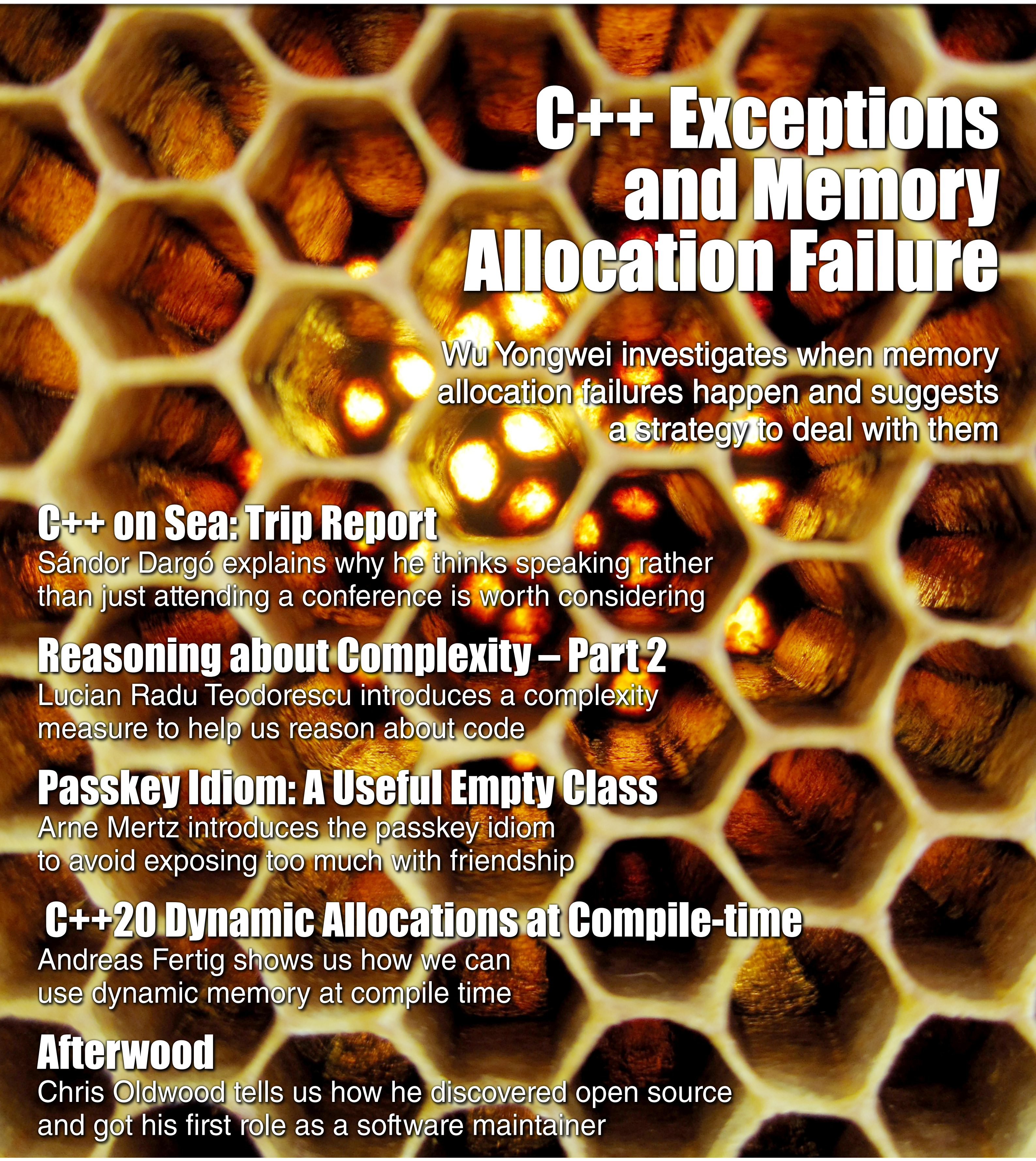
overload 176

AUGUST 2023 £4.50



C++ Exceptions and Memory Allocation Failure

Wu Yongwei investigates when memory allocation failures happen and suggests a strategy to deal with them

C++ on Sea: Trip Report

Sándor Dargó explains why he thinks speaking rather than just attending a conference is worth considering

Reasoning about Complexity – Part 2

Lucian Radu Teodorescu introduces a complexity measure to help us reason about code

Passkey Idiom: A Useful Empty Class

Arne Mertz introduces the passkey idiom to avoid exposing too much with friendship

C++20 Dynamic Allocations at Compile-time

Andreas Fertig shows us how we can use dynamic memory at compile time

Afterwood

Chris Oldwood tells us how he discovered open source and got his first role as a software maintainer

A magazine of ACCU ISSN: 1354-3172

accu

professionalism in programming

Monthly journals, available printed and online

Discounted rate for the ACCU Conference

Email discussion lists

Technical book reviews

Local groups run by ACCU members

ACCU is a not-for-profit organisation.

Become a member and support your programming community.

www.ACCU.org

August 2023
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

Advisors
Ben Curry
b.d.curry@gmail.com
Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi
Steve Love
steve@arventech.com
Chris Oldwood
gort@cix.co.uk
Roger Orr
rogero@howzatt.co.uk
Balog Pal
pasa@lib.hu
Tor Arve Stangeland
tor.arve.stangeland@gmail.com
Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net

Copy deadlines
All articles intended for publication in *Overload* 177 should be submitted by 1st September 2023 and those for *Overload* 178 by 1st November 2023.

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

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.

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

Asleep at the Wheel

Are you cruising on autopilot?
Frances Buontempo wonders if we need to change direction from time to time.

I volunteered at *C++OnSea* [C++ On Sea] last week. It was fun but exhausting and, of course, I didn't get a moment to write an editorial. The conference had a packed schedule, with workshops on Tuesday and talks from Wednesday through Friday. Fortunately, we drove home on the Saturday, giving us a chance to catch up on some sleep over Friday night.

Conferences are a brilliant way to keep your knowledge up to date. They can be expensive though, so if your company won't pay for you, or you are a student or unemployed, the only way to attend might be as a volunteer. To be honest, that wasn't my driving factor for volunteering. It is also wholesome to help out when you can. Each volunteer was scheduled to cover reception, be on hand or be in a specific talk to ensure the speaker had what they needed and didn't run over the allotted time. Rather than having to choose where to be when, I mostly ran on my rails, following the rota. Sometimes cruising on autopilot is a good thing. Decision making can be difficult, so following a timetable means you just 'do the thing' and don't have to decide. I attended talks I might not have considered had I been given a free choice, and each talk was very informative.

There are some situations where running on autopilot with your eyes shut, sleeping at the wheel, may not be ideal. We are promised self-driving cars, which in theory could make this possible. However, we're not there yet. Even if a car has some technology instigated by self-driving car research, such as automatic braking if you get too close to a vehicle in front, or tech to keep you in a lane on a motorway, you cannot safely fall asleep. "Keep your eyes on the road and your hands upon the wheel", as the song goes. Keeping your eyes on the road is a way to say watch where you're going. Where are self-driving cars going? I personally think they are leading to some improvements in car safety, but still wish for transporters or fewer items needing to be driven on roads. As ever, it's worth pausing and considering what you are trying to achieve. A reliable public transport service would suit me, along with places that are safe to walk or cycle along. Just because you think of a potential new technology doesn't mean it's a good idea or even worth pursuing. I am convinced self-driving cars are trying to solve the wrong problem.

So, more generally, we could ask where AI is going. Recently, there has been much discussion around the ethics of AI and whether we need to stop and think about potential dystopian outcomes. Many people are embracing various large language models [Wikipedia-1], including ChatGPT. I mentioned ChatGPT sucking up lots of my time in our last issue [Buontempo23]. Bryce Adelstein Lelbach gave the last keynote at *C++OnSea*, talking about 'AI-Assisted Software Engineering' [Lelbach23]. Specifically, he shared how he tried to cajole ChatGPT into producing code for **`std::unique`** [C++ Reference] allowing for parallel execution. The function eliminates all except the first element from every consecutive group of equivalent elements from a range. ChatGPT frequently eliminated any duplicates, rather than just consecutive duplicates, but did sketch out some code that nearly worked after much prompting. I won't manage to do the talk justice here, but a take home for me was asking the model for options without any code, and then picking an option and asking for code. This gives an opportunity to backtrack if ChatGPT hallucinates itself into a dead-end or tries to use the function it is supposed to be implementing. It is very easy to find yourself going around in circles otherwise. I'm sure Bryce was perfectly capable of writing the function himself, but the experiment with AI generating code was interesting and informative. It certainly didn't prove we can do away with programmers and leave AI to write our code for us. Bryce suggested the AI fumbled its way through the implementation in a manner similar to many humans, taking wrong turns, but certainly suggesting some useful ideas.

C++OnSea was a great opportunity to catch up with people and above all, keep learning. No matter how hard you try, there is always more to learn. It's very easy to stick inside your comfort zone, and not notice new features that might improve your code. C++ is continuing to evolve and there is so much to learn. Taking a step back from the daily grind and taking time to listen and reflect is always good. A conference jam-packed with information is even better. I overheard several people saying "Ooh, I didn't realise that." No matter how well informed you think you are, there's always more to learn. I'm writing a C++ book at the moment, aimed at people who want to catch up on what they have missed since C++11 [Buontempo]. I felt a little shy about mentioning this at a C++ conference, but it turns out several people are interested. I had assumed everyone there would already know everything. I was wrong. There's nothing wrong with knowing you have gaps in your knowledge and doing something about it. In order to learn, it's good to have a target, otherwise straying off into sidetracks and erroneous details is a problem. Following a course or reading a book can keep you on track. Even better, proposing a talk or writing an article can focus the mind, and help you discover things you don't fully understand. Deadlines and external accountability can stop you drifting off track.

All programming languages evolve over time, or at least ones in use do. As a programmer, you can either keep your eyes firmly shut and stick with older idioms, doing what you learnt years ago, or embrace change. Like all knowledge workers, we need time to open our eyes and continue learning. Do you have some topics you want to learn more about? Take a moment to think of a few. Is there something you can't manage, and your heart sinks each time you need to try it? Multi-threaded code? An algorithm, for example a linked list in an interview? Some UI or database work? Maybe try to think of a tiny project to try for a couple of hours and

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

give yourself time to face your fear. Sometimes you can manage to crack something difficult. You might need help, so ask. The ACCU general email group is always so helpful. Or, try ChatGPT or an alternative (for example, [Amazon]) and see what happens.

Now, we can't all be brilliant or even competent at everything. It's OK to know your limits and delegate to someone else instead. Maybe focusing on what you are already good at and getting better at that is acceptable too. I don't enjoy database or UI work, and can muddle through if needs be, by reading the docs and piecing something basic together. If you need something polished, go ask someone else. Don't sleepwalk into a place or role you don't enjoy if you are privileged enough to have options. Doing something you are not competent at, or hate doing, is unsustainable in the long run.

Jutta Eckstein has been talking about sustainability a lot recently. Her website has a section devoted to the topic [Eckstein]. I attended her session at ACCU 2022 [Eckstein22]. Her abstract said,

> …some forecasts project that in 2030 IT will account for 21% of all energy consumption. The software lifecycle creates direct and indirect carbon emissions: it has a footprint, worsening environmental problems. If we do not change the way we implement software, we will contribute to the increase of the carbon footprint. However, the environmental aspect is not the only one we need to focus on. If we take sustainability seriously, we have to examine software development holistically from these three perspectives: social, economic, and environmental (as defined by the triple bottom line).

Of course, these are some forecasts, and could be wrong, but thinking about environmental impact is important. Jutta frequently mentions taking the environmental, social, and economic footprint of products (and their creation) into account, and encourages exploring how the agile principles can contribute to an organization's sustainability, and how a greater awareness can change your current way of working and contribute to increased sustainability [Agile]. I'm hoping Jutta will write an article for us one day soon. You could argue that software development is partly asleep at the wheel, using more and more resources. We probably need to wake up. Two people at *C++OnSea* independently mentioned massive data centre costs due to huge electricity bills, and one lightning talk demonstrated that thinking about the data structures and algorithms we use can reduce power consumption. It turns out using single instruction multiple data (SIMD) [Wikipedia-2] can help to save the world. The speaker, Andrew Drakeford, starting by pointing out that,

> Being particularly energy-intensive, the data center industry accounts for around 4% of global electricity consumption and 1% of global greenhouse gas emissions.

His talk clearly showed reduced time and power usage when using SIMD to find the `std::max_element`. I think the lightning talks will be online at some point. I'll let you know.

Running on your rails can be fine. In fact, sometimes we need to do things on autopilot. Having to think about each breath or each step would also be unsustainable. However, things change, so sometimes we need to adapt to survive. Our internet has been somewhat intermittent of late, which has made me aware just how reliant I am on looking things up online. I have resorted to looking things up in books recently. I always used to do this and have a huge library, including many of my father's mathematics books, along with double copies of many programming books, since my husband codes too. I have caught myself looking for free PDFs of books I own in preference to getting up and walking a few steps to the bookcase. Without a reliable internet, I have dusted off a few books and done some bonus steps. In some ways, the lack of internet has forced a change. We're switching providers, so normal service may be resumed shortly. It has been interesting to notice just how reliant I am on the internet though. You often don't realise what you're doing day to day until something forces you to do differently. They say a change is as good as a rest. That said, I am hoping we find a better ISP soon, because frankly I do rely on the internet for many things. Looking up references in books is great, and I shall endeavour to use the books I spent money on rather than surfing the internet from time to time. However, being able to send emails is easier with some internet.

