_tile);
}
void do_read_tile(size_t tile_idx, tiles_t&
tiles)
{...}
```

Listing 2

An analogy with iterators

We've seen that computations and the corresponding algorithms are powerful mechanisms that:

- allow us to build any concurrent application
- allow us to build complex concurrent applications by composing simpler functionalities

This is excellent. But we still don't have a good answer to whether the proposed abstractions are too complex. The perceived complexity of a software system is hard to define, and probably complexity of abstractions are even harder to define, so I don't have a definitive answer here. Instead, I'll try to provide a few hints.

Probably the best way to shed some light into this matter is an excellent analogy I've heard recently: *senders/receivers are like iterators*. I think this really captures the essence of the problem.

Coming from other programming languages, it may seem that iterators are too complex. However, they are proven to be the right abstraction to separate containers from algorithms. Moreover, C++ programmers are accustomed to iterators, and they don't consider it as an overly complex feature.

Yes, it is probably easier to write `my_vector.sort()` than using iterators, but with C++20 we can express the same thing as `std::ranges::sort(my_vector)`. It's not bad at all.

But, even if we didn't have ranges, the advantages of having iterators far outweighs the complexity costs that we are paying.

Let's enumerate some of the benefits that the [P2300R1] proposal brings us:

- ability to represent all computations (from simple ones to complex ones)
- ability to represent computations that have inner parts that can be executed on different execution contexts (e.g., different threads, different computation units, different machines)
- can use computations to solve all concurrency problems (without using synchronisation primitives in the user code)
- composability
- proper error handling and cancellation support
- no memory allocations needed to compose basic computations
- no blocking waits needed to implement most of the algorithms
- allows flexibility in specialising algorithms, thus allowing implementors to create highly efficient implementations
- ability to interoperate with coroutines

All these put together seem to suggest that [P2300R1] brings the right abstractions in the C++ world. We can build correct and efficient applications with them. The standard committee seemed to hit the sweet spot with the abstraction level of senders and receivers. Not too low, so that it would be extremely hard for users to build concurrent applications, and not so high that we lose efficiency.

Raw tasks and executors might be a bit higher level or a bit easier to use. But they do not provide all the benefits that this proposal does; for example, error handling and achieving no memory allocations for composing computations is harder to obtain with raw tasks.

Yes, tasks and executors might be simpler on the surface, but senders/receivers have more advantages. Just like iterators are better than the apparently simpler algorithms-as-methods approach.

And, probably just like with iterators, one of the biggest challenges in the coming years for the C++ community is to find ways to teach computations to new programmers. This is something that may not be as easy as it sounds. But we must embark full speed on this endeavour.

A final perspective

During the course of this article, we showed that changing the perspective on senders and receivers, while maintaining the same core ideas, can result in a reduction of complexity. We argued that these abstractions will allow us to solve, in general, all concurrency problems, without needing blocking synchronisation in user code, and also the composability of computations allows us to easily represent with them any part of the system. Eventually, we made an analogy between these computations and iterators.

Although we might have some challenges teaching the new model, the end result is probably worth it. If everything goes well, we would have a good solution to concurrency. We hope to fix a concurrency model that has plagued the software industry for decades.

But is this the final perspective we will have on senders/receivers, computations or concurrency? Most probably not. History teaches us that we often come up with new perspectives that make old models look more appealing without changing the core concepts. Software engineering is a discipline of knowledge acquisition (see, for example, Kevlin's brilliant presentation [Henney19]), so, changing the perspective (into a better one) is probably the best weapon to tame complexity.

Perspective change is something that happens a lot to me as well. And it's not just happening to me out of the blue, it's something that I constantly work towards: I constantly strive to refine my assumptions and thus my perspectives. Thus, I know already that this is not going to be my final perspective; it's a better perspective, which will be someday superseded by yet another perspective.

