overload 172

DECEMBER 2022 £4.50

Compile-Time Strings

Wu Yongwei summarises his experience of using compile-time strings.

The Year of C++ Successor Languages

Lucian Radu Teodorescu reports on the languages created to rival C++.

An Introduction to Go for C++ Programmers

Arun Saha walks us through Go from the perspective of a C++ programmer.

The Model Student: The Regular Travelling Salesman – Part 2

A reprint of the second article in the series from Richard Harris investigating modelling problems on a computer.

The Testing Iceberg

Seb Rose explains when we should invest effort in making a test readable to non-technical people.

Afterwood

Chris Oldwood brings us some seasonal cheer.

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

December 2022
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

Cover photo by Elliot Wilkinson on Unsplash.

Copy deadlines

All articles intended for publication in *Overload* 173 should be submitted by 1st January 2023 and those for *Overload* 174 by 1st March 2023.

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

4 **Compile-Time Strings**
Wu Yongwei summarises his experience of using compile-time strings.

8 **The Year of C++ Successor Languages**
Lucian Radu Teodorescu reports on the languages created to rival C++.

15 **An Introduction to Go for C++ Programmers**
Arun Saha walks us through Go as a C++ programmer.

21 **The Testing Iceberg**
Seb Rose introduces the Testing Iceberg to explain when we should invest effort in making a test readable to non-technical people.

22 **The Model Student: The Regular Travelling Salesman – Part Two**
Richard Harris explores more of the mathematics of modelling problems with computers.

28 **Afterwood**
Chris Oldwood git-pull's a cracker, bringing us some seasonal cheer.

The ACCU

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

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

Copyrights and Trademarks

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

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

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

Don’t Believe the Hype

Attention grabbing announcements can usually be safely ignored. Frances Buontempo considers how to pick her way through the hyperbole.

Yet again the chaotic news from the UK has distracted me and I therefore haven’t even begun thinking about an editorial. The rolling news frequently claims, ‘Breaking news: Announcement expected soon.’ People then talk for ages filling time until ‘Something Happens’. The big bold lettering claiming there is breaking news is certainly attention grabbing. Newspapers also try to draw our eyes with cleverly worded headlines. Programming articles and talks also have an honourable history of catchy titles. ‘How I Learned To Stop Worrying and Love X’, ‘X considered harmful’, ‘What is X and why do I care?’, and the like.

Often these seem very formulaic so it should be simple to get AI to generate them. I say AI, but I jest. Picking a verb or noun at random from a list to fill in some blanks would work. The internet seems to be littered with ‘awesome’ (or other over-the-top word) headline generators. I tried one for my ACCU conference proposal and the suggestions were varied. ‘How to Use Random to Understanding’, ‘10 Steps to a Successful Random’or ‘What [Current Popular TV Show] Can Teach You About Random’. Maybe you don’t fall for cynical marketing or other such distractions, but some people do. We could dig into why conspiracy theories work, but that is outside my area of expertise. When we lived in London, we spent an amount of time talking to Dr Gordon Wright, a lecturer in Psychology and researcher at Goldsmiths, University of London, about conspiracy theories and why people take them on board. You can follow up by reading a few of his publications [Wright] if you want. This is a broad topic, and Gordon understands it far better than I do. There are many reasons conspiracies gain ground, but sometimes feeling like you have realized something few other people know becomes a feedback loop. The more people tell you that you are wrong ‘proves’ your point. An easy trap to fall into and a hard one to escape. Sometimes I convince myself I know where a bug is hiding or the root of a performance issue and would waste hours if someone doesn’t stop me. Of course, this differs from believing a conspiracy theory, because I can be persuaded around relatively quickly. Likewise, most of us can see through the hyped-up headlines. Listening to both sides, trying to find evidence, and avoiding confirmation bias all help.

Sometimes out and out lies or ‘spun’ headlines aren’t the problem. Some of us are distracted by shiny new things. For a long time, we have seen various languages touted as the successor to C++. Go was introduced by Google a while ago, with version 1.0 released in 2012. This issue has an introduction to the language if you’ve not tried it before. I recall being told Go is safer because it uses garbage collection. Many other languages do as well, and some would suggest that deterministic destruction can have its advantages. Some claim Go compiles quicker too [Golang]. It comes with inbuilt concurrency options too, having been specifically designed for networking and multiprocessing. Elements of concurrency are now part of C++ though. I couldn’t possibly say if one is better than the other. It probably depends on how you define ‘better’. Then came Rust. I am told Rust emphasizes performance, type safety, and concurrency and enforces memory safety. Many people do seem to be enjoying using it. Carbon is another language started at Google and explicitly touted as “an experimental successor to C++” [Carbon-1]. It claims to have “Safer fundamentals, and an incremental path towards a memory-safe subset.” There are various other successor languages too, including Cpp2; see Lucian’s article in this edition of *Overload*.

C++ was not introduced by a company. It is an ISO language, so agreement is required to introduce new features or make changes. It also tries to keep backwards compatibility, though will sometimes make breaking changes, and this includes elements inherited from C, though C is also evolving. Wikipedia notes that C++ began as an early fork of pre-standardised C++ [Wikipedia-1]. Bjarne Stroustrup has written about C and C++ interoperability [Stroustrup02]. This paper investigated how the future evolution of C and C++ can best serve that community. The paper is now over twenty years old, but still contains many sensible and relevant ideas. The second section is entitled ‘Red herrings’ and he nails the reasons statements “confound and inflame debates” about C and C++, but I believe these apply to more recently statements about C++ versus ShinyNewLanguage. He talks about mischaracterisations deflecting away from more salient matters. For example, “I don’t like OO so C is better than C++.” It’s very hard to decide which language is better suited for a task, and a company deciding to use Carbon, for example, will have trouble finding people with five plus years’ experience for the language. When Go first came out, I did see recruitment agents asking for several years’ experience in Go. You couldn’t make this stuff up! However, that’s a recruitment agent problem, rather than a language war issue. Finally, Bjarne also points out “Often, a language is chosen for a project based on little knowledge of the future task, mostly on a couple of programmers’ previous experience, and on what happens to be available.” Even if there were a perfect language for a task and you knew all your future requirements, if you can’t get the staff, you will either need a training budget, or have to make do with an ‘inferior’ language. And I suspect no language is perfect. Perhaps I should invent a language called Perfect, if no one has beaten me to it. We can be sure it will be Perfect in name only. Don’t believe the hype.

New rivals to C++ frequently point out the legacy that C++ needs to support. The committee does tread carefully. Releasing ABI breaking changes is infrequent. Compiler implementers have to tread carefully too. Gcc talks about the complexity of managing different version numbers and options [GNU]. They also talk about ABI checks they use, ending by saying “Perhaps there are other C++ ABI checkers. If so, please notify

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad’s BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

us. We'd like to know about them!" Any new-fangled, upstart language that isn't ISO standardized is free to do whatever it chooses, right? Well, maybe. I had always thought of C# as a Microsoft language, leaving them free to change things at will. This may be partially true, and I have lost track of many newer features since I haven't used the language in anger for a couple or so years. However, C# was open sourced a while ago and the common language infrastructure (CLI) is ISO and ECMA standardised [ISO]. This allows .Net code to run on non-Windows platforms. Having standards might not be a bad thing.