Life is often a random walk. Even if we have plans, sometimes the accidental interactions and discoveries change our direction. That doesn't mean we shouldn't have plans. Being asleep at the wheel is not a good idea. Taking a break from time to time, either to go to a conference, have a holiday or take a sabbatical is a sensible idea. Stepping aside to reflect or do something different can be energizing, and lead to insights inspiring new approaches we wouldn't otherwise have thought of.

Doing something new is a good thing too. If you've never written an article, give it a go. The *Overload* team is here to help. If you want to do a guest editor spot, get in touch. *Overload* probably deserves a proper editorial at some point. If you've attended a talk, workshop or conference recently, do send us a write-up. And if you haven't, what's stopping you?

References

[Agile] Agile Sustainability Initiative, *Agile Alliance*, https://www.agilealliance.org/resources/initiatives/agile-sustainability-initiative/

[Amazon] CodeWhisperer: https://aws.amazon.com/codewhisperer/

[Buontempo] Frances Buontempo *C++ Bookcamp* (unpublished – expected 2024) https://www.manning.com/books/c-plus-plus-bookcamp

[Buontempo23] Frances Buontempo, Editorial: 'Production and Productivity' in *Overload* June 2023, available at https://accu.org/journals/overload/31/175/buontempo/

[C++ On Sea] *C++ On Sea* conference: http://cpponsea.uk/

[C++ Reference] `std::unique`, details at: https://en.cppreference.com/w/cpp/algorithm/unique

[Eckstein] Jutta Eckstein, 'Agile & Sustainability' page on her website: https://www.jeckstein.com/sustainability/

[Eckstein22] Jutta Eckstein 'Software for Future' at *ACCU Conference 2022*, details available at: https://accu.org/conf-previous/2022/sessions/software-for-future-whats-the-impact-of-the-agile-manifesto-on-our-carbon-footprint/session/index.html

[Lelbach23] Bryce Adelstein Lelbach 'Endnote: AI-Assisted Software Engineering' (abstract), available at: https://cpponsea.uk/2023/sessions/endnote-ai-assisted-software-engineering.html

[Wikipedia-1] Large language model: https://en.wikipedia.org/wiki/Large_language_model

[Wikipedia-2] Single instruction, multiple data: https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

C++ Exceptions and Memory Allocation Failure

Memory allocation failures can happen. Wu Yongwei investigates when they happen and suggests a strategy to deal with them.

Background

C++ exceptions are habitually disabled in many software projects. A related issue is that these projects also encourage the use of **`new (nothrow)`** instead of the more common **`new`**, as the latter may throw an exception. This choice is kind of self-deceptive, as people don't usually disable completely all mechanisms that potentially throw exceptions, such as standard library containers and **`string`**. In fact, every time we initialize or modify a **`string`**, **`vector`**, or **`map`**, we may be allocating memory on the heap. If we think that **`new`** will end in an exception (and therefore choose to use **`new (nothrow)`**), exceptions may also occur when using these mechanisms. In a program that has disabled exceptions, the result will inevitably be a program crash.

However, it seems that the crashes I described are unlikely to occur… When was the last time you saw a memory allocation failure? Before I tested to check this issue, the last time I saw a memory allocation failure was when there was a memory corruption in the program: there was still plenty of memory in the system, but the memory manager of the program could no longer work reliably. In this case, there was already undefined behaviour, and checking for memory allocation failure ceased to make sense. A crash of the program was inevitable, and and it was a good thing if the crash occurred earlier, whether due to an uncaught exception, a null pointer dereference, or something else.

Now the question is: If there is no undefined behaviour in the program, will memory allocation ever fail? This seems worth exploring.

Test of memory allocation failure

Due to the limitation of address space, there is an obvious upper limit to the amount of memory that one process can allocate. On a 32-bit system, this limit is 2^{32} bytes, or 4 GiB. However, on typical 64-bit systems like x64, this limit is not 2^{64} bytes, but 2^{48} bytes instead, or 256 TiB.

On a 32-bit system, when a process's memory usage reaches around 2 GiB (it may vary depending on the specific system, but will not exceed 4 GiB), memory allocation is guaranteed to fail. The physical memory of a real-world 32-bit system can often reach 4 GiB, so we do expect to see memory allocation failures.

The more interesting question is: What happens when the amount of physical memory is far less than the address space size? Ignoring abnormal scenarios like allocating more memory than the physical memory size at a time (which would likely be a logic error in the program), can a reasonable program still experience memory allocation failures?

The core logic of my test code is shown in Listing 1.

Wu Yongwei Having been a programmer and software architect, Yongwei is currently a consultant and trainer on modern C++. He has nearly 30 years' experience in systems programming and architecture in C and C++. His focus is on the C++ language, software architecture, performance tuning, design patterns, and code reuse. He has a programming page at http://wyw.dcweb.cn/, and he can be reached at wuyongwei@gmail.com.

```
try {
  std::size_t total_alloc = 0;
  for (;;) {
    char* ptr = new char[chunk_size];
    if (zero_mem) {
      memset(ptr, 0, chunk_size);
    }
    total_alloc += chunk_size;
    std::cout << "Allocated "
              << (zero_mem ? "and initialized "
                           : "")
              << total_alloc << " B\n";
  }
}
catch (std::bad_alloc&) {
  std::cout << "Successfully caught bad_alloc "
               "exception\n";
}
```

Listing 1

The program allocates memory repeatedly – optionally zeroing the allocated memory – until it catches a **`bad_alloc`** exception.

(I did not change the *new-handler* [CppReference-1], as I do not usually do this in projects, and it is not helpful in testing whether memory allocation failures can really happen. When the new-handler is invoked, memory allocation has already failed – unless the new-handler can free some memory and make allocation succeed, it can hardly do anything useful.)

The test shows that Windows and Linux exhibit significantly different behaviour in this regard. These two are the major platforms concerned, and macOS behaves similarly to Linux.

Windows

I conducted the test on Windows 10 (x64). According to the Microsoft documentation, the total amount of memory that an application can allocate is determined by the size of RAM and that of the page file. When managed by the OS, the maximum size of the page file is three times the size of the memory, and cannot exceed one-eighth the size of the volume where the page file resides [Microsoft]. This total memory limit is shared by all applications.

The program's output is shown below (allocating 1 GiB at a time on a test machine with 6 GiB of RAM):

```
Allocated 1 GiB
Allocated 2 GiB
Allocated 3 GiB
…
Allocated 14 GiB
Allocated 15 GiB
Successfully caught bad_alloc exception
Press ENTER to quit
```

The outputs are the same, regardless of whether the memory is zeroed or not, but zeroing the memory makes the program run much slower. You can observe in the Task Manager that the memory actually used by the program is smaller than the amount of allocated memory, even when the

If a program encounters a memory allocation failure, other programs will immediately experience memory issues too, until the former one exits

memory is zeroed; and that when the amount of allocated (and zeroed) memory gets close to that of available memory, the program's execution is further slowed down, and disk I/O increases significantly – Windows starts paging in order to satisfy the program's memory needs.

As I mentioned a moment ago, there is an overall memory limit shared by all applications. If a program encounters a memory allocation failure, other programs will immediately experience memory issues too, until the former one exits. After running the program above, if I don't press the *Enter* key to quit, the results of newly opened programs are as follows (even if the physical memory usage remains low):

```
Successfully caught bad_alloc exception
Press ENTER to quit
```

Assuming that a program does not allocate a large amount of memory and only uses a small portion (so we exclude some special types of applications, which will be briefly discussed later), when it catches a memory allocation failure, the total memory allocated will be about 4 times the physical memory, and the system should have already slowed down significantly due to frequent paging. In other words, even if the program can continue to run normally, the user experience has already been pretty poor.

Linux

I conducted the test on Ubuntu Linux 22.04 LTS (x64), and the result was quite different from Windows. If I do not zero the memory, memory allocation will only fail when the total allocated memory gets near 128 TiB. The output below is from a run which allocates 4 GiB at a time:

```
Allocated 4 GiB
Allocated 8 GiB
Allocated 12 GiB
…
Allocated 127.988 TiB
Allocated 127.992 TiB
Successfully caught bad_alloc exception
Press ENTER to quit
```

In other words, the program can catch the **`bad_alloc`** exception only when it runs out of memory address…

Another thing different from Windows is that other programs are not affected if memory is allocated but not used (zeroed). A second copy of the test program still gets close to 128 TiB happily.

Of course, we get very different results if we really use the memory. When the allocated memory exceeds the available memory (physical memory plus the swap partition), the program is killed by the Linux OOM killer (out-of-memory killer). An example run is shown below (on a test machine with 3 GiB memory, allocating 1 GiB at a time):

```
Allocated and initialized 1 GiB
Allocated and initialized 2 GiB
Allocated and initialized 3 GiB
Allocated and initialized 4 GiB
Allocated and initialized 5 GiB
Allocated and initialized 6 GiB
Killed
```

The program had successfully allocated and used 6 GiB memory, and was killed by the OS when it was initializing the 7th chunk of memory. In a typical 64-bit Linux environment, memory allocation will never fail – unless you request for an apparently unreasonable size (possible only for **`new Obj[size]`** or **`operator new(size)`**, but not **`new Obj`**). *You cannot catch the memory allocation failure.*

Modify the overcommit_memory setting?

We can modify the overcommit_memory setting [Kernel], you probably have shouted out. What I described above was the default Linux behaviour, when `/proc/sys/vm/overcommit_memory` was set to 0 (heuristic overcommit handling). If its value is set to 1 (always overcommit), memory allocation will always succeed, as long as there is enough virtual memory address space: you can successfully allocate 32 TiB memory on a machine with only 32 GiB memory – this can actually be useful for applications like sparse matrix computations. Yet another possible value is 2 (don't overcommit), which allows the user to fine-tune the amount of allocatable memory, usually with the help of `/proc/sys/vm/overcommit_ratio`.

In the *don't-overcommit* mode, the default overcommit ratio (a confusing name) is 50 (%), a quite conservative value. It means the total address space commit for the system is not allowed to exceed swap + 50% of physical RAM. In a general-purpose Linux system, especially in the GUI environment, this mode is unsuitable, as it can cause applications to fail unexpectedly. However, for other systems (like embedded ones) it might be the appropriate mode to use, ensuring that applications can really catch the memory allocation failures and that there is little (or no) thrashing.

(Before you ask, no, you cannot, in general, change the overcommit setting in your code. It is global, not per process; and it requires the root privilege.)

Summary of memory allocation failure behaviour

Looking at the test results above, we can see that normal memory allocations will not fail on general Linux systems, but may fail on Windows or special-purpose Linux systems that have turned off overcommitting.

Strategies for memory allocation failure

We can classify systems into two categories:

- Those on which memory allocation *will not* fail
- Those on which memory allocation *can* fail

The strategy for the former category is simple: we can simply ignore all memory allocation failures. If there were errors, it must be due to some logic errors or even undefined behaviour in the code. In such a system, you cannot encounter a memory allocation failure unless the requested size if invalid (or when the memory is already corrupt). I assume you must have checked that *size* is valid for expressions like **`new Obj[size]`** or **`malloc(size)`**, haven't you?

The strategy for the latter category is much more complicated. Depending on the requirements, we can have different solutions:

refactoring old code with RAII (including smart pointers) can be beneficial per se, even without considering whether we want exception safety or not

1. **Use `new (nothrow)`, do not use the C++ standard library, and disable exceptions.** If we turned off exceptions, we would not be able to express the failure to establish invariants in constructors or other places where we cannot return an error code. We would have to resort to the so-called 'two-phase construction' and other techniques, which would make the code more complicated and harder to maintain. However, I need to emphasize that notwithstanding all these shortcomings, this solution is self-consistent – troubles for robustness – though I am not inclined to work on such projects.
2. **Use `new (nothrow)`, use the C++ standard library, and disable exceptions.** This is a troublesome and self-deceiving approach. It brings about troubles but no safety. If memory is really insufficient, your program can still blow up.
3. **Plan memory use carefully, use `new`, use the C++ standard library, and disable exceptions; in addition, set up recovery/restart mechanisms for long-running processes.** This might be appropriate for special-purpose Linux devices, especially when there is already a lot of code that is not exception-safe. The basic assumption of this scenario is that memory should be sufficient, but the system should still have reasonable behaviour when memory allocation fails.
4. **Use `new (nothrow)`, use the C++ standard library, and enable exceptions.** When the `bad_alloc` exception does happen, we can catch it and deal with the situation appropriately. When serving external requests, we can wrap the entire service code with `try ...catch`, and perform rollback actions and error logging when an exception (not just `bad_alloc`) occurs. This may not be the easiest solution, as it requires the developers know how to write exception-safe code. But neither is it very difficult, if RAII [CppReference-2] is already properly used in the code and there are not many raw owning pointers. In fact, refactoring old code with RAII (including smart pointers) can be beneficial *per se*, even without considering whether we want exception safety or not.