I do not write articles to show I know something, but because in the process of laying down arguments my understanding becomes better; in fact, I can't remember a single article that I have written for which I fully knew the conclusions beforehand. And, of course, I write because I believe that the readers too can benefit from this process. Hopefully, they can forgive me for the lack of definite solutions. ■

References

[Henney19] Kevlin Henney, 'What Do You Mean?', *ACCU 2019*, https://www.youtube.com/watch?v=ndnvOElnyUg

[Niebler21] Eric Niebler, 'Asynchronous Stacks and Scopes', 2021, https://ericniebler.com/2021/08/29/asynchronous-stacks-and-scopes/

[P0443R14] Jared Hoberock et al., 'P0443R14: A Unified Executors Proposal for C++', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html

[P1897R3] Lee Howes, 'P1897R3: Towards C++23 executors: A proposal for an initial set of algorithms', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1897r3.html

[P2300R1] Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, 'P2300R1: std::execution', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2300r1.html

[Parent13] Sean Parent, C++ Seasoning, GoingNative, 2013, https://www.youtube.com/watch?v=W2tWOdzgXHA

[Teodorescu20] Lucian Radu Teodorescu, 'The Global Lockdown of Locks', *Overload* 158, August 2020

[Teodorescu21] Lucian Radu Teodorescu, 'C++ Executors: the Good, the Bad, and Some Examples', *Overload* 164, August 2021

Afterwood

It's useful to step back and evaluate from time to time. Chris Oldwood reflects on reflection.

Way back in 2014, I gave a slightly extended lightning talk at the ACCU Conference titled 'The Art of Code' [Oldwood14]. One of the 'humorous' code snippets I presented lampooned the hidden complexity that can arise from the overuse of reflection based libraries and frameworks, most notably of the dependency injection kind. At the time, I had found myself face-to-face with a C# codebase that was difficult to reason about due to this very affliction and it also lead me to make the following pun on Twitter:

> They say C# & Java developers have a habit of overusing reflection. Given the quality of their code, I'd say they're not reflecting enough...

The vast majority of my Twitter puns are poor attempts at wordplay and throwaway comments, but every once in a while I realise there may be something deeper in the punchline that I wasn't immediately aware of. Consequently, I now find myself reflecting on reflection, which feels awfully meta!

While I had never realised it until very recently, I believe my own interest in reflection is another one of those characteristics which I can attribute to Steve Maguire's 90's book *Writing Solid Code*. One of the ideas he introduces quite early on is the notion that when you find a bug, before fixing it, you ask yourself some questions, such as "How could I have prevented this?" and "How could I have automatically detected it?" The book goes on to introduce the use of `ASSERT`s to verify assumptions, along with cranking up optional compiler diagnostics and using static analysers to work smarter instead of harder.

Initially, I limited this practice to what the book was addressing – writing code – but as my experience as a programmer grew and I got to branch out into other disciplines, I found myself adopting the questions more and more whenever something went awry. Around the same time, Ed Nisley's column in *Dr Dobbs Journal* began to explore some of the post mortems that NASA had been publishing and while I took solace in even their ability to fail spectacularly on occasion with the vast resources they had available, their desire to reflect and improve at every scale was an attitude I felt was laudable.

Consequently, that question grew from being a way to help avoid bugs in native code to being one that pervaded more and more aspects of both my professional and personal life. It wasn't just limited to trying to help avoid my own mistakes either but became a more natural question to ask whenever something went wrong. I found I started developing a strong desire to avoid settling for simply fixing incidents in isolation and instead to see if there might be an underlying pattern and therefore find a way to avoid the entire class of incidents in future.

After many late nights in those early days of debugging native code, I was perfectly happy to buy into Maguire's advice around leveraging tooling and practices that would minimise these problems. It didn't take me long to be sold on unit testing and then Test Driven Development (TDD) once I became aware of them and my own inadequacies at manual testing. (Interestingly, Maguire only makes a passing comment about unit tests: *if you have them*, you should run them. I now wonder what unit tests looked like back in the early '90s!)

In a metaphorical sense, you might consider a test-first mentality to be the 'reflection' in a timeline from a post-debugging retrospective – a desire to unearth defects as early as possible is a natural consequence of frustration after being by bitten by a problem that could have easily been avoided. Even if the problem was unavoidable, being in a position to automatically validate and deploy the resulting fix quickly is still a much better place to be than facing another phase of handovers and manual testing.

That level of pessimism which comes with hard won experience was once summed up wonderfully by @fioroco on Twitter in 2017 [Fioroco17]:

> junior dev: "i found the bug"
> senior dev: "i found a bug"

Naturally, there is a balance here. Sometimes it is just a simple mistake but other times it's a more fundamental misunderstanding. I once discovered a memory leak caused by a non-virtual destructor in a base class. Rather than fix it and move on, I checked for a pattern and quickly found the same author had done it another 19 times and, although not in my team any more, was still only a few desks away. They really appreciated the heads up and I managed to stem the tide in two systems.