Trying to learn a new language can be difficult at the best of times. For a new language, we have extra challenges. The docs for Carbon say it's "currently an experimental project. There is no working compiler or toolchain." You can try out code on the compiler explorer [Carbon-2], and it will be interesting to watch how this plays out. Back in 2013, I wrote about learning fantasy languages. [Buontempo13] and suggested a new language wouldn't have code you could copy on Stack Overflow (SO) and there wouldn't be any books you could buy to learn from. I can't currently see a cpp2 or Carbon tag on SO and if I search for books, I find ones relating to Mac programming using the Carbon API, which is a different matter. Naming is one of the hardest problems in programming, and programming language names are often really rather difficult to search for on the internet. C, C++, D, r, G; sometimes slapping "lang" on the end helps, but not always.

Many people have a pot shot at C++. It is a frustrating language at times and can be difficult to learn. However, I enjoy coding in C++ and think many of the recent changes have made life better. I am very grateful to the committee members who spend time and money keeping things moving. While thinking about hype, I recalled Russel Winder giving a talk at Canary Wharf in London a long while ago, entitled something like 'C++ is dead'. The talk wasn't recorded, but I did find a slide deck [Winder13] from Russel's lightning talk for the 2013 ACCU conference. His title was 'Who needs C++ when you have D and Go?' He walked through an example calculating the sum of the squares of numbers between 0 and 100 that are divisible by seven. The slides show various approaches in Python, D and Go. He then shows what we used to have to do in C++. His conclusion was "D is the real winner as the functions work out of the box. The Go code requires lots of extra code. Until `std::range` exists it (C++) is the loser." And here we are now, with ranges. I suspect Russel would have been delighted with the introduction of ranges to C++, but then gone on to lambast C++ in other ways. Calling out problems with a language and showing other approaches often leads to incremental improvement. Causing controversy with attention grabbing titles can lead to positive outcomes.

Sometimes attention grabbing is purely gratuitous. Modal dialog boxes materializing just as I am typing being a case in point. Or my PC (personal computer) announcing an immediate reboot is required. We are often told to avoid scams by being wary of anything demanding immediate action. The sense of urgency is purported to produce a slight panic, rendering one incapable of thinking straight. I am not suggesting my PC is trying to scam me, but I do wonder sometimes. It's possible to flag chats or emails as high importance, and I often accidentally find a key combination to do this by mistake. If I see an email marked as being of high importance, I am usually somewhat skeptical.

We are used to red flags indicating high importance or warnings. We use symbols to convey ideas. Stock phrases and headlines or titles use patterns to convey a lot of information in very few words. If we see a title ending in a question mark, we suspect Betteridge's law of headlines applies [Wikipedia-2]. Can any headline that ends in a question mark be answered by the word 'no'? I'm not sure how to think through the self-reference in this question. Betteridge's law suggests the answer is 'no', which proves the law is wrong. This takes us rather close to a Gödel sentence and then we hit the limits of provability in formal systems. You can't have consistency and completeness. (See [Gödel] for more details.) You can't have your cake and eat it.

Now, some stock phrases are culture specific, so forgive me if I have failed to take this into account as I write. Furthermore, some words and phrases fall out of favour. In April 2019, I wrote a piece entitled 'This means war!' [Buontempo19], exploring how careless use of language can upset people. We often use foobar or similar terms borrowed from the military when we write code snippets, without realizing the background to the words. Recently people have been discussing the default branch name of 'master' in version control and moving to a different name. The words 'master' and 'slave' do conjure up much unpleasantness. Whenever this happens, some people will complain about PC (political correctness) gone mad, or more recently about "tofu eating wokerati" [Guardian22]. Perhaps that is somewhat culture specific too, being tied to current affairs in the UK. It's a great phrase though. I, for one, fully embrace tofu.

We use titles, headlines and even variable names, like temp, to indicate more context. If someone says, "Hold my pint," we expect a diatribe or long tale of woe. A friend stayed over recently and we showed him around our house. The previous owners told us many tales about the house. We were told of a plague pit at the end of the garden, a cock fighting pit under the floorboards and many similar tales. We settled into starting each with the phrase "Legend has it", as a useful shorthand.

There is nothing wrong with a spot of controversy or hype. The trick is to pick your way through the attention grabbing silliness and make things better. I'd like to think that, in some small way, Russel contributed to C++'s ranges without realizing it. Let's call out the things we don't like and work on incremental improvement of whichever language we choose to code in.

References

[Buontempo13] Frances Buontempo 'Learning Fantasy Languages' *Overload* 116 August 2013, https://accu.org/journals/overload/21/116/overload116.pdf#page=3

[Buontempo19] Frances Buontempo 'The Means War!' *Overload* 150 https://accu.org/journals/overload/27/150/overload150.pdf#page=4

[Carbon-1] Carbon on github: https://github.com/carbon-language/carbon-lang

[Carbon-2] Compiler Explorer: https://carbon.compiler-explorer.com/

[GNU] 'ABI Policy an Guidelines' in *The GNU C++ Library Manual* https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html

[Gödel] 'Gödel's Incompleteness Theorems' (2013) in *Stanford Encylopedia of Philosophy* https://plato.stanford.edu/entries/goedel-incompleteness/

[Golang] Golang Vs C++: https://mindmajix.com/golang-vs-cpp

[Guardian22] 'Suella Braverman blames 'Guardian-reading, tofu-eating wokerati' for disruptive protests' at https://www.theguardian.com/politics/video/2022/oct/18/suella-braverman-blames-guardian-reading-tofu-eating-wokerati-for-disruptive-protests-video

[ISO] 'ISO/IEC 23271:2012 Information technology – Common Language Infrastructure (CLI)': https://www.iso.org/standard/58046.html

[Stroustrup02] Bjarne Stroustrup (2002) 'C and C++: a Case for Compatibility', *The C/C++ Users Journal*, https://www.stroustrup.com/compat_short.pdf

[Wikipedia-1] Compatibility of C and C++: https://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B

[Wikipedia-2] 'Betteridge's law of headlines': https://en.wikipedia.org/wiki/Betteridge%27s_law_of_headlines

[Winder13] Russel Winder (2013) 'Who Needs C++ When You Have D and Go?" from a Lightening Talk at ACCU Conference 2013 https://www.slideshare.net/Russel_Winder/who-needscwhenyouhaved-andgo

[Wright] Dr Gordon Wright, Goldsmiths, University of London: biography and various articles: https://www.gold.ac.uk/psychology/staff/wright-gordon/

Compile-Time Strings

Compile-time strings have been used in many projects over the years. Wu Yongwei summarises his experience.

std::string is mostly unsuitable for compile-time string manipulations.