Somebody may think: Can we modify the C++ standard library so that it does not throw exceptions? Let us have a look what a standard library that does not throw exceptions may look like.

Standard library that does not throw?

If we do not use exceptions, we still need to have a way to express errors. Traditionally we use error codes, but these have the huge problem that a *universal* way does not exist: `errno` encodes errors in its way, your system has your way, and yet a third-party library may have its own way. When you put all things together, you may find that the only thing in common is that 0 means successful…

Assuming that you have solved the problem after tremendous efforts (make all subsystems use a single set of error codes, or adopt something like `std::error_code` [CppReference-3]), you will still find yourself with the big question of when to check for errors. Programmers that have been used to the standard library behaviour may not realise that *using* the following `vector` is no longer safe:

```
my::vector<int> v{1, 2, 3, 4, 5};
```

The constructor of `vector` may allocate memory, which may fail but it cannot report the error. So you must check for its validity when using `v`. Something like:

```
if (auto e = v.error_status();
    e != my::no_error) {
  return e;
}
use(v);
```

OK… At least a function can use an object passed in by reference from its caller, right?

```
my::error_t process(const my::string& msg)
{
  use(msg);
  …
}
```

Naïve! If `my::string` behaves similarly to `std::string` and supports implicit construction from a string literal – i.e. people can call this function with `process("hello world!")` – the constructor of the temporary `string` object may fail. If we really intend to have complete safety (like in Solution 1 above), we need to write:

```
my::error_t process(const my::string& msg)
{
  if (auto e = msg.error_status();
      e != my::no_error) {
    return e;
  }
  use(msg);
  …
}
```

And we cannot use overloaded operators if they may fail. `vector::operator[]` returns a reference, and it is still OK. `map::operator[]` may create new map nodes, and can cause problems. Code like the following needs to be rewritten:

```
class manager {
public:
  void process(int idx, const std::string& msg)
  {
    store_[idx].push_back(msg);
  }
private:
  std::map<int, std::vector<string>> store_;
};
```

The very simple `manager::process` would become many lines in its exception-less and safe version (Listing 2).

Ignoring how verbose it is, writing such code correctly seems more complicated than making one's code exception-safe, right? It is not an easy thing just to remember which APIs will always succeed and which APIs may return errors.

```
class manager {
public:
  error_t process(int idx,
                  const my::string& msg)
  {
    if (auto e = msg.error_status();
        e != my::no_error) {
      return e;
    }
    auto* ptr =
      store_.find_or_insert_default(idx);
    if (auto e = store_.error_status();
        e != my::no_error) {
      return e;
    }
    ptr->push_back(msg);
    return ptr->error_status();
  }
  …

private:
  my::map<int, my::vector<string>> store_;
};
```

Listing 2

And obviously you can see that such code would be in no way compatible with the current C++ standard library. The code that uses the current standard library would need to be rewritten, third-party code that uses the standard library could not be used directly, and developers would need to be re-trained (if they did not flee).

Recommended strategy

I would like to emphasize first that deciding how to deal with memory allocation failure is part of the system design, and it should not be just the decision of some local code. This is especially true if the 'failure' is unlikely to happen and the cost of 'prevention' is too high. (For similar reasons, we do not have checkpoints at the front of each building. Safety is important only when the harm can be higher than the prevention cost.)

Returning to the four solutions I discussed earlier, my order of recommendations is 4, 3, 1, and 2.

- Solution 4 allows the use of exceptions so that we can catch `bad_alloc` and other exceptions while using the standard library (or other code). You don't have to make your code 100% bullet-proof right in the beginning. Instead, you can first enable exceptions and deal with exceptions in some outside constructs, without actually throwing anything in your code. When memory allocation failure happens, you can at least catch it, save critical data, print diagnostics or log something, and quit gracefully (a service probably needs to have some restart mechanism external to itself). In addition, exceptions are friendly to testing and debugging. We should also remember that error codes and exceptions are not mutually exclusive: even in a system where exceptions are enabled, exceptions should only be used for *exceptional* scenarios. Expected errors, like an unfound file in the specified path or an invalid user input, should not normally be dealt with as exceptions.
- Solution 3 does not use exceptions, while recognizing that memory failure handling is part of the system design, not deserving local handling anywhere in the code. For a single-run command, crashing on insufficient memory may not be a bad choice (of course, good diagnostics would be better, but then we would need to go to Solution 4). For a long-running service, fast recovery/restart must be taken into account. This is the second best to me.
- Solution 1 does not use exceptions and rejects all overhead related to exception handling, time- or space-wise. It considers that safety is foremost and is worth extra labour. If your project requires such safety, you need to consider this approach. In fact, it may be the *only* reasonable approach for real-time control systems (aviation, driving, etc.), as typical C++ implementations have a high penalty when an exception is really thrown.
- Solution 2 is the worst, neither convenient nor safe. Unfortunately, it seems quite popular due to historical momentum, with its users unaware how bad it is…

Keep in mind that C++ is not C: the C-style check-and-return can look much worse in C++ than in C. This is because C++ code tends to use dynamic memory more often, which is arguably a good thing – it makes C++ code safer and more flexible. Although fixed-size buffers (common in C) are fast, they are inflexible and susceptible to *buffer overflows*.

Actually, the main reason I wanted to write this article was to point out the problems of Solution 2 and to describe the alternatives. We should not follow existing practices blindly, but make rational choices based on requirements and facts. ■

Test code

The complete code for testing the memory failure behaviour is available at either:

- http://wyw.dcweb.cn/mem_alloc_test.zip (for downloading)
- http://wyw.dcweb.cn/mem_alloc_test.cpp.html (for browsing)

You can clearly see that I am quite happy with exceptions. ☺

References

[CppReference-1] cppreference.com, **`std::set_new_handler`**, https://en.cppreference.com/w/cpp/memory/new/set_new_handler

[CppReference-2] cppreference.com, 'RAII', https://en.cppreference.com/w/cpp/language/raii

[CppReference-3] cppreference.com, **`std::error_code`**, https://en.cppreference.com/w/cpp/error/error_code

[Kernel] kernel.org, 'Overcommit accounting',https://www.kernel.org/doc/Documentation/vm/overcommit-accounting

[Microsoft] Microsoft, 'How to determine the appropriate page file size for 64-bit versions of Windows', https://learn.microsoft.com/en-us/troubleshoot/windows-client/performance/how-to-determine-the-appropriate-page-file-size-for-64-bit-versions-of-windows

C++ on Sea 2023: Trip Report

C++ on Sea happened again in June this year. Sándor Dargó explains why he thinks speaking rather than just attending a conference is worth considering.

Last week, from 27th to 30th June, I had the privilege of attending and presenting at *C++ on Sea* 2023 [C++OnSea] for the 4th time in a row! Despite having been accepted as a speaker, I was not sure if I would make it this year, as I changed jobs recently, but my management at Spotify was encouraging and supportive. They granted me the time so that I didn't have to use my vacation days. Also, I am grateful to my family, in particular to my wife, for taking care of the kids alone for the week.

Let me share with you a few thoughts about the conference.

First, I'm going to write about the three talks that I liked the most during the 3 days, then I'm going to share 3 interesting ideas I heard about and then I'll share some personal impressions about the conference.

My favourite talks

Over those few days, I pondered a lot about what makes a talk good and enjoyable. What makes a presenter good, at least for me? While my thoughts are not crystal clear yet, I definitely enjoyed talks that covered 'beginner' topics in depth. Another feeling I have is that good presenters limit the amount of knowledge they want to share so they have enough time to explain and they don't talk at the speed of Eminem.

Special member functions in C++ by Kris van Rens

Kris's talk [vanRens23]about special member functions in C++ is a good reminder of how difficult it can be to write a simple class in C++. Especially if you cannot follow the rule of 0. But do you know about the rule of 0? Or the rule of 5? Or the rule of four and a half? ☺

At first, I was not sure if I want to mention this talk among my favourite ones. But as I listed 2–3 favourite ideas from this talk, I realized that this in fact was one of my best picks.

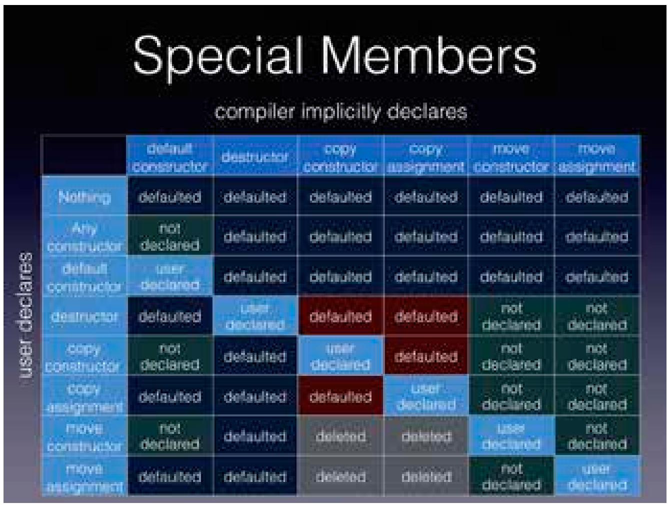
Let's see those ideas.

Have you heard about the Hinnant table [Hinnant20]? The one in Figure 1 shows when you can or cannot rely on the compiler to generate the special functions for you.

Kris shared how you can memorize it easily. While the table has 42–48 fields (depending on whether you count the diagonal), you only need three rules in order to memorize it.

- When the user declares any other constructor then the default constructor is not declared
- When the user declares any copy or move operation or the destructor, then the move operations are not declared.

Sándor Dargó is is a passionate software craftsman focusing on reducing maintenance costs by applying and enforcing clean code standards. He loves knowledge sharing, both oral and written. When not reading or writing, he spends most of his time with his two children and wife in the kitchen or travelling. Feel free to contact him at sandor.dargo@gmail.com



Special Members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Figure 1

- When the user declares any move operation then the copy operations are deleted

Another idea I really appreciated was that we should test special functions. You might think that testing those are cumbersome. But not so! You don't necessarily want to test the internals of a copy constructor. You don't necessarily have to test if all the members are copied promptly. Maybe you want to, but you don't have to go that far.

It's already a great step if you can ensure, with the help of type traits (or concepts) and **`static_cast`**, that a given class satisfies certain characteristics (see Listing 1).

Then even if you modify the class, you make sure that you don't lose its copyablity. Such tests might even enhance your understanding of how certain types of members influence a class.

While I think these tests also serve documentational reasons and they would look great in the header file along with the class declaration,

```
class X{};

static_assert
  (std::is_trivially_destructible<X>{});
static_assert
  (std::is_trivially_default_constructible<X>{});
static_assert
  (std::is_trivially_copy_constructible<X>{});
static_assert
  (std::is_trivially_copy_assignable<X>{});
static_assert
  (std::is_trivially_move_constructible<X>{});
static_assert
  (std::is_trivially_move_assignable<X>{});
```

Listing 1

Will C++ ever be fully safe? No, it's impossible. As a minimum, language safety would mean no undefined behaviour.

probably it's wiser to put them along with the unit tests so that you don't make the compilation of the production code any longer.

One last thought! An explicit `=delete` is better than relying on that others know the Hinnant table as well as you.

Typical C++, but why? by Björn Fahller

Björn Fahller spoke at *C++ On Sea* 3 times this year! He volunteered to replace one of the speakers who sadly couldn't make it to the conference, and he also did a lighting talk.

One of my favourite talks was his presentation about how to use C++'s type system effectively [Fahller23].

No, not because of the great images of jigsaw montages.

No, not because at the end he mentioned my talk from last year as a valuable reference [Dargo22a]. But to be fair, it really touched me. Especially that it was not because I was in the room; it was already mentioned on the references slide.

As I also covered here [Dargo22b], using several `bool` parameters is both difficult to read and dangerous. But it's not only about `bool`s. Adjacent parameters of the same type always have the risk of being mixed up.

Instead of relying on good eyes, you might want to rely on the compiler and use `enum`s and classes with descriptive names.

And as Björn said, don't use type aliases instead of strong types, a type alias is just a comment, nothing more.

An interesting idea he mentioned was how to deal with parameters when you have a bunch of them and many of them would be defaultable. In that case, use a `struct`, let the members have their default values declared in place and then take advantage of C++20's designated initializers [Filipek21]!

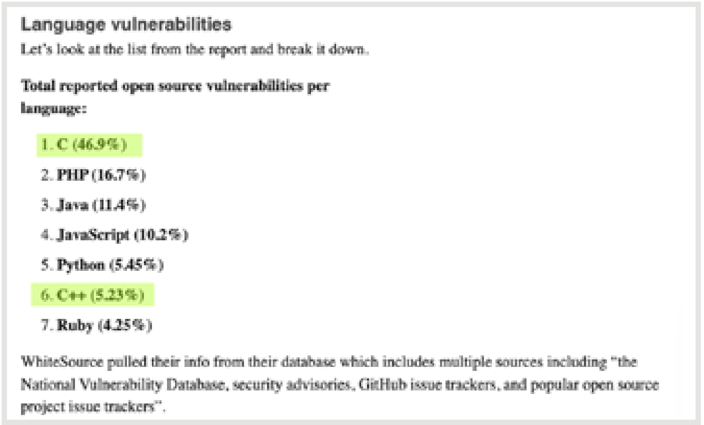
What a nice idea!