On the interviewing front, I've begun to feel that a tendency to reflect on one's work is probably a strong indicator of quality, even if they haven't performed it as a formal exercise like a retrospective. I've never really been one to ask classic interview questions like "What does SOLID stand for?" but I wanted candidates to try and embellish more on the 'why' when talking about what they've worked on – I like decisions to be conscious rather than done by rote. Hence if a candidate lists something like SOLID on their CV, I'd ask them "Give me an example of a time when you applied one of the SOLID principles as part of a refactoring." I've found this style of question provides more avenues to explore their thought processes and understanding of a topic.

Even if behaviours are apparently done by rote now, maybe they were once done more consciously and it's been a while since any reflection was performed on the practice or the conditions under which it once applied. It's taken a beating over the last few years due to misuse in some circles but the principle of 'strong opinions, weakly held' is still one I feel is valuable. The school of hard knocks can send us down a more cautious path which often bears fruit for a considerable time, but we should be open to being challenged and accept that progress may have been made in the intervening years. I now find that usually comes from others, but not as direct criticism, more as a fleeting comment which I once might have simply dismissed but now embrace as an opportunity to re-evaluate past decisions.

Maguire's book also taught me to "step through your code [with a debugger]" – a practice which I found invaluable for so many years. Even after adopting unit testing and then TDD, I struggled to let go of the debugger even when writing tests. But what caused me to drop it in the end wasn't more confidence in my ability to solve problems correctly first time but that the massively reduced cost of failure and recovery in modern software development meant that I could be far less paranoid of the consequences. ■

References

[Fioroco17] Tweet posted 4 Dec 2017, available at https://twitter.com/fioroco/status/937824968594853888

[Oldwood14] Chris Oldwood (2014) 'The Art of Code', *ACCU Conference 2014*, available at https://www.slideshare.net/chrisoldwood/the-art-of-code

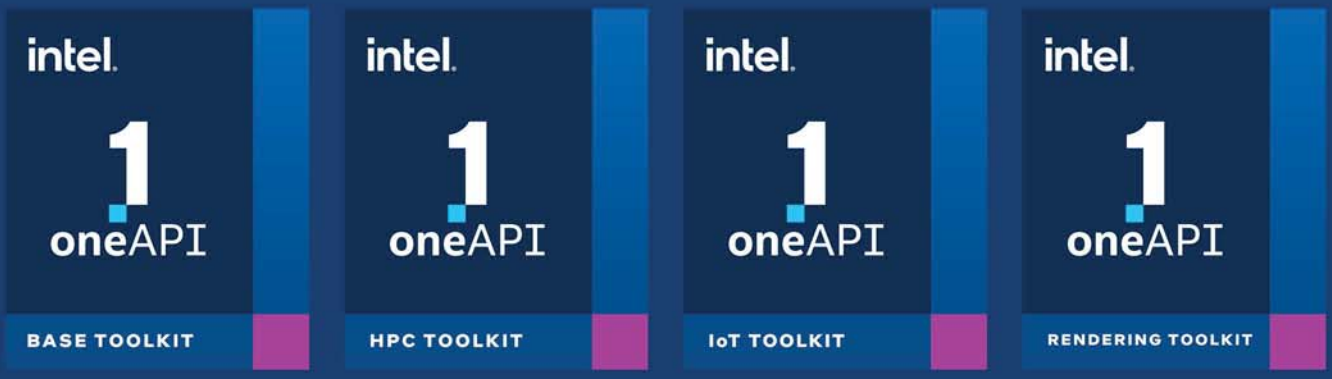
Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He has resumed commentating on the Godmanchester duck race but continues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

intel software

oneAPI: New Era of Accelerated Computing

oneAPI

Take the open, productive path to accelerate cross-architecture computing using Intel® oneAPI Toolkits.



Developers, take advantage of oneAPI's unified, standards-based, cross-architecture programming model that sets you free to develop applications for your choice of architectures. Get full hardware performance using a complete set of proven tools without the limits of proprietary language lock-in.

Learn more: www.qbssoftware.com