There are several reasons:

- Before C++20, one could not use **string**s at all at compile time. In addition, the major compilers didn't start to support compile-time **string**s until quite late. MSVC [MSVC] was the front runner in this regard, GCC [GCC] came second with GCC 12, and Clang [Clang] came last with Clang 15 (released a short while ago).
- With C++20 one can use **string**s at compile time, but there are still a lot of inconveniences, the most obvious being that **string**s generated at compile time cannot be used at run time. Besides, a **string** cannot be declared **constexpr**.
- A **string** cannot be used as a template argument.

So we have to give up this apparent choice, but explore other possibilities. The candidates are:

- const **char** pointer, which is what a string literal naturally decays to
- **string_view**, a powerful tool added by C++17: it has similar member functions to those of **string**, but they are mostly marked as **constexpr**!
- **array**, with which we can generate brand-new strings

We will try these types in the following discussion.

Functions commonly needed

Getting the string length

One of the most basic functions on a string is getting its length. Here we cannot use the C function **strlen**, as it is *not* **constexpr**.

We will try several different ways to implement it.

First, we can implement **strlen** manually, and mark the function **constexpr** (see Listing 1). However, is there an existing mechanism to retrieve the length of a string in the standard library? The answer is a definite *Yes*. The standard library does support getting the length of a string of any of the standard character types, like **char**, **wchar_t**, etc. With the most common character type **char**, we can write:

```
constexpr size_t length(const char* str)
{
  return char_traits<char>::length(str);
}
```

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

```
namespace strtools {
  constexpr size_t length(const char* str)
  {
    size_t count = 0;
    while (*str != '\0') {
      ++str;
      ++count;
    }
    return count;
  }
} // namespace strtools
```

Listing 1

It's been possible to use **char_traits** methods at compile time since C++17. (However, you may encounter problems with older compiler versions, like GCC 8.)

Assuming you can use C++17, **string_view** is definitely worth a try:

```
constexpr size_t length(string_view sv)
{
  return sv.size();
}
```

Regardless of the approach used, now we can use the following code to verify that we can indeed check the length of a string at compile time:

```
static_assert(strtools::length("Hi") == 2);
```

At present, the **string_view** implementation seems the most convenient.

Finding a character

Finding a specific character is also quite often needed. We can't use **strchr**, but again, we can choose from a few different implementations. The code is pretty simple, whether implemented with **char_traits** or with **string_view**.

Here is the version with **char_traits**:

```
constexpr const char*
find(const char* str, char ch)
{
  return char_traits<char>::find(
    str, length(str), ch);
}
```

Here is the version with **string_view**:

```
constexpr string_view::size_type
find(string_view sv, char ch)
{
  return sv.find(ch);
}
```

I am not going to show the manual lookup code this time. (Unless you have to use an old compiler, simpler is better.)

If we want a value to be used as a template argument inside a function, it must be passed to the function template as a template argument.

Comparing strings

The next functions are string comparisons. Here **`string_view`** wins hands down: **`string_view`** supports the standard comparisons directly, and you do not need to write any code.

Getting substrings

It seems that **`string_view`**s are very convenient, and we should use **`string_view`**s wherever possible. However, is **`string_view::substr`** suitable for getting substrings? This is difficult to answer without an actual usage scenario. One real scenario I encountered in projects was that the **`__FILE__`** macro may contain the full path at compile time, resulting in different binaries when compiling under different paths. We wanted to truncate the path completely so that the absolute paths would not show up in binaries.

My tests showed that **`string_view::substr`** could not handle this job. With the following code:

```
puts("/usr/local"sv.substr(5).data());
```

we will see assembly output like the following from the compiler on [Godbolt] (at https://godbolt.org/z/1dssd96vz):

```
.LC0:
      .string "/usr/local"
      …
      mov     edi, OFFSET FLAT:.LC0+5
      call    puts
```

We have to find another way.

Let's try **`array`**. It's easy to think of code like the following:

```
constexpr auto substr(string_view sv,
  size_t offset, size_t count)
{
  array<char, count + 1> result{};
  copy_n(&sv[offset], count, result.data());
  return result;
}
```

The intention of the code should be very clear: generate a brand-new character **`array`** of the requested size and zero it out (**`constexpr`** variables had to be initialized on declaration before C++20); copy what we need; and then return the result. Unfortunately, the code won't compile.

There are two problems in the code:

- Function parameters are not **`constexpr`**, and cannot be used as template arguments.
- **`copy_n`** was not **`constexpr`** before C++20, and cannot be used in compile-time programming.

The second problem is easy to fix: a manual loop will do. We shall focus on the first problem.

A **`constexpr`** function can be evaluated at compile time or at run time, so its function arguments are not treated as compile-time constants, and cannot be used in places where compile-time constants are required, such as template arguments.

Furthermore, this problem still exists with the C++20 **`consteval`** function, where the function is only invoked at compile time. The main issue is that if we allow function parameters to be used as compile-time constants, then we can write a function where its arguments of different *values* (same type) can produce return values of different *types*. For example (currently illegal):

```
consteval auto make_constant(int n)
{
  return integral_constant<int, n>{};
}
```

This is unacceptable in the current type system: we still require that the return values of a function have a unique type. If we want a value to be used as a template argument inside a function, it must be passed to the function *template* as a template argument (rather than as a function argument to a non-template function). In this case, each distinct template argument implies a different template specialization, so the issue of a multiple-return-type function does not occur.

By the way, a standard proposal P1045 [Stone19] tried to solve this problem, but its progress seems stalled. As there are workarounds (to be discussed below), we are still able to achieve the desired effect.

Let's now return to the **`substr`** function and convert the **`count`** parameter into a template parameter. Listing 2 is the result

The code can really work this time. With:

```
puts(substr<5>("/usr/local", 5).data())
```

we no longer see **`"/usr/"`** in the compiler output.

Regretfully, we now see how compilers are challenged with abstractions: With the latest versions of GCC (12.2) and MSVC (19.33) on Godbolt, this version of **`substr`** does not generate the optimal output. There are also some compatibility issues with older compiler versions. So, purely from a practical point of view, I recommend the implementation in Listing 3 (overleaf) that does not use **`string_view`**:

```
template <size_t Count>
constexpr auto substr(string_view sv,
                      size_t offset = 0)
{
  array<char, Count + 1> result{};
  for (size_t i = 0; i < Count; ++i)
  {
    result[i] = sv[offset + i];
  }
  return result;
}
```

Listing 2

```
template <size_t Count>
constexpr auto substr(const char* str,
                      size_t offset = 0)
{
  array<char, Count + 1> result{};
  for (size_t i = 0; i < Count; ++i) {
    result[i] = str[offset + i];
  }
  return result;
}
```

Listing 3

If you are interested, you can compare the assembly outputs of these two different versions of the code:

- https://godbolt.org/z/7nYK97oKr
- https://godbolt.org/z/Ts563oaYj

Only Clang is able to generate the same efficient assembly code with both versions:

```
mov     word ptr [rsp + 4], 108
mov     dword ptr [rsp], 1633906540
mov     rdi, rsp
call    puts
```

If you don't understand why the numbers 108 and 1633906540 are there, let me remind you that the hexadecimal representations of these two numbers are 0x6C and 0x61636F6C, respectively. Check the ASCII table and you should be able to understand.