C++ and Safety by Timur Doumler

The topic of safety often comes up in C++. It's been an important topic for many years, but the topic has become even more prevalent since the NSA wrote that "exploitable software vulnerabilities are still frequently based on memory issues" and recommended that "the private sector, academia and the U.S. Government use a memory-safe language when possible".

In his talk [Doumler23a], Timur discussed the different forms of safety, and how they relate to correctness. He debunked some myths and shared his view of whether C++ is in trouble or not.

When it comes to safety, we can think about both functional and language safety. When we talk about C++, we are talking about language safety. Language safety can be broken down into memory, thread, arithmetic and definition safety. Timur showed through a set of small and simple examples how much C++ lacks basically any aspect of language safety.



Language vulnerabilities

Let's look at the list from the report and break it down.

Total reported open source vulnerabilities per language:

1. C (46.9%)
2. PHP (16.7%)
3. Java (11.4%)
4. JavaScript (10.2%)
5. Python (5.45%)
6. C++ (5.23%)
7. Ruby (4.25%)

WhiteSource pulled their info from their database which includes multiple sources including "the National Vulnerability Database, security advisories, GitHub issue trackers, and popular open source project issue trackers".

Figure 2

He also showed that even if you have language-safe programs, having a functionally safe, God forbid, correct program is so difficult. In that sense, C++ is not the problem.

But otherwise, how much is C++ the problem? Why did the NSA explicitly target C++?

Those who complain most often speak about 'C/C++'. Anyone who speaks about 'C/C++' shows how little they understand these programming languages. Those are two separate languages!

While it's true that almost 50% of the reported language vulnerabilities are coming from C, C++ is actually only the 6th on the list, behind languages such as PHP, Java and Javascript. Even Python. (See Figure 2 [Doumler23b].)

C++ took a long journey and is full of safety features and it's still getting further safety features. Will they be completely safe? No. Will C++ ever be fully safe? No, it's impossible. As a minimum, language safety would mean no undefined behaviour.

But we need undefined behaviour and we often have to make tradeoffs between safety and performance, portability or cost.

Yet, Timur showed the different strategies we could take to achieve the different kinds of language safety and also shared how viable these strategies would be.

Timur's conclusion is that C++ mostly has a PR issue. C++ isn't as behind so many safety issues as many think. Even so, it's getting safer and many UBs have been or are being removed when this doesn't compromise performance and compatibility, which are often the main reasons behind using C++.

There is no such thing as a safe coding language. Languages call other languages and even so-called safe languages such as Rust have vulnerabilities. On the other hand, the C++ committee should probably be clearer on its strategy and also on how far we have already come.

An interesting talk with full of easy-to-follow examples!

What the **main risks are leading to burnout** and what **best practices** are available for us in case we want to **guard against it**?

My favourite ideas

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

Jonathan Müller's favourite C++ question

Jonathan's talk would have probably been among my favourite ones if he had 30 minutes more to present the same content. My brain is too slow for interesting ideas coming so rapidly! ☺ He talked about C++ features that are either forgotten or undervalued.

He mentioned many interesting topics, and some I'll probably write about more deeply in the coming months (I'll not forget to refer to his talk!), but here I want to mention only one thing.

His favourite C++ question that was originally posted by Richard Smith [Smith19])

```
// Assuming an LP64 / LLP64 system,
// what does this print?

short a = 1;
std::cout << sizeof(+a)["23456"] << std::endl;
```

So, what is the output?

The answer requires quite a few steps. I don't want to go into an explanation in this article: I'd like you to think about it. Here are a few hints:

- What does unary plus do?
- What's the type of `"23456"`?
- What does `sizeof` return?
- What is a little-known characteristic of the built-in index operator?
- What's the precedence of operations in this expression?

If you are stuck and desperately looking for the answer, check it out on Jonathan's site [Muller23].

Bryce Adelstein Lelbach thinks we often treat AI unfairly

In his endnote, Bryce talked about his experiments with ChatGPT and how it was helping him create a parallel algorithm.

Listening to him probably made many of us think that oh, okay, it's hard to use AI-assisted tools effectively, they are still too dumb for this, and they need too many iterations, too many rounds.

But at the end, Bryce reminded us that we are just being unfair towards ChatGPT and other large language models.

Do we expect ourselves to write perfect code on the first run?

Not really, right?

If you post a pull request and someone asks you if there were any bugs in it, would you reply that yes, here they are?!

Not really, right?

ChatGPT *et al.* cannot write perfect code on the first or second run either, but it can analyse its own code more objectively than you or I could our own code. At the same time, it can iterate on code and write better and better solutions of the same problem.

So, let's reconsider how we think about them.

Dr. Allessandria Polizzi shared that boredom can also lead to burnout

Even at a conference dedicated to C++, you might find topics that are not necessarily about the language (such as my talk about clean code), or about software development.

Dr. Allessandria Polizzi spoke about mental health. She shared what the main risks are leading to burnout and what best practices are available for us in case we want to guard against it. Burnout is real and it doesn't simply happen to you, you can prevent it.

There is one risk here that I want to emphasize from her presentation. You might think it's great when you have a low workload. I think that if you are conscious enough of the issue, it's not so bad, but according to research, for most people a low workload can lead to burnout faster than a high workload. Not just burnout but even '**boreout**' is real.

In my opinion, if you have a low workload, take advantage of it. Work on your own initiatives and invest time in learning to get even better at your job.

Nevertheless, it's important to know what are the different factors that can lead to burnout.

Personal impressions

Finally, let me talk about some more personal feelings about the conference.

C++ On Kaizen

I remember that, even last year [Dargo22c], I appreciated the constant improvements at the conference. I think most of the complaints were about lack of water and long queueing times for lunch. The water problem was solved after the first day, and this year, the queueing situation improved a lot too. In different rooms, talks ended at different times right before the lunch break so that not everybody went to eat at the same time. That was a great idea! But what matters more – to me – is the mindset of constant improvement.

Hard to stay an introvert

By the end of the conference, I felt exhausted, but in a good way. I had inspiring discussions with so many people and I even met someone who went to the same high school as me and finished just one year earlier.

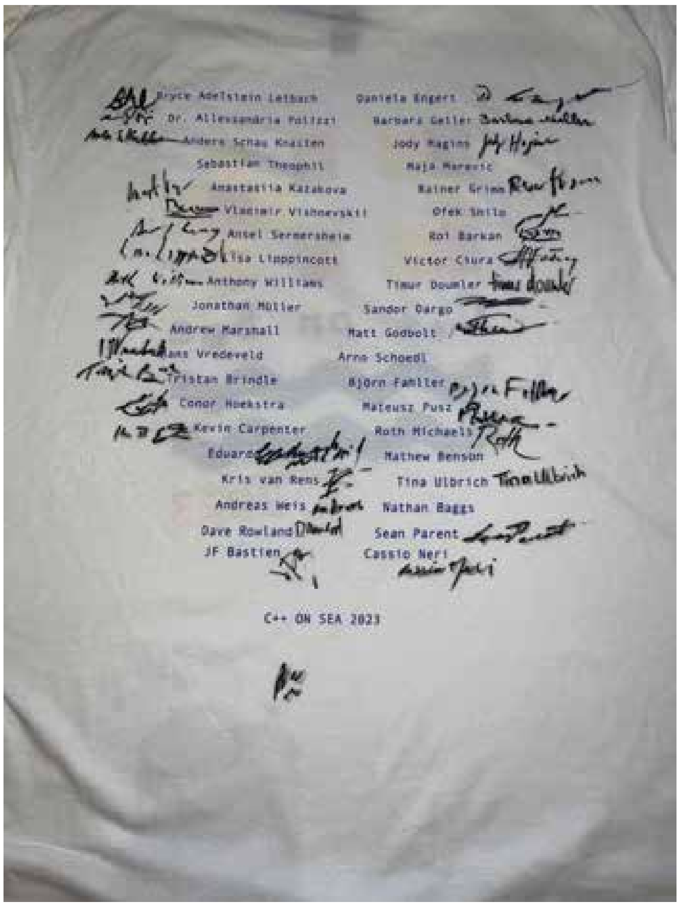
While I'm an introvert and I rarely start conversations with strangers, I tried my best at the conference. And even when I didn't, as a speaker I often got approached by others.

I was on the phone with my wife and I told her that I had got a baseball cap as a swag, though I'm not sure if I would ever use it. She reminded me that she wears such caps. "Oh I remember," I said, "you even have the one signed by Charles Leclerc!" At that moment I realized that I could also get some autographs at the conference. Not on a cap, but on the conference T-Shirt which has the name of all the speakers!

By the end, I had a signature from almost anyone. And mine is just next to the signature of the Explorer of Compilers! How cool is that!

My two talks

This year, my topic(s) were not technical. I signed up for a lightning talk, where I shared my findings on how one can improve his or her job hunt experience. After all, I joined Spotify less than a year ago! Such an important topic for everyone!



Thursday afternoon, I got an hour to speak about why clean code is not the norm [Dargo23]! In particular, about what clean code is, what it has to do with software quality and also how it is related to professional ethics.

At the end, there were some good and/or provoking questions and remarks. I was humbled by the ratio of other speakers in the audience, and I received a lot of great feedback. Even if we didn't agree on everything, my talk was thought-provoking and sparked many discussions.

Conclusion

In this article, I have covered one way that can help you get closer not only to attending but to speaking at conferences. In my opinion, this is way better than just attending, because often (most of) your costs will be covered, you'll learn way more and build more connections. Not to mention that it's easier to convince your management to let you speak at a conference than to buy you a ticket and finance the trip.

C++ On Sea was once again an awesome experience! Great organization, a strong line-up and awesome attendees! I hope I can be back in Folkestone in 2024. ■

Connect deeper

If you liked this article, please

- hit on the like button on the original post
- subscribe to my newsletter (http://eepurl.com/gvcv1j)
- and let's connect on Twitter (https://twitter.com/SandorDargo)!

References

[C++OnSea] The *C++ on Sea* website: https://cpponsea.uk/

[Dargo22a] Sándor Dargó 'Strongly Typed Containers' presented at *C++ on Sea* 2022, available at https://www.youtube.com/watch?v=0cTOqwrvq94

[Dargo22b] Sándor Dargó 'Use strong types instead of bool parameters' posted 5 April 2023 at https://www.sandordargo.com/blog/2022/04/06/use-strong-types-instead-booleans

[Dargo22c] Sándor Dargó 'Trip Report: C++ on Sea 2022', posted on 26 July 2022 at https://www.sandordargo.com/blog/2022/07/27/cpp-on-sea-trip-report

[Dargo23] Sándor Dargó 'Why clean code is not the norm?' (abstract) available at https://cpponsea.uk/2023/sessions/why-clean-code-is-not-the-norm.html

[Doumler23a] Timur Doumler 'C++ and Safety' (abstact) available at https://cpponsea.uk/2023/sessions/cpp-and-safety.html

[Doumler23b] Timur Doumler 'The C++ Undefined Behaviour Survey', posted on Timur.Audio on 14 April 2023 at https://timur.audio/

[Fahller23] Björn Fahller 'Typical C++, But Why?' (abstract) available at https://cpponsea.uk/2023/sessions/typical-cpp-but-why.html

[Filipek21] Bartlomiej Filipek 'Designated initializers in C++20' posted on the *C++ Stories* blog: https://www.cppstories.com/2021/designated-init-cpp20/

[Hinnant20] Howard Hinnant 'How I Declare My class And Why', posted 24 February 2020 at https://howardhinnant.github.io/classdecl.html

[Muller23] Jonathan Müller 'C++ Features You Might Not Know' (slides and video) available at https://www.jonathanmueller.dev/talk/cpp-features/

[Smith19] Richard Smith, quiz question posted on Twitter on 18 March 2019, available at https://twitter.com/zygoloid/status/1107740875671498752?lang=en

[vanRens23] Kris van Rens 'Special member functions in C++' (abstract) available at https://cpponsea.uk/2023/sessions/special-member-functions-in-cpp.html

This article was published on Sándor Dargó's blog in July 2023 and is available at https://www.sandordargo.com/blog/2023/07/05/trip-report-cpp-on-sea-2023.

Reasoning about Complexity – Part 2

Understanding code could increase our productivity by an order of magnitude. Lucian Radu Teodorescu introduces a complexity measure to help us reason about code to tackle complexity.

This is the second part of the article on reasoning and complexity. In the first part, we argued the importance of reasoning in software engineering, and started exploring some dimensions of reasoning that we might apply in our field.

In this part, we use this reasoning to tackle complexity. Starting from Brooks's 'No Silver Bullet' article [Brooks95], we make the distinction between what's *essential* and what's *accidental* in software engineering. To properly reason about these two, we make a sharp distinction between the two. We define a framework for analysing essential complexity in a more formal manner. For accidental complexity, we cannot find such a formal system, but we use the discussion about reasoning from the first part to give us a hint on how we can approach accidental complexity found in software.

If the first part of the article looked like an essay, this part is like a play in 14 acts.

Essential and accidental complexity

Act 1: The actors introduce themselves

Brooks talks about software as having two types of difficulties: essential and accidental [Brooks95]. In the first category, we can find the *difficulties inherent to the nature of the software*, and in the second, *those difficulties that today attend its production but are not that inherent*. In the inherent category, Brooks lists *complexity* (no two parts are alike), *conformity* (there isn't a more fundamental level of software so that we can reduce all the software to that level), *changeability* (software is constantly changing), and *invisibility* (software cannot be drawn in space; software is many-many-dimensional).

Brooks argues that by now we have solved a major part of the accidental difficulties and what's left are essential difficulties, and we cannot get a ten-fold increase in productivity as we cannot improve on these essential difficulties. He lists a series of promising technologies and paradigms and argues that they cannot bring that ten-fold increase in productivity.

If we strictly follow the wording of Brooks, and his categorisation, it makes no sense for us to discuss *essential complexity*. Complexity is always inherent, is always essential. However, in present times, we often shift the terms of Brook's problem into *essential complexity* and *accidental complexity*. For example, Kevlin Henney makes use of these new terms [Henney22b].

But neither of the two approaches properly defines a clean boundary between what's essential and what's accidental. This is always left to interpretation.

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

Act 2: The dilemma

Let's take an example. Let's say we have a project in which we need to create an echo server: it accepts TCP/TLS connections, and whenever it receives a message, it replies with the content of the received message.

By both accounts, there is an inherent complexity of the problem itself (accepting connections, communication protocols, reading and writing messages, etc.) This is most probably essential.

Now, if one chooses to solve this problem imperatively (i.e., using object-oriented programming) or in a functional manner, is this essential or accidental? A naive read of Brooks may suggest that this is essential, not accidental. After all, the paradigm we are using has a great influence on the data structures, algorithms, and function invocations we need to solve the problem. On the other hand, Kevlin argues that this is part of the *accidental*.

Moving on, we are making a choice of a programming language to use, and then probably multiple choices of which technologies to be used in this project. Is this *essential* or *accidental*? When we implement this, we may have a clean implementation, or maybe we end up with a lot of technical debt in our implementation. Does technical debt account for *essential* or for *accidental*?

Furthermore, during the implementation of this project, we choose some tooling to facilitate our development. We are probably using Git, we may want to use CI/CD, we may want to create some architecture documentation, we may want to write some detail design documentation, etc. We may also have processes to follow that dictate which individuals should work on the project, which individuals need to be informed, which need to review, which must approve various deliverables during the lifetime of the project. Most of these seem to be related to the *accidental* part; there are plenty of difficulties that we have to solve in order to get the project complete.

Act 3: An unexpected event; war preparations

To make progress on the previous dilemma, let's set a convention: we talk about the *essential complexity* of a problem as being the complexities inherently associated with the problem, and not what different solutions may look like. If there are two solutions to the same problem, one of them being less complex, and one being more complex, we say that the problem is no more complex than the first solution.

This is consistent with Kevlin's perspective. [Henney22b]

Furthermore, let's assume that we can associate a value with the complexity of a problem and the corresponding solutions. If P is a problem and S_1, S_2, …, S_n are solutions to P, and $C(S_i)$ is the complexity associated with solution S_i, then we can say that the complexity of the problem P is defined by:

$$C(P) = \min C(S_i)$$

That is, the complexity of the problem is the minimum possible complexity of all the solutions.

We count the **complexity of the solution** as being the **sum of complexities associated with each node and each link**. This is a relatively **simplistic model**, and not very precise, but it **gives us a good approximation** of what we need

In this setup, we assume that the complexity function $C(S_i)$ is only a function of the code for solution S_i, and does not relate to any difficulties about producing solution S_i (i.e., processes, build systems performance, etc.).

Furthermore, we want to distinguish between the difficulties associated with the code of solution S_i, and the difficulties related to the tools and processes that were used to produce S_i. We will mainly focus on the actual complexity of the code and ignore any complexity that is not directly visible in the code. Based on the data we have (mostly informal) the main difficulty is dealing with the code itself (for example, we spend more time reasoning about the code than using git to submit a patch).

We also discard any non-functional requirements and constraints that were not explicitly specified in the problem domain. For example, if the problem is just *sort an array of elements*, then *sleep-sort* and *bogo-sort* [Henney22a] are suitable solutions for the problem, and we consider them while evaluating the complexity of the problem.

Let's say that each solution can be represented as a graph; for example, let's say that the nodes are instructions, and the links are relationships between instructions. We count the complexity of the solution as being the sum of complexities associated with each node and each link. This is a relatively simplistic model, and not very precise, but it gives us a good approximation of what we need. It turns out we don't need anything else to have basic reasoning about the complexity.

We can complicate this model by allowing the nodes of a graph to be formed by other graphs, not just by instructions. This way, we build a hierarchy of graphs for representing solutions.

All the complexities of a solution that do not appear as complexities of the problem can be labeled as accidental. We don't have (yet) a good measure for *accidental* complexity.

Reasoning on essential complexity

Act 4: An old ally

The reader may be familiar with an old ally of ours from an older episode named 'Performance Considered Essential' [Teodorescu22]. While arguing why performance is important for all (practical) software problems, we were helped by our friend: **`the_one_algorithm`**. This algorithm solves most practical problems by trying out all possible combinations of outputs (i.e., using backtracking), and selecting the one that matches the expected requirements. That is, we need the requirements of the problem to be encoded as tests. The algorithm would try any possible combination of output values, and checks if the output can be a solution to the problem input.

For a problem that doesn't have explicit performance constraints, if we can find a set of tests that properly captures the requirements of the problem, the use of **`the_one_algorithm`** is a solution to our problem. Thus, we can use this powerful ally to launch an attack on the complexity value of a problem.

For most problems, the easiest way to derive the set of tests is to start analysing the requirements of the problem. We have functional requirements, non-functional requirements (quality attributes) and constraints. Usually, the functional requirements are the only ones that are explicit and directly associated with the problem; however, our construction works even if we make non-functional requirements explicit. And, as we said, we only care about explicit requirements when assessing a problem.

Thus, in most cases, to satisfy our ally, we would iterate over the list of explicit requirements and provide a list of one or more tests that we can apply. This list of tests can then be transformed using conjunction into a global test for a solution of the problem.

Act 5: The first complexity wars

Let us prepare our attempt at conquering essential complexity.

We have a problem P, that has a set of requirements $R_1, R_2, \ldots R_n$. For this set of requirements, we come up with a set of tests $T_1, T_2, \ldots T_n$. A test can be simple or more complex. For each test, we can define an underlying problem, so that means that we can associate a complexity value to it. Let's say the complexity values for the tests will be $C_1, C_2, \ldots C_n$.

Our **`the_one_algorithm`** algorithm also has a complexity. Let's note that with C_0.

It is worth mentioning that the actual complexity values we associate don't matter that much to our approach. For simplicity, we can define a basis, a fixed set of instructions/algorithms that all have complexity equal to 1. For complex operations that are composed of basis operations, we can calculate the complexity appropriately.

For example, it makes sense to associate a complexity value of 1 to our **`the_one_algorithm`** ally. We understand it enough to reason about it, we don't always have to analyse its constituent parts each time we are interested in analysing the complexity of a problem/solution.

Furthermore, to simplify things, we can always find the tests $T_1, T_2, \ldots T_n$ to be independent of each other. That is, the complexity of the overall test will just be the sum of the complexity of the individual requirement tests.

Moreover, we assume that the sequence of tests $T_1, T_2, \ldots T_n$ is the simplest that we can find.

With this, for a problem P for which we find the solution of using **`the_one_algorithm`** with associated tests $T_1, T_2, \ldots T_n$, we find that the complexity of the solution is:

$$C\left(solution\right) = C_0 + \sum C\left(T_i\right)$$

Thus, the complexity of the problem is

$$C\left(P\right) \leq C_0 + \sum C\left(T_i\right)$$

we have managed to define a metric that can approximate essential complexity

In plain English, for every problem, the complexity of the problem is at least one plus the complexity of all the tests we need to fully specify the requirements.

Act 6: The peace treaty

For most problems, the complexity obtained in this way is probably smaller than the complexity obtained by analysing the algorithm itself. (Note to future self: this is not properly argued; the reader didn't receive proper reasoning to support this statement.) Thus, we can approximately define the complexity of the problem as the complexity of the solution involving **`the_one_algorithm`**.

Even if the above statement is not true in all cases, we can still use it to compare two problems, even if the comparison is approximate. For example, we can compare the essential complexity of a problem defined as *sort N elements* with the problem defined as *stable sort N elements*. For the second problem, we have more requirements, thus more tests to be performed, so the complexity is greater:

$$C(P_2) > C(P_1)$$

Thus, for practical purposes, we will use the complexity of our solution involving the backtracking algorithm as an approximation of the essential complexity of the problem:

$$C(P) \sim C_0 + \sum C(T_i)$$

And thus, we have a definition for the essential complexity of a problem, even if this is just an approximation.

Act 7: Aftermath; an example

Let's say that we assign complexities of 1 to the following operations:

- running our backtracking algorithm (**`the_one_algorithm`**)
- accessing elements in an array (either input or output)
- comparing two elements (of the same type) for equivalence or for ordering
- comparing indices
- using the existential or universal quantifier on one variable, with a predicate (predicate complexity is added separately)
- implication operator →

Let's say that we have P_1 as *sort N elements of an array, in place*. This problem can be defined thanks to the following tests (in the interest of space, not extremely formal, more like a sketch):

- T_1: $\forall i \in [0,N), \exists j \in [0,N), array^{orig}[i] == array^{final}[j]$
  in English: all the elements that were initially in the array are still present in the output array
- T_2: $\forall i \in [0,N), \exists j \in [i+1,N),\ array^{final}[i] \leq array^{final}[j]$
  in English: the elements in the final array are sorted

With the above rules, the complexities associated with the tests are $C(T_1) = 5$ (one *forall*, one *exists*, two array accesses and one equality comparison) and $C(T_2) = 5$ (same).

That is, $C(P_1) = 1 + 5 + 5 = 11$.

Let's now take P_2 to mean *search an element X in an array of N elements; if found, return its index (R), otherwise return NULL*. We can have the following tests for this algorithm:

- T_1: $R \neq \text{NULL} \to R \in [0,N) \,\&\&\, array[R] == X$
- T_2: $R = \text{NULL} \to \neg\exists\, i \in [0,N),\ array[i] == X$

With the tests written this way, we can have $C(P_2) = 1 + 4 + 5 = 10$.

Act 8: Enjoying the victory

After a relatively long journey, we have managed to define a metric that can approximate essential complexity. Considering the fact that we started from not knowing what essential complexity is, I hope the reader agrees with me that this is a pretty good result.

This allows us to compare problems in terms of essential complexity, allowing us to say that one problem is more complex than another.

But there is another interesting change that we've achieved here. We managed to simplify our reasoning about the problem by transforming it from something that is inherently complex into a linear sequence of predicates. Instead of having a quadratic reasoning of the problem, we now can apply a linear algorithm for reasoning about its complexity.

Why quadratic? Well, on an inherently complex problem, one can assume that every part of the problem is connected to every other part of the problem. If the problem has *N* parts, then there may be *N*(*N*-1)/2 connections inside the problem.

It turns out that any transformation that enables representation of the problem in a linear form can enable a simplification in how we reason about the problem.[1]

Reasoning on accidental complexity

Act 9: Dark clouds gather again in our minds

We've won the first battle, we've found a way to reason about essential complexity, but we haven't won the war.

By construction, we've moved into essential complexity only what is inherently related to the problem, but all the practical things about various solutions have been left in the accidental complexity part.

If, for example, we have a concrete implementation of a sorting algorithm, we don't have a good way of reasoning about it. Does it matter the choice

1 Please note that the problem is still as complex as before. In our case, the difficulty of the problem moved into the process of generating good tests for the problem. The process of generating a linear sequence of tests is not necessarily linear. But, once we have that transformation done, it's much easier to reason about the problem.

Even the fact that we created a function has introduced a new element into our program that we have to reason about

of programming paradigm, or the choice of programming language, or the choice of the algorithm being used? Of course, it does.

Unfortunately, accidental complexity radiates from essential complexity. Similar to how a black hole radiates light, in the same way, essential complexity continuously generates accidental complexity. Particles split at the border of a black hole, part of them being pulled into a black hole and part of them being emitted as light from the direction of the black hole. Similarly, trying to write code for solving essential parts of the problem always creates more accidental difficulties.

For example, creating functions to solve a particular aspect of the problem always comes with naming them, with dividing the logic in two parts (what's inside the function and what's outside the function) and having different types of coupling between those two parts. Naming and these divisions are not inherently to the problem, so they are accidental complexity. Even the fact that we created a function has introduced a new element into our program that we have to reason about (it provides benefits, but always has costs too).

In all software projects, there is always a dark force in the accidental complexity that we have to constantly face. And, as the main bottleneck is our brain, the only weapon we seem to have against it is by improving our ways of reasoning about the problem.

Act 10: Sorting it out

Let's consider the problem of sorting, in place, an array of numbers. We've already shown a system in which the essential complexity of the problem equals 5. Let's consider now the complexity of a sorting solution, namely insertion sort. Listing 1 shows a C++ implementation.