Since we have stopped using `string_view` in the function parameters, the parameter `offset` has become much less useful. Hence, I will get rid of this parameter, and rename the function to `copy_str` (Listing 4).

Passing arguments at compile time

When you try composing the compile-time functions together, you will find something lacking. For example, if you wanted to remove the first segment of a path automatically (like from `"/usr/local"` to `"local"`), you might try some code like Listing 5.

The problem is still that it won't compile. And did you notice that this code violates exactly the constraint I mentioned above that the return type of a function must be consistent and unique?

I have adopted a solution described by Michael Park [Park17]: using lambda expressions to encapsulate 'compile-time arguments'. I have defined three macros for convenience and readability:

```
template <size_t Count>
constexpr auto copy_str(const char* str)
{
  array<char, Count + 1> result{};
  for (size_t i = 0; i < Count; ++i)
  {
    result[i] = str[i];
  }
  return result;
}
```

Listing 4

```
constexpr auto remove_head(const char* path)
{
  if (*path == '/') {
    ++path;
  }
  auto start = find(path, '/');
  if (start == nullptr) {
    return copy_str<length(path)>(path);
  } else {
    return copy_str<length(start + 1)
      >(start + 1);
  }
}
```

Listing 5

```
#define CARG typename
#define CARG_WRAP(x)  [] { return (x); }
#define CARG_UNWRAP(x)  (x)()
```

`CARG` means 'constexpr argument', a compile-time constant argument. We can now make `make_constant` really work:

```
template <CARG Int>
constexpr auto make_constant(Int cn)
{
  constexpr int n = CARG_UNWRAP(cn);
  return integral_constant<int, n>{};
}
```

And it is easy to verify that it works:

```
auto result = make_constant(CARG_WRAP(2));
static_assert(
 std::is_same_v<integral_constant<int, 2>,
 decltype(result)>);
```

A few explanations follow. In the template parameter, I use `CARG` (instead of `typename`) for code readability: it indicates the intention that the template parameter is essentially a type wrapper for compile-time constants. `Int` is the name of this special type. We will not provide this type when instantiating the function template, but instead let the compiler deduce it.

When calling the 'function' (`make_constant(CARG_WRAP(2))`), we provide a lambda expression (`[] { return (2); }`), which encapsulates the constant we need. When we need to use this parameter, we use `CARG_UNWRAP` (evaluate `[] { return (2); }()`) to get the constant back.

Now we can rewrite the `remove_head` function (Listing 6).

This function is similar in structure to the previous version, but there are many detail changes. In order to pass the result to `copy_str` as a template argument, we have to use `constexpr` all the way along. So we have to give up mutability, and write code in a quite functional style.

Does it really work? Let's put the following statement into the `main` function:

```
puts(strtools::remove_head(
  CARG_WRAP("/usr/local")) .data());
```

And here is the optimized assembly output from GCC on x86-64 (see https://godbolt.org/z/Mv5YanPvq):

```
main:
        sub     rsp, 24
        mov     eax, DWORD PTR .LC0[rip]
        lea     rdi, [rsp+8]
        mov     DWORD PTR [rsp+8], eax
        mov     eax, 108
        mov     WORD PTR [rsp+12], ax
        call    puts
        xor     eax, eax
        add     rsp, 24
        ret
.LC0:
        .byte   108
        .byte   111
        .byte   99
        .byte   97
```

```
template <CARG Str>
constexpr auto remove_head(Str cpath)
{
  constexpr auto path = CARG_UNWRAP(cpath);
  constexpr int skip = (*path == '/') ? 1 : 0;
  constexpr auto pos = path + skip;
  constexpr auto start = find(pos, '/');
  if constexpr (start == nullptr) {
    return copy_str<length(pos)>(pos);
  } else {
    return copy_str<length(start + 1)>(start
                                       + 1);
  }
}
```

Listing 6

As you can see clearly, the compiler will put the ASCII codes for `"local"` on the stack, assign its starting address to the *rdi* register, and then call the `puts` function. There is absolutely no trace of `"/usr/"` in the output. In fact, there is no difference between the output of the `puts` statement above and that of `puts(substr<5>("/usr/local", 5).data())`.

I would like to remind you that it is safe to pass and store the character `array`, but it is not safe to store the pointer obtained from its `data()` method. It is possible to use such a pointer *immediately* in calling other functions (like `puts`, above), as the lifetime of `array` will extend till the current statement finishes execution. However, if you saved this pointer, it would become *dangling* after the current statement, and dereferencing it would then be undefined behaviour.

String template parameters

We have tried turning strings into types (via lambda expressions) for compile-time argument passing, but unlike integers and `integral_constant`s, there is no one-to-one correspondence between the two. This is often inconvenient: for two `integral_constant`s, we can directly use `is_same` to determine whether they are the same; for strings represented as lambda expressions, we cannot do the same – two lambda expressions *always* have different types.

Direct use of string literals as non-type template arguments is not allowed in C++, because strings may appear repeatedly in different translation units, and they do not have proper comparison semantics – comparing two strings is just a comparison of two pointers, which cannot achieve what users generally expect. To use string literals as template arguments, we need to find a way to pass the string as a sequence of characters to the template. We have two methods available:

- The non-standard GNU extension used by GCC and Clang (which can be used prior to C++20)
- The C++20 approach suitable for any conformant compilers (including GCC and Clang)

Let's have a look one by one.

The GNU extension

GCC and Clang have implemented the standard proposal N3599 [Smith13], which allows us to use strings as template arguments. The compiler will expand the string into characters, and the rest is standard C++. Listing 7 is an example.

The definition of the class template is standard C++, so that:

```
compile_time_string<'H', 'i'>
```

is a valid type and, at the same time, by taking the `value` member of this type, we can get `"Hi"`. The GNU extension is the *string literal operator template* – we can now write `"Hi"_cts` to get an object of type `compile_time_string<'H', 'i'>`. The following code will compile with the above definitions:

```
constexpr auto a = "Hi"_cts;
constexpr auto b = "Hi"_cts;
static_assert(
  is_same_v<decltype(a), decltype(b)>);
```

```
template <char... Cs>
struct compile_time_string {
  static constexpr char value[]{Cs..., '\0'};
};

template <typename T, T... Cs>
constexpr compile_time_string<Cs...>
  operator""_cts()
{
  return {};
}
```

Listing 7

The C++20 approach

Though the above method is simple and effective, it failed to reach consensus in the C++ standards committee and did not become part of the standard. However, with C++20, we can use more types in non-type template parameters. In particular, user-defined literal types are amongst them. Listing 8 is an example.