To analyse the complexity of this algorithm, we would use the metric that we introduced in 'How We (Don't) Reason About Code' [Teodorescu21]. That is, we count all the postconditions that can infer by reading the code. To make things simpler, we would not count the syntactical aspects of the code, and not bother about the types and semantic information present in the code. We would only reason about the possible values. And, even here, we would take some small shortcuts to keep things simple. We would compute the *reasoning complexity* of the code, by counting the number of postconditions we can infer from the code.

```
// inputs: int n, int arr[]
for (int i=1; i<n; i++) {
  int key = arr[i];
  int j=i-1;
  // Move elements in arr[0..i-1] that are 
greater
  // than the key one step right
  while (j>=0 && arr[j]>key) {
    arr[j+1] = arr[j];
    j--;
  }
  // Put the element at the right position
  arr[j+1] = key;
  // Postcondition: arr[0..i] is sorted
}
```

Listing 1

Here it is:

1. `i` is always greater or equal to `1`;
2. `i` is always less than `n` in the body of the loop;
3. `key` always has a value of an array element (`arr[i]`);
4. `j` starts as `i-1`;
5. `j` is never incremented;
6. `j` is decremented each time the body of the `while` loop is run;
7. `j` is always less than `i`;
8. `j` is always greater or equal to `0` in the `while` body;
9. in the `while` body, `arr[j]` is always a valid value in the range `arr[0..i-1]`;
10. in the `while` body, `arr[j+1]` is always a value in the range `arr[1..i]`;
11. if `arr[j]>key` then we move the element `arr[j]` one position right, overwriting the value we have there;
12. while shifting the elements right in the body of the `while` loop, we are not losing the value of any element (considering that the value of `arr[i]` is stored in `key`);
13. in the `while` body `arr[j+1]` is always a value in the range `arr[0..i-1]`;
14. at the end of the `while` loop, if `j>=0` then `arr[j] <= key`;
15. at the end of the `while` loop, `arr[j+1] > key`;
16. at the end of the `while` loop, all the elements `arr[j+1..i-1]` (assuming `j+1<=i-1`) are moved one position to the right (keeping their order);
17. if all the elements in range `arr[0..i-1]` are sorted at the start of the `for` loop, then at the end of the `while` loop, all elements in range `arr[0..j]` (assuming `j>=0`) are smaller than `key`;
18. if all the elements in range `arr[0..i-1]` are sorted at the start of the `for` loop, then at the end of the `while` loop, all elements in range `arr[j+1..i-1]` (assuming `j+1<=i-1`) are greater than `key`;
19. storing the value of `key` at `arr[j+1]` does not lose any value from the original array;
20. if all the elements in range `arr[0..i-1]` are sorted at the start of the `for` loop, then at the end of the `for` loop, all the elements in range `arr[0..i]` would be sorted;
21. at the end of the `for` loop, all the elements original present in the input array will still be present in the array;
22. at the end of the `for` loop, all the elements will be sorted.

In the end, the reasoning complexity of this sorting algorithm is 22, as we have 22 preconditions to complete our reasoning. The astute reader

The more connections there are between the parts of the problem, the more complex the problem is for us, as it gets harder and harder to reason about it

may remark that we went quickly over the items that required induction. A more in-depth analysis would probably yield a bigger complexity for our algorithm.

In the case of this *reasoning complexity*, we counted the number of postconditions that we can deduce from the code. Previously, when measuring the essential complexity, we measured the number of elements of the predicates that describe the problem. We have two slightly different approaches, but their core is the same: counting the number of reasoning units involved in the two things. Generalising, we can say that the two metrics are compatible.

While it's not quite correct, we can compare the essential complexity value of 5 for the problem of sorting in place, with the reasoning complexity of 22 for the insertion sort algorithm. This gives us an indication that the complexity of a solution is, in general, higher than the complexity of the problem. The difference is accidental complexity.

Please note that, for this example, the complexity of the solution is 4.4 times bigger than the essential complexity.

Act 11: Self-inflicted pain

It is unclear to me why this is the case, but it feels to me that software engineers are the most masochistic out of all the engineering disciplines I know of. The amount of self-inflicted pain that software engineers cause is staggering. It feels to be even more than the number of problems that are solved.

Bugs, technical debt, optimistic estimates, late projects, you name it. They all come with accidental complexity.

Starting from the assumption that all engineers want to avoid such pain, the problem lies somewhere between the actions that engineers undertake and the consequences of those actions. I'm trying to avoid going through the rabbit hole of entering a discussion on moral logic.

This disconnect is most probably generated by incomplete reasoning. If I'm doing action *A* now, I must not fully realise that it leads to the consequences *C* that are harmful to me.

This comes back to the ideas that I touched on in the first part of this article. We need to get better at reasoning about different aspects of software engineering. If we do, then maybe we figure out better strategies to reduce the amount of pain and accidental complexity that have left.

Act 12: Linearising the problem

If we have a problem (or a sub-problem for that matter) that is complex to understand, then perhaps linearising the problem will make it easier for us to reason about it.

To make this clearer, let's repeat the reasoning we had above when we analysed the process of reasoning about essential complexity. Let's assume that the problem has *N* parts (potentially each of these parts hiding more complexity). In a system with N parts, there can be $N(N-1)/2$ communication channels. That is a quadratic order of magnitude.

The more connections there are between the parts of the problem, the more complex the problem is for us, as it gets harder and harder to reason about it.

For example, if we have a 10 parts problem, we have 45 connections between these parts. Thus, to fully reason about this problem, we need to keep track of 45 connections and 10 parts, in total 55 things. If, however, we can arrange the parts in a sequence, and each part would only be related to the adjacent part, then there would be only 9 communication channels. In total, 19 things to keep track of. The difference between 45 and 19 is significant.

But even this isn't our biggest difficulty. We may face a bigger challenge when trying to reason about the non-linearised problem. Studies show that we can only keep track of 7 things at once (plus/minus 2) [Miller56]. Thus, it becomes harder for our minds to reason when the number of elements grows over this threshold.

If the 10 parts of the problem are linearised, then if one fully wants to reason about a part, they need to consider that part and the 2 relations it might have with the adjacent parts. That is, one needs to keep track of 3 things.

But, if all the parts are connected to all the other parts, one can't easily reason about any single part. This is because they must keep track of 11 different things at the same time.

Linearisation is not always possible, but maybe we can group the parts, and reduce the cognitive load for reasoning about these parts. But, as always, we must have better reasoning strategies for breaking up systems formed from multiple parts into smaller systems. While there are great advancements in this area, I feel that we still need more thought put into how to organise the parts of our systems.

Act 13: In search of the silver bullet

After exploring essential complexity and accidental complexity, let's quickly try to address Brooks' question: can we find a *silver bullet* that would increase our productivity by an order of magnitude?

Brooks put this problem in terms of difficulties. But, let's reduce this question to complexity. That would be: can we find a way to reduce the complexity of our code by a factor of 10?

The reader should note that the difficulty of working in the code may not be directly proportional to our complexity measure. But, in the lack of a better measure, we can assume that the difficulty is linear with the complexity number. That is, with our assumption, a code with twice the value for complexity will be twice as difficult to work with.

We start from the idea that the complexity of some code cannot be smaller than the complexity of the problem we are trying to solve. We always have some accidental complexity. Mathematically, we have $C(solution) = C(problem) + C(accidental)$.

To have a ten-fold decrease, we need to have $C(solution) > 10C(problem)$ or $C(accidental) > 9C(problem)$.

we need to be prepared to have **more and more tools** at our disposal **to fight complexity**, whether it's **essential or accidental**

Moreover, we should be able to reduce the accidental complexity by that much.

In our sorting example, the ratio between the complexity of the solution and the complexity of the problem was only 4.4. That is, for this problem, we cannot get a 10-fold improvement even if we could magically remove all the accidental complexity.

I would argue that, for most problems, we would get similar ratios. That is, the complexity of the solution isn't 10 times bigger than the complexity of the problem.

On the other hand, there are so many code bases with heavy piles of technical debt. In those cases, one can reduce a lot of accidental complexity and get a 10x improvement on working in that codebase. But, maybe those cases are just exceptions.

As mentioned above, besides the complexity of the code we are writing for the solution, there are also difficulties related to tools and processes that are not captured in the code. Making a stance similar to Brooks, we assume that these difficulties are not significant in the grand scheme of things.[2]

So, with our set of assumptions, we can only confirm Brooks' postulate:

> There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

Act 14: Epilogue

The war is not over, and it probably will never be. It just leaves deep scars on countless people, who willingly or unwillingly take part in the software engineering wars.

Problems will become more and more complex, and thus we need to be prepared to have more and more tools at our disposal to fight complexity, whether it's essential or accidental.

By now, we know what the main challenge is. It's not about the tools, about libraries and frameworks, or following the steps of a specific process. Although all these can help. It's about utilising our brain in a more efficient manner. And, because we cannot rewire our brain, we need to change how we structure all the activities in software engineering to better fit the model of the brain.

In our long discussion we covered three things: reasoning in software engineering (the topic of part one, in the last issue), reasoning on essential complexity, and reasoning on accidental complexity. In part one, we argued that a certain type of philosophical reasoning is fundamental for software engineering, and we set ourselves on a track to explore different reasoning strategies, with the hope that we will have a better grasp on software engineering. In the second part of this article (this issue), we approached essential and accidental complexity. While we were able to provide a framework for reasoning about essential complexity, we soon realised that this framework doesn't directly help that much in practice. We started our exploration of accidental complexity; however, things are far muddier here, as we are dealing with almost all the aspects of software engineering. Instead of providing a semiformal description of accidental complexity, we started to reason on some aspects that seem to be of great importance. While by no means complete, we believe that the discussion covers some important aspects of accidental complexity.

After our analysis, we tried to have an answer for whether there can be a *silver bullet* that would reduce the difficulties of programming by a factor of 10. With a series of assumptions, we concluded that this is probably not the case. Even if it appears that accidental complexity constitutes a large part of what we need to solve in software engineering, it doesn't fully cover 9/10 of our projects. And, even if it did, the complexity of reasoning about different parts of the solution is pretty high, so we cannot hope to increase it dramatically. Especially since we can't rewire our brains.

Thus, once again, if we cannot rewire our brains, the only hope we have is to get better at reasoning on different aspects of our solution, and on different aspects of software engineering, and maybe *trick* our brain into being more productive. ■

References

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

[Henney22a] Kevlin Henney, 'The Most Bogus Sort', 2022, https://kevlinhenney.medium.com/the-most-bogus-sort-3879e2e98e67

[Henney22b] Kevlin Henney, 'Why It Can Be Hard to Test', 2022, https://www.youtube.com/watch?v=aqkvapSSrKg

[Miller56] George A. Miller, 'The magical number seven, plus or minus two: Some limits on our capacity for processing information', *Psychological Review*. 63 (2), 1956, http://psychclassics.yorku.ca/Miller/

[Seemann19] Mark Seemann, 'Yes silver bullet', 2019, https://blog.ploeh.dk/2019/07/01/yes-silver-bullet/

[Teodorescu21] Lucian Radu Teodorescu, 'How We (Don't) Reason About Code', *Overload* 163, June 2021, https://accu.org/journals/overload/29/163/overload163.pdf#page=13

[Teodorescu22] Lucian Radu Teodorescu, 'Performance Considered Essential', *Overload* 169, June 2022, https://accu.org/journals/overload/30/169/overload169.pdf#page=6

2 This is different to the point that Mark Seemann argues in his 'Yes silver bullet' article [Seemann19].

Passkey Idiom: A Useful Empty Class

How do you share some but not all of a class?
Arne Mertz introduces the passkey idiom
to avoid exposing too much with friendship.

Let's have a look at an example for useful empty classes. The passkey idiom can help us regain the control that we give up by simply making classes **`friend`**s.

The problem with friendship

Friendship is the strongest coupling we can express in C++, even stronger than inheritance. So, we'd better be careful and avoid it if possible. But sometimes we just can't get around giving one class more access than another.

A common example is a class that has to be created by a factory. The factory needs access to the class's constructors. Other classes should not have that access so as not to circumvent the bookkeeping or whatever else makes the factory necessary.

The problem with the **`friend`** keyword is that it gives access to everything. There is no way to tell the compiler that the factory should not have access to any other private elements except the constructor. It's all or nothing. See Listing 1.

```
class Secret {
friend class SecretFactory;
private:
  //Factory needs access:
  explicit Secret(std::string str)
    : data(std::move(str)) {}
  //Factory should not have access but has:
  void addData(std::string const& moreData);
private:
  //Factory DEFINITELY should not have access
  //but has:
  std::string data;
};
```

Listing 1

Whenever we make a class a **`friend`**, we give it unrestricted access. We even relinquish the control of our class invariants, because the **`friend`** can now mess with our internals as it pleases.

The passkey idiom

There is a way to restrict that access. As so often is the case, another indirection can solve the problem. Instead of directly giving the factory access to everything, we can give it access to a specified set of methods, provided it can create a little key token. See Listing 2.

```
class Secret {
  class ConstructorKey {
    friend class SecretFactory;
  private:
    ConstructorKey() {};
    ConstructorKey(ConstructorKey const&)
      = default;
  };
public:
  //Whoever can provide a key has access:
  explicit Secret(std::string str,
  ConstructorKey) : data(std::move(str)) {}

private:
  //these stay private, since Secret itself has
  // no friends any more
  void addData(std::string const& moreData);

  std::string data;
};

class SecretFactory {
public:
  Secret getSecret(std::string str) {
    return Secret{std::move(str), {}};
    //OK, SecretFactory can access
  }

  // void modify(Secret& secret,
  // std::string const& additionalData) {
  //   secret.addData(additionalData);   //ERROR:
  //       // void Secret::addData(const string&)
  //                                 // is private
  // }
};

int main() {
  Secret s{"foo?", {}};    //ERROR:
  // Secret::ConstructorKey::ConstructorKey()
  // is private

  SecretFactory sf;
  Secret s = sf.getSecret("moo!"); //OK
}
```

Listing 2

A few notes

There are variants to this idiom: The key class need not be a private member of **`Secret`** here. It can well be a public member or a free class on its own. That way the same key class could be used as key for multiple classes.

A thing to keep in mind is to make both constructors of the key class private, even if the key class is a private member of **`Secret`**. The default constructor needs to be private and actually defined, i.e. not defaulted, because sadly even though the key class itself and the defaulted constructor are not accessible, it can be created via uniform initialization [Mertz15] if it has no data members .

```
//...
  ConstructorKey() = default;
//...
Secret s("foo?" , {}); //Secret::ConstructorKey
// is not mentioned, so we don't access a
// private name or what?
```

Arne Mertz has been working with modern and not-so-modern C++ codebases for over 15 years in embedded and enterprise contexts. He is a mentor and teacher for clean code and modern C++ for colleagues and customers at Zühlke Engineering.

There was a small discussion about that in the 'cpplang' Slack channel [Slack] a while ago. The reason is that uniform initialization, in this case, will call aggregate initialization which does not care about the defaulted constructor as long as the type has no data members. It seems to be a loophole in the standard causing this unexpected behaviour.

The copy constructor needs to be private especially if the class is not a private member of `Secret`. Otherwise, this little hack could give us access too easily:

```
ConstructorKey* pk = nullptr;
Secret s("bar!", *pk);
```

While dereferencing an uninitialized or null pointer is undefined behaviour, it will work in all major compilers, maybe triggering a few warnings. Making the copy constructor private closes that hole, so it is syntactically impossible to create a `ConstructorKey` object.

Conclusion

While it is probably not needed often, small tricks like this one can help us to make our programs more robust against mistakes. ■

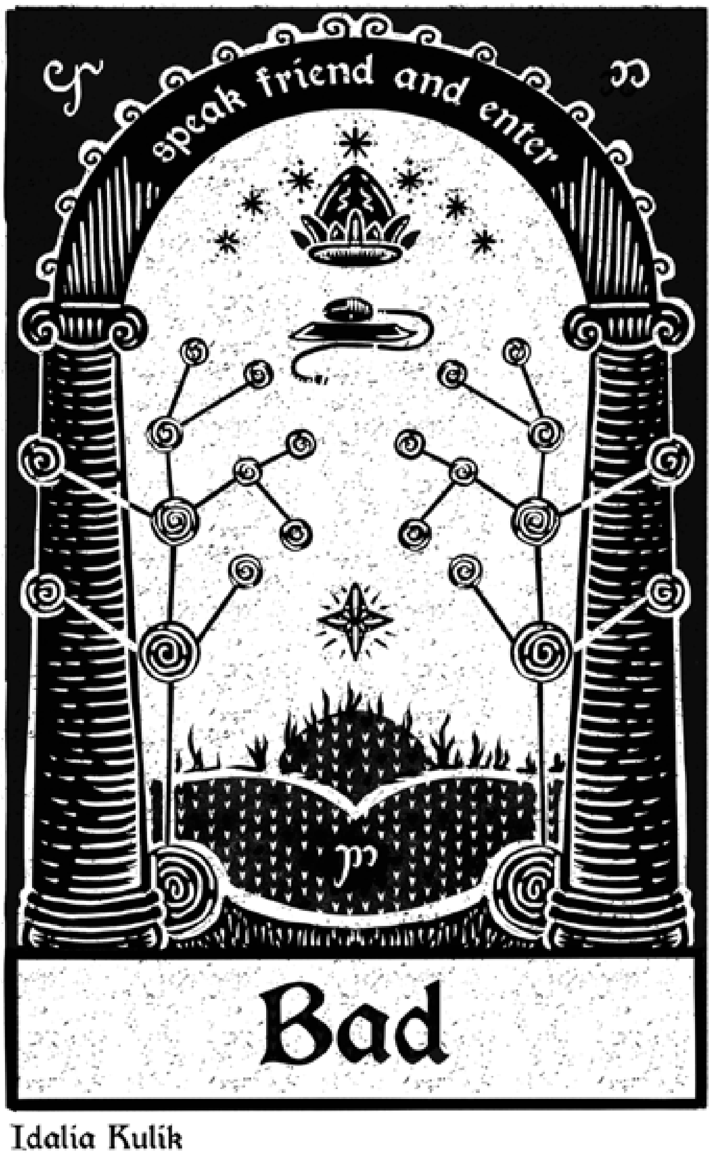
References

[Mertz15] Arne Mertz 'Modern C++ Features – Uniform Initialization and initializer_list', posted 5 July 2015 at: https://arne-mertz.de/2015/07/new-c-features-uniform-initialization-and-initializer_list/

[Slack] Cpplang discussion: https://cpplang.slack.com/

This article was first published on Arne's blog – Simplify C++! – on 19 October 2016 at https://arne-mertz.de/2016/10/passkey-idiom/

Illustration by Idalia Kulik.



C++20 Dynamic Allocations at Compile-time

People often say constexpr all the things. Andreas Fertig shows where we can use dynamic memory at compile time.

You may already have heard and seen that C++20 brings the ability to allocate dynamic memory at compile-time. This leads to **`std::vector`** and **`std::string`** being fully **`constexpr`** in C++20. In this article, I like to give you a solid idea of where you can use that.

How does dynamic allocation at compile-time work?

First, let's ensure that we all understand how dynamic allocations at compile-time work. In the early draft of the paper 'Standard containers and constexpr' [P0784R1], proposed so-called *non-transient* allocations. They would have allowed us to allocate memory at compile-time and keep it to run-time. The previously allocated memory would then be promoted to static storage. However, various concerns did lead to allowing only *transient* allocations. That means what happens at compile-time stays at compile-time. Or in other words, the dynamic memory we allocate at compile-time must be deallocated at compile-time. This restriction makes a lot of the appealing use-cases impossible. I personally think that there are many examples out there that are of only little to no benefit.

The advantages of constexpr

I like to take a few sentences to explain what are the advantages of **`constexpr`**.

First, computation at compile-time does increase my local build-time. That is a pain, but it speeds up the application for my customers – a very valuable benefit. In the case where a **`constexpr`** function is evaluated only at compile-time, I get a smaller binary footprint. That leads to more potential features in an application. I'm doing a lot of stuff in an embedded environment which is usually a bit more constrained than a PC application, so the size benefit does not apply to everyone.

Second, **`constexpr`** functions, which are executed at compile-time, follow the perfect abstract machine. The benefit here is that the compiler tells me about undefined behavior in the compile-time path of a constexpr function. It is important to understand that the compiler only inspects the path taken if the function is evaluated in a **`constexpr`** context. Here is an example to illustrate what I mean.

```
constexpr auto div(int a, int b)
{
  return a / b;
}

constexpr auto x = div(4, 2); ❶
auto           y = div(4, 0);  ❷
// constexpr auto z = div(4, 0); ❸
```

```
<source>:8:16: error: constexpr variable 'z' must
be initialized by a constant expression
constexpr auto z = div(4, 0);
               ^   ~~~~~~~~~
<source>:3:14: note: division by zero
    return a / b;
             ^
<source>:8:20: note: in call to 'div(4, 0)'
constexpr auto z = div(4, 0);
                   ^
1 error generated.
Compiler returned: 1
```

Figure 1

This simple function **`div`** is marked **`constexpr`**. Subsequently, **`div`** is used to initialize three variables. In ❶, the result of the call to **`div`** is assigned to a **`constexpr`** variable. This leads to **`div`** being evaluated at compile time. The values are 4 and 2. The next two calls to **`div`** divide four by zero. As we all know, only Chuck Norris can divide by zero. Now, ❷ assigns the result to a non-**`constexpr`** variable. Hence **`div`** is executed at run-time. In this case, the compiler does not check for the division by zero despite the fact that the function **`div`** is **`constexpr`**. This changes as soon as we assign the call to **`div`** to a **`constexpr`** variable, as done in ❸. Because **`div`** gets evaluated at compile-time now, and the error is on the **`constexpr`** path, the compilation is terminated with an error like that shown in Figure 1.

Aside from not making it, catching such an error right away is the best thing that can happen.

Dynamic allocations at compile-time

As I stated initially, I think many examples of dynamic allocations at compile-time are with little real-world impact. A lot of the examples look like this:

```
constexpr auto sum(const vector<int>& v)
{
  int ret{};
  for(auto i : v) { ret += i; }
  return ret;
}
constexpr auto s = sum({5, 7, 9});
```

Yes, I think there is a benefit to having **`sum constexpr`**. But whether this requires a container with dynamic size or if a variadic template would have been the better choice is often unclear to me. I tend to pick the template solution in favor of reducing the memory allocations.

The main issue I see is that, most often, the dynamically allocated memory must go out of the function. Because this is impossible, it boils down to either summing something up and returning only that value or falling back to, say **`std:array`**.

So, where do I think dynamic allocations at compile-time come in handy and are usable in real-world code?

Andreas Fertig is a trainer and lecturer on C++11 to C++20, who presents at international conferences. Involved in the C++ standardization committee, he has published articles (for example, in *iX*) and several textbooks, most recently *Programming with C++20*. His tool – C++ Insights (https://cppinsights.io) – enables people to look behind the scenes of C++, and better understand constructs. He can be reached at contact@andreasfertig.com

while the code may look nice, the **functions are unusable at compile-time** due to the **restriction of allocations** at compile-time

A practical example of dynamic allocations at compile-time for every C++ developer

All right, huge promise in this heading, but I believe it is true.

Here is my example. Say we have an application with a function `GetHome` that returns the current user's home directory. Another function `GetDocumentsDir`, returns, as the name implies, the documents folder within the user's home directory. In code, this can look like this:

```
string GetHome()
{
  return getenv("HOME"); // assume /home/cpp
}

string GetDocumentsDir()
{
  auto home = GetHome();
  home += "/Documents";
  return home;
}
```

Not rocket science, I know. The only hurdle is that the compiler figures out that `getenv` is never `constexpr`.

For now, let's just use `std::is_constant_evaluated` and return an empty string.

What both functions return is a `std::string`.

Now that we have a `constexpr std::string`, we can make these two functions `constexpr`, as shown next.

```
constexpr string GetHome()
{
  if(std::is_constant_evaluated()) {
    return {}; // What to do here?
  } else {
    return getenv("HOME");
  }
}

constexpr string GetDocumentsDir()
{
  auto home = GetHome();
  home += "/Documents";
  return home;
}
```

The issue is that while the code may look nice, the functions are unusable at compile-time due to the restriction of allocations at compile-time. They both return a `std::string` which contains the result we are interested in. But it must be freed before we leave compile-time. Yet, the user's home directory is a dynamic thing that is 100% run-time dependent. So absolutely no win here, right?

Well, yes. For your normal program, compile-time allocations do nothing good here. So time to shift our focus to the non-normal program part, which is testing. Because the dynamic home directory makes tests environment-dependent, we change `GetHome` slightly to return a fixed home directory if `TEST` is defined. The code then looks like Listing 1.

```
constexpr string GetHome()
{
#ifdef TEST
  return "/home/cpp";
#else
  if(std::is_constant_evaluated()) {
    return {};  // What to do here?
  } else {
    return getenv("HOME");
  }
#endif
}
constexpr string GetDocumentsDir()
{
  auto home = GetHome();
  home += "/Documents";
  return home;
}
```

Listing 1

Say we like to write a basic test checking that the result matches our expectations. I use Catch2 here [Catch2]:

```
TEST_CASE("Documents Directory")
{
  CHECK(GetDocumentsDir()
    == "/home/cpp/Documents");
}
```

Still no use at compile-time of `GetDocumentsDir` or `GetHome`. Why not? If we look closely, we now have everything in place. Due to the defined test environment, `GetHome` no longer depends on `getenv`. For our test case above, we are not really interested in having the string available at run-time. We mostly care about the result of the comparison in `CHECK`.

How you approach this is now a matter of taste.

A neat trick with consteval

Among the various improvements of C++20 are changes to `constexpr`, namely a new keyword `consteval`. In this part of the article, I want to dig into `consteval` a bit and see what we can do with this new facility.