Again, the first class template is not special, but allowing this `compile_time_string` to be used as the type of a non-type template parameter (quite a mouthful ☺), as well as the *string literal operator template*, is a C++20 improvement. We can now write `"Hi"_cts` to generate a `compile_time_string` object. Note, however, that this object is of type `compile_time_string<3>`, so `"Hi"_cts` and `"Ha"_cts` are of the same type – which is very different from the results of the GNU extension. However, the important thing is that `compile_time_string` can now be used as type of a template parameter, so we can just add another layer:

```
template <compile_time_string cts>
struct cts_wrapper {
  static constexpr compile_time_string str{cts};
};
```

Corresponding to the previous compile-time string type comparison, we now need to write:

```
auto a = cts_wrapper<"Hi"_cts>{};
auto b = cts_wrapper<"Hi"_cts>{};
static_assert(
  is_same_v<decltype(a), decltype(b)>);
```

Or we can further simplify it to (as `compile_time_string` has a non-`explicit` constructor):

```
auto a = cts_wrapper<"Hi">{};
auto b = cts_wrapper<"Hi">{};
static_assert(
  is_same_v<decltype(a), decltype(b)>);
```

They have proved to be useful in my real projects, and I hope they will help you too. ■

References

[Clang] https://clang.llvm.org/

[GCC] http://www.gcc.org/

[Godbolt] Matt Godbolt, Compiler Explorer, https://godbolt.org/

[MSVC] https://visualstudio.microsoft.com/

[Park17] Michael Park, '`constexpr` function parameters', May 2017, https://mpark.github.io/programming/2017/05/26/constexpr-function-parameters/

[Smith13] Richard Smith, 'N3599: Literal operator templates for strings', March 2013, http://wg21.link/n3599

[Stone19] David Stone, 'P1045R1: `constexpr` Function Parameters', September 2019, https://wg21.link/p1045r1


```
template <size_t N>
struct compile_time_string {
  constexpr compile_time_string(
    const char (&str)[N])
  {
    copy_n(str, N, value);
  }
  char value[N]{};
};
template <compile_time_string cts>
constexpr auto operator""_cts()
{
  return cts;
}
```

Listing 8

The Year of C++ Successor Languages

2022 has seen many languages created to rival C++. Lucian Radu Teodorescu reports on the current state of the art.

C++ is a peculiar programming language. It is one of the most used programming languages, and yet it is one of the most criticised. According to TIOBE index [TIOBE22], for 30 years, C++ has been in the top 4 programming languages (using a 12-month average). See also Figure 1 (the TIOBE Programming Community Index for October 2022) for language trends in the past 20 years.

For a language that has existed for almost 40 years, to be constantly in the list of top programming languages is a great achievement. It must be a language that is loved by its users. Well, paradoxically, that is not true. C++ is one of the most criticised languages. Personally, I couldn't find any C++ programmer who argues that C++ is a beautiful language. Virtually everyone complains that the language is too big, too complex, with features that should be killed, with too many features, and, conversely, with not enough features. Over-generalising, C++ can be seen as a random collection of features without a clear, cohesive story.

Some of the most notable criticisms can be found on the C++ programming language Wikipedia page [Wikipedia]. While defending the language, Bjarne Stroustrup argues that "within C++, there is a much smaller and cleaner language struggling to get out" [Stroustrup94]. This quote is still in widespread use today, after 28 years. While this is meant to be defending C++, if we analyse it carefully, we realise that it's also an implicit criticism: C++ still hasn't become that smaller and cleaner language that people expect it to be. It may simply mean that this smaller and cleaner language is just a mirage.

So, the main question is: How can we obtain a better language that is simpler and cleaner than the current C++, and occupies the same space (system programming language) as C++? What does a C++ successor language look like?

And, while there have been some attempts to answer this question in the past, 2022 was the only year in which three possible successor languages were announced, all in keynote talks at major C++ conferences.

First, we have Val announced at *C++ Now* by Dave Abrahams and Dimitri Racordon [Abrahams22a, Abrahams22b, Val]. At the core of Val there is the idea that one can build safe and efficient programs using *mutable value semantics* [Racordon22a]

Two months later, at *CppNorth*, the Carbon language was announced by Chandler Carruth [Carruth22, Carbon]. The Carbon language tries to solve several aspects of C++: technical debt accumulated over decades, the prioritisation of backwards compatibility and the C++ evolution process.

Another two months after that, at *CppCon*, Herb Sutter announces CppFront, as a possible successor of C++ [Sutter22]. His main goal was to "bring C++ itself forward and double down on C++" and prevent users from migrating to other languages. The declared aims are to make C++ 50× times safer and 10× simpler.

This article tries to provide a critical perspective on these three languages. I'm not doing this because I think that they can't be C++ successors; quite the opposite, I'm trying to lay out the problems that these languages need to solve before hoping to claim C++'s place. While I do have some personal biases, I'll try my best to be objective in my analysis.

Previous attempts

The D programming language was created by Walter Bright and appeared in 2001; later in 2007, Andrei Alexandrescu joined the design and development effort. This language was supposed to learn from C++'s mistakes and be its successor. It promises the same level of efficiency but adds a ton of new features and simplifies some of the more complex parts of C++. The D homepage advertises D as a language in which one can "write fast, read fast, and run fast".

D had attracted some commercial users, but it's safe to say that it did not reach the status of an important programming language. While Andrei is one of my long-time heroes and I have a considerable respect for Walter, I mainly viewed D as a large collection of language features, loosely tied together. It feels to me that the language lacks a clear foundation that would give cohesiveness to all the features.

The Go programming language was introduced in 2009 by Google; version 1.0 was released in 2012. The goal of this language is to allow programmers to "build fast, reliable, and efficient software at scale". The designers of the Go language disliked C++, so, as a consequence, Go seems more like an evolution of C than an evolution of C++. Go only added generics in 2022, and still lacks widely used features like exception handling.

Go is a language that implies the presence of garbage collection; this makes numerous C++ users consider it inappropriate for system programming. While Go can be called a successful programming language (number 11th in TIOBE Index [TIOBE22]), its success is mainly in the cloud business. Despite its relative success, it can't be called a C++ successor.

Rust is a programming language developed at Mozilla, announced in 2010, with the first version released in 2015. Rust focuses on reliable (memory and thread safety) and efficient software. The Rust language model is based around the so-called *borrow checker*, which tracks the lifetime of all the objects; thus, it can detect safety errors at compile-time and does not require the use of a garbage collector.

Rust, although not as popular as Go ([TIOBE22]), seems to be considered a good replacement for C++. The problem is that there is no clear/clean/universal way to interface between Rust and C++; this makes C++ programmers that want to move to Rust experience an abrupt migration.

Val

Dave Abrahams and Dimitri Racordon announced Val at C++ Now 2022 [Abrahams22a, Abrahams22b] in a talk called 'A Future of Value Semantics and Generic Programming'. They did not claim that Val might

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

The code has a serious safety issue. And this issue cannot be easily seen if we look at this code alone; we have to look at the surrounding code as well

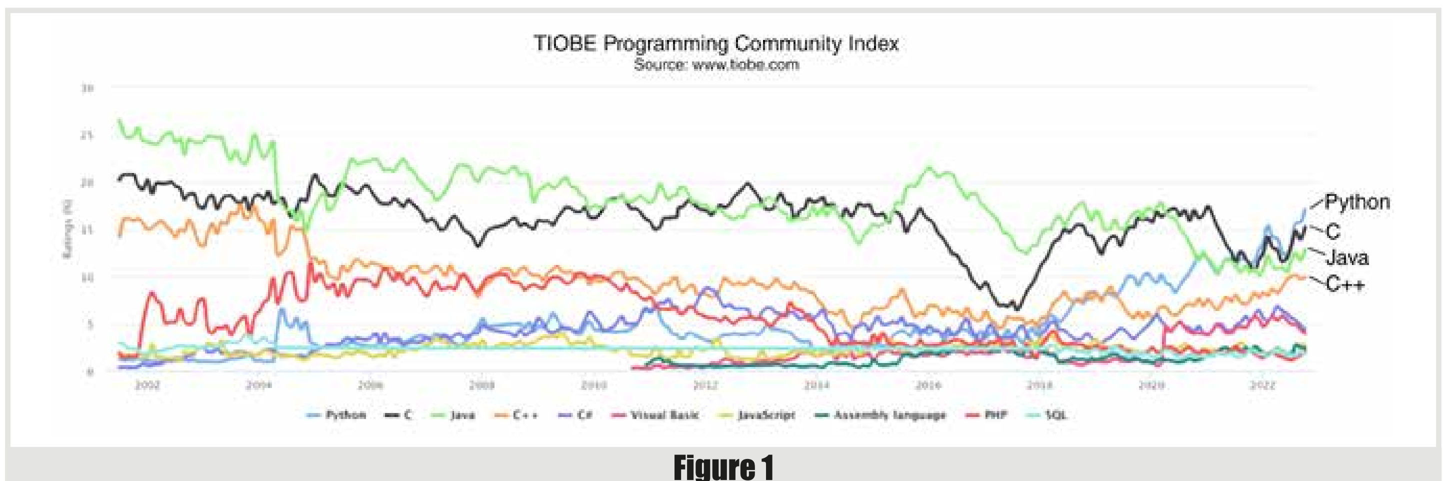


Figure 1

be a C++ successor language, but based on the title of the talk and the surrounding context (keynote at a major C++ conference) people inferred that Val might be one. Dave and Dimitri gave two more talks at *CppCon* 2022 that strengthened this position ([Abrahams22c, Racordon22b]).

Val positions itself with the following aims [Val]:

- Fast by definition
- Safe by default
- Simple
- Interoperable with C++

Val targets the audiences of C++, Rust and Swift languages with these goals. It aims to achieve the performance of C++ but guarantees safety in a simpler way than Rust does it. In terms of performance, Val aims to reduce the amount of object copying and memory allocations needed for writing safe software. In terms of safety, all constructs in Val are guaranteed to be safe, unless the user explicitly asks for extra control (marking portions of the code as *unsafe*). The simplicity of the language mainly comes from its strong Swift influence, which is usually considered to be a simple-to-use language.

Many programming languages don't necessarily have a core idea that goes through all its features and acts like a catalyst for the language; this creates the impression that those languages lack coherence. This cannot be said about Val. This language stands out as having a model to programmatically eliminate safety issues: it's called Mutable Value Semantics [Racordon22a]. But, before we get there, let's explore the main problem that it solves.

C++ is inherently unsafe

It all starts with the observation that, in the presence of mutation, reference semantics can lead to unsafe programs. Because reference semantics allow the creation of complex dependency graphs, mutation cannot guarantee that safety is preserved across the entire graph. If, for example, a function operates on two objects and changes one of them, there is no guarantee that the other object doesn't change in a completely unexpected way. This creates a problem in both single-threaded and multi-threaded environments. Moreover, there isn't a systematic way for us to validate the consequences of a mutation without deeply inspecting all the code that is potentially impacted. This simply breaks the core ideas of structured programming.

Take the following C++ code snippet:

```
void append_vec(vector<int>& dest,
                const vector<int>& src) {
  for ( auto x: src )
    dest.push_back(x);
}
```

Ignoring the inefficiency in the implementation, the code has a serious safety issue. And, this issue cannot be easily seen if we look at this code alone; we have to look at the surrounding code as well. If the caller of this function provides the same vector both as source and as destination parameter, then this leads to undefined behaviour.

To ensure proper semantics for functions like this, we need an *independence guarantee*: we need to ensure that the objects we interact with (and we write to at least one of them) are not identical. This cannot be properly enforced in the language; thus we are inherently in unsafe territory.

I would like to point out that the issue here is more complicated than it looks. If both arguments of a function are `const` references (i.e., we are not changing anything in them), then there is no issue. The problem only arises when we have mutation.

Swift solves this problem by using the copy-on-write technique. But this can lead to inefficiencies.

Rust solves this problem by keeping track of lifetimes for the objects. This adds a burden to the programmer, and can add unnecessary restrictions to the programs.

I'm not saying that Val can't properly interoperate with C++; it's just that implementing this may not be a simple endeavour

Mutable value semantics

Functional programming languages avoid the above issue by forbidding mutation. It's OK to have multiple references to objects, as nobody can change these objects. This feels unnatural for many programmers, and it's inefficient for countless algorithms.

Val solves this problem in an entirely different way: it adds restrictions to references, and ensures that nobody can read an object while somebody else is allowed to change it.

Val recognises the importance of whole/part relationships. These can only form a tree, not a cyclic graph. If we want to modify an object of this tree, we immediately know the impact of that change, i.e., all other objects that can potentially affected by this mutation. It allows us to reason what objects are safe to be passed as read and as write into a function.

In the end, following this logic, we can safely add references to represent whole/part relationships.

In the Val model, mutation is not forbidden, but each time we mutate an object, the compiler can compute which objects can be safely read and which objects can be safely written at the same time. Safety can be guaranteed by construction.

Eliminating arbitrary references between objects and focusing on whole/part relationships is what gives Val value semantics. But, because Val also allows mutation of values, we can call this model Mutable Value Semantics. More information about this model can be found in [Racordon22a].

Scientific approach

Reaching this point, it makes sense for me to touch on an aspect that I consider important: Val seems to follow a scientific approach.

The reader can see that in the previous section we (briefly) describe a computation model that ensures safety. It's not just a claim that the author makes about the language being safe. They have a proof of safety, under the restrictions imposed by the language.

Dimitri Racordon, the main creator of the language, is actually a post-doc researcher. Dave Abrahams also seems to be like-minded. Dave joined Sean Parent to re-form Adobe's STLabs. The research-oriented influence of Alex Stepanov (creator of STL, and previous member of STLabs) on both Dave and Sean can be seen.

There is no guarantee that Val will be as successful as C++, but one can spot the sound approach of solving some fundamental issues of C++: clearly define the problem and then come up with a general and elegant solution.

Using ad hoc references

Val simply denotes as unsafe the usage of ad hoc references. This makes it unclear how one can implement programs that need references beyond expressing whole/part relationships.

For example, implementing a doubly linked list requires references that cannot be modelled as whole/part relationships. It is not clear how to implement doubly linked lists with *mutable value semantics*. As another example, consider a shared cache component in an application. By definition, such a component needs to be accessed by multiple parties, and needs to allow mutation. Again, it's not clear how this can be implemented in Val.