What consteval does

As the name of the keyword tries to imply, it forces a constant evaluation. In the standard, a function that is marked as `consteval` is called an immediate function. The keyword can be applied only to functions. *Immediate* here means that the function is evaluated at the front-end, yielding only a value, which the back-end uses. Such a function never goes into your binary. A `consteval`-function must be evaluated at compile-time or compilation fails. With that, a `consteval`-function is a stronger version of `constexpr`-functions. We have now a choice:

- Compile-time only (`consteval`)
- Compile- or -run-time (`constexpr`)
- Run-time (no attribution required)

consteval is handy in a situation where you want to ensure that a certain function is always evaluated at compile-time

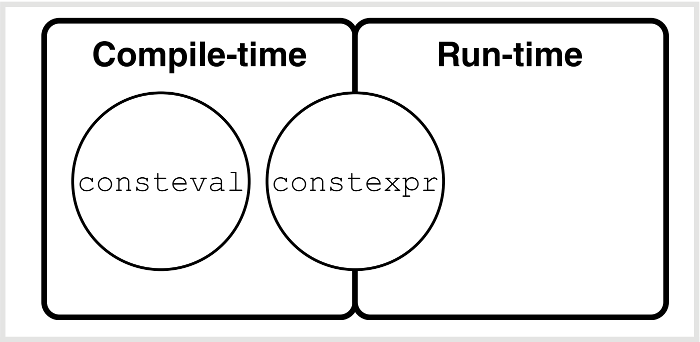


Figure 2

Figure 2 visualizes the three different variants.

The behavior of `consteval` is handy in a situation where you want to ensure that a certain function is *always* evaluated at compile-time.

We already have constexpr

Now, let's circle back and see what we can do with `constexpr` and where things get complicated.

A typical pattern I see in my training classes is the following:

```
constexpr int Calc(int x)
{ ❶
  return 4 * x;
}
int main()
{
  auto res = Calc(2); ❷
}
```

In ❶, we have a `constexpr`-function, so far so good. Then in ❷, this function gets called, and the result is stored in `res`. The natural expectation is that `Calc` is evaluated at compile-time. All criteria are met:

- The function is marked as `constexpr`;
- All input values are constants.

However, `Calc` is evaluated at run-time. Depending on your optimizer and optimization level, things may be different, but `Calc` is called at run-time from a standards point. What is missing is making the variable `res` itself `constexpr`:

```
constexpr int Calc(int x)
{
  return 4 * x;
}
int main()
{
  constexpr auto res = Calc(2); ❸
}
```

In this version, we achieved what we wanted. `Calc` is called at compile-time because the variable itself is marked as `constexpr` (❸). While in a lot of situations, this is okay, there is one where this pattern doesn't work. You may already know this. Marking a variable as `constexpr` also makes this variable implicitly `const`. If you struggle here, use *C++ Insights* to show you what `constexpr` brings piggyback.

Now, assume that we like to have that call to `Calc` happen at compile-time, but `res` should be writable at run-time. This is where we can use `consteval`, to force evaluation at compile-time, regardless of the `constexpr`'ness of the variable:

```
consteval int Calc(int x)
{ // consteval now
  return 4 * x;
}
int main()
{
  auto res = Calc(2); // Compile-time due to
                      // consteval
  ++res;              // Modify res at run-time
}
```

Your new friend: `as_constant`

All right, so far, so good. In the version above `Calc` is now a compile-time only function. Now, what if we like to have both? `Calc` should be usable at compile- and run-time. But at the same time we would like `res` to be writable at run-time? Let me introduce you to `as_constant`, a handy new helper (you have to copy or write yourself):

```
consteval auto as_constant(auto value)
{
  return value;
}
```

Yes, `as_constant` appears to be a very silly function. The function simply returns its input without any modification. I would probably make you remove such a silly function in a code review. But thanks to the consteval modifier, `as_constant` serves a greater purpose:

```
constexpr int Calc(int x)
{ // constexpr again ❹
  return 4 * x;
}
int main()
{
  // Forcing compile-time with as_constant ❺
  auto res = as_constant(Calc(2));
  ++res; // Modify res at run-time ❻
  res = Calc(res); // Run-time use of Calc ❼
}
```

In ❹, `Calc` is `constexpr` again. We use `as_constant` in ❺ to force compile-time evaluation of `Calc`. As before, we can modify `res` in ❻, but we can now also use `Calc` at run-time as ❼ shows. This is something you cannot achieve with another new compile-time keyword in C++20, `constinit`, as `constinit` works only with static initialized data.

Since `as_constant` is evaluated purely at compile-time, the by-value semantic is okay. No need to care about moving things.

One thing is left to mention, with the approach shown with `as_constant` the destructor of the type used in the function must be `constexpr`.

With the large RAM we have today, memory leaks are hard to test at run-time, but not so in a constexpr context...the compiler is our friend

Using as_constant

If you want to use **`as_constant`** in the check for the home directory, the test would look like this:

```
TEST_CASE("Documents Directory constexpr")
{
  CHECK(as_constant(GetDocumentsDir()
    == "/home/cpp/Documents"));
}
```

I probably would soon start defining something like **`DCHECK`** for dual execution and encapsulate the **`as_constant`** call there. This macro then executes the test at compile and run-time. That way, I ensure to get the best out of my test.

```
#define DCHECK(expr)                            \
  CHECK(as_constant(expr));                     \
  CHECK(expr)

TEST_CASE("Documents Directory dual")
{
  DCHECK(GetDocumentsDir()
    == "/home/cpp/Documents");
}
```

In an even better world, I would detect whether a function is evaluable at compile-time and then simply add this step of checking in **`CHECK`**. However, the pity here is that such a check must check whether the function is marked as **`constexpr`** or **`consteval`** but not execute it because once such a function contains UB, the check would fail.

But let's step back. What happens here, and why does it work?

`as_constant` enforces a compile-time evaluation of what it gets called with. In our case, we create two temporary **`std::string`**s, which are compared, and the result of this comparison is the parameter value of **`as_constant`**. The interesting part here is that temporaries in a compile-time-context are compile-time. We forced the comparison of **`GetDocumentsDir`** with the expected string to happen at compile-time. We then only promote the boolean value back into run-time.

The huge win you get with that approach is that in this test at compile-time, the compiler will warn you about undefined behavior:

- like an of-by-one error (which happened to me while I implemented my own **`constexpr`** string for the purpose of this article);
- memory leaks because not all memory gets deallocated;
- comparisons of pointers of different arrays;
- and more...

With the large RAM we have today, memory leaks are hard to test at run-time, but not so in a **`constexpr`** context. As I said so often, the compiler is our friend. Maybe our best friend when it comes to programming.

Of course, there are other ways. You can make the same comparison as part of a **`static_assert`**. The main difference I see is that the test will fail early, leading to a step-by-step failure discovery. Sometimes it is nicer to see all failing tests at once.

Another way is to assign the comparison result to a **`constexpr`** variable that saves you from introducing the helper function **`as_constant`**.

I hope you agree with my initial promise; the example I showed you is something every programmer can adapt.

Recap

Sometimes it helps to think out of the box a bit. Even with the restrictions of compile-time allocations, there are ways where we can profit from the new abilities.

- Make functions that use dynamic memory **`constexpr`**.
- Look at which data is already available statically.
- Check whether the result, like the comparison above, is enough, and the dynamic memory can happily be deallocated at compile-time.

Your advantages are:

- Use the same code for compile and run-time;
- Catch bugs for free with the compile-time evaluation;
- The result can stay in the compile-time context in more complex cases because it is more like in the initial example with **`sum`**.
- Over time, maybe we will get non-transient allocations. Then your code is already ready.

I hope you have learned something today. If you have other techniques or feedback, please contact me. ■

References

[Catch2] A modern, C++-native, test framework for unit-tests, TDD and BDD: https://github.com/catchorg/Catch2

[P0784R1] Standard containers and constexpr: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0784r1.html

This article was published as two posts on Andreas Fertig's blog:

- 'C++20: A neat trick with consteval' (posted 6 July 2021) available from: https://andreasfertig.blog/2021/07/cpp20-a-neat-trick-with-consteval/
- 'C++20: Dynamic Allocations at Compile-time' (posted 3 August 2021) available from: https://andreasfertig.blog/2021/08/cpp20-dynamic-allocations-at-compile-time/

Afterwood

Open source code has a long history. Chris Oldwood tells us how he discovered open source and got his first role as a software maintainer.

A tweet recently appeared in my timeline that caused me to go all Obi-Wan Kenobi and exclaim “now there’s a name I’ve not heard in a very long time”. The name was Phil Karn, although his twitter handle of KA9Q might ring a few bells to those in the amateur (ham) radio scene. But that’s not where I know his name from, at least, not directly.

My first professional programming gig was with a small software house (GST) in the UK back in the early ’90s. This was an era where we weren’t all permanently connected to the Internet. Although small, the company had a Novell NetWare network on which we could send internal email using the Pegasus Mail (aka pmail) DOS based email client which had special support for Novell NetWare. However, it didn’t have any native support for sending email over the Internet because there was no formal TCP/IP support in DOS.

Enter stage left: Phil Karn.

Back in the mid ’80s Phil wrote a TCP/IP application called KA9Q [KA9Q] (the name being based on his ham radio callsign) which he later ported to MS-DOS. This application allowed a PC to connect to the Internet via a modem and came bundled with a number of popular clients such as Telnet, FTP, and, more importantly for this story, SMTP. (At the time KA9Q was referred to as a NOS – Network Operating System – because networking wasn’t a ubiquitous part of an OS like it is today.) Another key feature of this application was that the source was freely available, so you could add support for additional hardware, fix bugs, etc.

Although the Novell NetWare sysadmin at GST was also an amateur radio fan, apparently it was another employee (John Bradley) who worked out that if they could fork KA9Q and tweak the SMTP server code to work with PMail / NetWare it would allow the company to send and receive mail externally, as well as internally. And thus was born “nonet” – a [NO] vell fork of the KA9Q net program that delivered incoming mail directly to a NetWare user’s inbox.

Of course, it wasn’t quite that simple as a direct Internet connection cost a small fortune, but luckily Demon Internet [Wikipedia] had started its ‘tenner-a-month’ offering (£10 + VAT being the subscription price) in the UK which allowed mere mortals and small companies to ‘get on the internet’ at a more affordable price. You still had to pay telephone call charges which meant you couldn’t simply leave your modem permanently online, but using a classic scheduler like cron allowed you to regularly dial-up, exchange emails, and then disconnect – a process affectionately known as a ‘blink’. (Although the modem handshake alone took way longer than the blink of an eye, let alone the actual exchange of emails!)

This all happened before I even joined GST so you’re probably wondering where this somewhat obscure history lesson is going…

Eventually the existing maintainer of Nonet (John Bradley) left the company, and somebody needed to take over the reins because either a change in PMail or NetWare (I’m hazy on the details 30 years later) was causing a problem. Actually, I had already started taking an interest in networking and KA9Q because I realised I could use its FTP client to download this up-and-coming new UNIX-like OS for PCs (and the Atari TT) called ‘Linux’. The company was also spending more time connected to the Internet due to the rise in Internet email and despite my best efforts to reconfigure the concurrency of the built-in SMTP server, a bug when handling bounced emails meant it was too unreliable and I had to revert it back to one.

With a genuine need to fix a couple of problems affecting the company, and my newfound skills in the C programming language, I rolled up my sleeves and offered to dive in and fix things. Except this was not just a simple C console application, it was like nothing I had seen before. To concurrently handle sending and receiving TCP/IP traffic while also processing screen and keyboard I/O on an OS with no built-in multi-tasking capabilities meant Phil had to implement his own form of co-operative multi-tasking which he did using coroutines and some clever hacks. This codebase wasn’t something you could easily attach a debugger to and single-step around the code and I wasn’t nearly clever enough to understand how it all worked, so I resorted to printf style debugging and a fast edit, compile, test loop. In the end I fixed the most pressing issue but the “bouncing email” hang continued to elude me.

With the fix in place and working nicely I realised there were other companies out there also relying on this free tool and so I had to put together a “formal” release (source and binary), upload it to various FTP servers, CiX, etc. and announce it. I also realised that I needed to update the support details in the README and become the point of contact for NOnet going forward, at least, until I also left the company a couple of years later. And so this was my introduction to becoming a maintainer of (a tiny fork) of an open source project.

Over the following 30 years I’ve written and released more than 30 free tools of my own, all with source code freely available, and with at least basic documentation, installer, etc. and continued to support them when I can (sometimes on company time, when they have benefitted from them, but mostly on my daily commute by train).

I only realised later that I had automatically given this stuff away, although I seriously doubt there is any value in any of it anyway. I now have genuine admiration too for those people that do start a company and turn their software projects into a saleable product. Hence, I attribute at least some part of my (unconscious) decision to adopt an (informal) open source model for my own tools to Phil Karn and KA9Q because it seemed like the right thing to do. My professional programming career has allowed me to stand on the shoulders of giants and I’m glad that I have been able to use my own position of privilege to contribute something (no matter how small) back to the software community. ■

References

[KA9Q] http://www.ka9q.net/code/ka9qnos

[Wikipedia] https://en.wikipedia.org/wiki/Demon_Internet

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

accu

professionalism in programming

Monthly journals, available printed and online

Discounted rate for the ACCU Conference

Email discussion lists

Technical book reviews

Local groups run by ACCU members

Visit www.ACCU.org to find out more

To connect with
like-minded people
visit accu.org

accu