Maybe the simple answer to these examples is that the user must mark some code as *unsafe*. That may be OK; we, as users of the language, just lack the experience on how these cases would be handled. Val has to provide good guidance for handling such cases.

C++ interoperability

As the time of writing this article, Val has no clear public plan for handling interoperability with C++; it just declared its intention. To become a C++ successor language, Val needs to solve this problem. And, it appears that this problem is not an easy one.

The first thing to notice is that, according to its description, Val is mostly inspired by Swift [Val]. This means that the gap between Val and C++ is not small (larger than the gaps between Carbon and Cpp2 on one side, and C++ on the other side). Closing this gap may require significant effort.

The second obstacle is the restrictions imposed by the *mutable value semantics* system. C++ inherently contains a lot of ad hoc references. This means, that C++ code would be seen in Val to contain countless unsafe operations. In my mind, it feels that almost all C++ operations ought to be marked *unsafe* in Val. This seems to increase the interoperability gap.

Please note, I'm not saying that Val can't properly interoperate with C++; it's just that implementing this may not be a simple endeavour.

Carbon

Carbon is a language announced as a (possible) C++ successor language at CppNorth 2022 [Carruth22, Carbon]. Carbon is backed by Google (and, according to Chandler, also by Adobe). Furthermore, as an interesting fact, Google was the big name absent at CppCon 2022; maybe this is an indicator that Google is serious about moving away from C++.

In his talk, Chandler, started enumerating the current problems with C++:

- a lot of technical debt (40 years of C++, plus all the technical debt from C)
- C++ prioritises backward compatibility over language evolution; this also prevents fixing technical debt
- the ISO process of language evolution is not optimised for the actual needs of C++ evolution

The solution to these problems, according to Chandler, is to start thinking about a C++ successor language. Similar to how C++ was created to be a successor of C, how Swift was created to be a successor of ObjectiveC

The learning curve for Carbon can be smooth, and the transition from C++ to Carbon made without jumping through too many hoops.

and how Kotlin was created as a successor of Java, we need to find a successor language to C++.

To create a C++ successor language, we need builds within the existing ecosystem, provide bidirectional interoperability and ensure we have tools to assist us in migration and learning. And those are actually the goals of the newly announced Carbon language.

Carbon doesn't seem to have an emblematic feature compared to C++. It just feels like a C++ cleanup project. In the announcement keynote, Chandler showed a cleaner syntax, cleaner pointer semantics, better packaging, better defaults for public/private members, explicit `self` parameter, inheritance cleanup, API extension points, and C++0x-style generics. All these features are present in other programming languages, in one way or another.

Better defaults

Carbon can be seen as C++ with better defaults. This is a good thing. People will see a familiar language that is just better/simpler. The learning curve for Carbon can be smooth, and the transition from C++ to Carbon made without jumping through too many hoops.

But, on the other hand, how is this different from D? D also attempted to be a C++ successor by learning from C++ mistakes and cleaning its rough edges. What would give the Carbon language its internal coherency and not let it feel like a group of unrelated features?

If we look at this from an evolution perspective, even if all the defaults make a lot of sense today, what guarantees that they would make sense in the following decades? How can we prevent Carbon from accumulating technical debt? A partial answer to this question is, as Chandler mentioned, the use of tools in assisting the migration. But, as we've all seen how painful the migration from Python 2 to Python 3 was; probably not everyone is convinced that tools can help up be future-proof.

All these are questions that the Carbon team need to answer. I'm not trying to claim that these are hard questions to answer, but they need to be answered.

Interoperability with C++ is hard

Even if Carbon can be a C++ with better defaults, interoperability with C++ is not necessarily easy. Here are some points brought up by Sean Baxter [ADSP22]:

- there is no function overloading in Carbon
- there is no exception handling in Carbon
- there is no multiple inheritance in Carbon, but people can still use it in C++
- Carbon doesn't handle raw pointers, unlike C++
- Carbon doesn't have constructors

Looking at these points, it can be easily seen that interoperability with C++ will be a complex topic. Most probably, even if the interoperability issues can be completely resolved, migrating from C++ to Carbon for large software will not be a simple transition.

The rise and fall of the Culture

Google is a company that strongly believes in culture as a driving force for software development. This was also expressed by Chandler in his keynote with a quote from Peter Drucker:

> Culture eats strategy for breakfast, technology for lunch, and products for dinner, and soon thereafter everything else too.

While I do believe that culture in an organisation is essential, just quoting Peter Drucker is not a recipe for success. The main problem is that it's hard to measure culture and its impact. Chandler lays out a couple of points about culture for Carbon (inclusiveness, community friendly, etc.). While all these points are good points, they are not enough to define culture or to make it work for the Carbon project. For example, Chandler doesn't mention technical excellence, perseverance, courage to try new things, or how to prioritise different (culture-related) goals.

In one of the previous companies I worked at, we had a mantra that said 'we never let a project fail'. Does Google and the Carbon project have a similar goal in its culture? People seem to see Google as a company that tries out many products and shuts them down after some time. See, for example, Figure 2 for a tweet from Victor Zverovich [Zverovich22] that capitalises on this perception in a joke about Carbon. This line of thought may not be too far-fetched considering that Chandler also announced that there is a different team in Google that has the same goal, but they start from Rust and move towards C++.

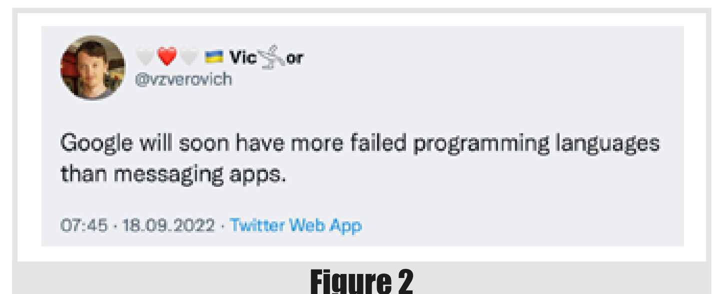


Figure 2

To reiterate: culture is good, and the points that Chandler brought up are good points. But, I'm an engineer: I need verifiable arguments if I'm to be convinced of something.

Governance model

One of the interesting points about the Carbon announcement is the governance model. The Carbon project aims at a governance in which no company dictates the future of the language. Everyone can participate

New code can interact with old code, but old code cannot simply depend on new code written with the new features

in the evolution of the language by creating pull-requests, but the more important the feature is, the more analysis/argumentation is needed.

For significant features that don't have consensus, there is a steering committee of three members (Chandler Carruth, Kate Gregory, Richard Smith) that is responsible to reach to a decision. They don't get the chance to contribute to the design; they just have to weigh the arguments presented to them and make the choice.

It is intriguing to notice that this model tries to emphasise a democratic process, which is somehow similar to the goal that ISO has. It's just a different division of parties involved, with clearer rules of what to do when an impasse is reached. If the same people that work on C++ standardisation worked on Carbon, it's not clear if the Carbon process would be significantly better.

While democratic methods are currently the best way to govern, we've seen recently a series of major political failures that can be directly correlated to downsides of democracy. And, it's worth mentioning, in Ancient Greece, democracy was considered a bad way to govern.

Cpp2

CppFront is a project that was announced by Herb Sutter in the closing keynote of *CppCon* 2022 [Sutter22]. It is a transpiler that converts from a "better C++", i.e., Cpp2, to old C++. While CppFront / Cpp2 was officially announced this year, Herb has been working on this project for about 7 years; each year, Herb has showcased a small part of Cpp2.

Herb wants to improve C++ significantly (i.e., 10×) rather than performing incremental changes (i.e., 10%). He hopes to bring C++ to that old goal of much simpler and cleaner language that Stroustrup envisioned 30 years ago. And, interestingly enough, takes the same approach that Stroustrup took when he wanted to improve on C: start a new language and translate the code to the previous language. Thus, CppFront is a small transpiler that takes Cpp2 code (Herb's new language) and outputs regular C++ code.

Herb also sets metrics that we can use to evaluate whether this experiment succeeds: 50 times safer (that is 98% fewer CVEs), and 10 times simpler (90% less total guidance to teach). Defining metrics upfront is a good strategy to be able to evaluate the success of an experiment; I really like this idea.

Backwards compatibility and interoperability

Cpp2 can be simpler than C++ by dropping backwards compatibility. This finally allows the language to remove features that are considered harmful, and to revisit some of the design choices that proved to be suboptimal. By dropping backwards compatibility, Cpp2 can finally address decades of accumulated technical debt in C++.

Truth be told, prioritising backwards compatibility over language evolution in C++ doesn't have a solid case. Each time we add a major feature to the language (e.g., concepts, coroutines, modules, etc.) we essentially create a new epoch in the language. New code can interact with old code, but old code cannot simply depend on new code written with the new features. Although the C++ standard doesn't officially talk in terms of language epochs, there is an underlying system of epochs in the language, dictated by the releases of new features.

One can think of Cpp2 a major new feature to C++. Things are a bit more complicated in terms of interoperability and tooling, but the essence is the same. There are no good technical reasons why old-style C++ cannot coexist with Cpp2 in the same application.

By design, Cpp2 is semantically close to C++; this makes interoperability easier. On the other hand, this can prevent Cpp2 from having entirely different features from C++. For example, it would be hard for Cpp2 to use C++0x-style generics.

Addressing safety

A goal of 50× improved safety sounds impressive. If Cpp2 can deliver this, I believe most users of the language will be happy.

Let's put this number in perspective, to thoroughly understand the impact. It means that 98% of C++ applications would not crash any more if they were translated to Cpp2 (assuming that crashes are produced only by unsafe applications). Or that 98% of the C++ web applications would not have vulnerabilities (if there are no other non-C++ vulnerabilities). That would be a drastic reduction of crashes and security vulnerabilities.

This seems too good to be true. Actually, if we analyse this in more detail, it appears that these numbers are too high.

First, if we discuss safety, we need to be clear on what safety is. Safety includes:

1. type safety
2. bounds safety
3. lifetime safety
4. initialisation safety
5. object access safety
6. thread safety
7. arithmetic safety

The first 4 items on this list are addressed by Herb in his keynote. However, not all the aspects of those safety items were addressed. As a prime example, lifetime safety cannot be guaranteed in the presence of raw pointers; just checking pointers for `null` is simply not enough. There is also not a single feature announced to detect use-after-delete cases with pointers.

Cpp2, as described in the CppCon keynote, cannot detect the problem with this code:

```
vec.push_back(vec.front());
```

All three C++ successor languages announced this year are considered to be experiments…we don't have good indicators whether they will actually succeed

Herb defines his safety metric to include the first four safety components; deliberately ignoring the other types of safety seems odd. Especially if the omitted ones are important.

Object access safety refers to safety rules that are influenced by object access patterns. In general, unsafe code in this category can translate into type safety, bounds safety or lifetime safety. The rules for invalidating iterators are great examples for this category.

Thread safety is a big issue in C++ and was not mentioned at all by Herb. In her 2021 C++ Now talk [Kazakova21], Anastasia Kazakova presents data showing that in the C++ community, Concurrency safety accounts for 27% of user frustration. For comparison, bounds safety issues only accounts for 16% and use-after-delete accounts for 15% of user frustration. Concurrency safety is the biggest pain point in terms of safety, and this is not even captured on Herb's list.

Herb claims on his slide that Cpp2 gets "safety by construction". That cannot be true. Safety by construction should mean that the language is built in such a way that always lead to safe constructs (unless programmers really ignore the type system and take safety into their own hands) – similar to how Val or Rust is built. But Cpp2 doesn't do that; it just adds more safety checks for some common sources of unsafe behaviour. This should immediately stand out if the reader has watched the talks given by Dave Abrahams and Dimitri Racordon [Abrahams22a, Abrahams22b, Abrahams22c, Racordon22b], and also Sean Parent's talk on exceptions [Parent22].

This makes me believe that 50× improvement on safety is not achievable as a goal.

On the measurability of the goals

As I mentioned above, I do love the fact that Herb set up metrics for his experiment. Theoretically, at any point, we can measure the progress against these metrics, and we can assess if the experiment is a success or can lead to success.

Let's start with the second metric: being 10× simpler, as measured in the guidance we need to teach in C++ books. It's less likely for people to write books on Cpp2 before this experiment proves to be a success, but we can imagine what the content of such a book would be. We can determine what would be the concepts we need to teach about Cpp2, and we can compare that to the list of things we are currently teaching about C++. Thus, we can measure this metric.

This is not as straightforward as one might think. C++ has a long history; thus we know its pitfalls, and people have documented these in C++ books. But, Cpp2 doesn't have such a rich history, so there is always the suspicion that we don't know all its pitfalls. However, Cpp2 being so close to C++, I honestly believe that we can dismiss these concerns and get an accurate measurement on simplicity.

But, I cannot say the same thing about the second metric. How can we measure the percentage of CVEs and safety bugs? We first need to have a sufficiently large corpus of Cpp2 programs, written by a large variety of programmers and companies. However, in order for that to happen, Cpp2 needs to be considered a success – a circular dependency. Thus, the safety metric, as defined in Herb's talk, is not a good metric to measure the success of the experiment.

Using this metric makes sense to assess the language some time after it has been used in the mainstream, but not to judge the success of the experiment.

To have or not to have monads

At 1h 33 min in the keynote talk (taking the YouTube video as a reference) [Sutter22], Herb Sutter proudly remarks: "I have not said the word monad once". Then he goes on to explain that Cpp2 is all about language ideas that we are currently using in C++; not weird foreign terms from other languages.

While this remark may appeal to the self-centred part of the C++ community, I believe it hurts the community more than it helps.

First, C++ uses monads all over the place. The new C++23 `std::expected` feature may be a known example of using monads, but C++ is fundamentally built around monads. We implicitly use monads when we call functions that may throw exceptions – that is, virtually everywhere.

Secondly, it creates a feeling of self-sufficiency within the language users. Instead of opening the community to new ideas, such a statement transmits the message that C++ doesn't need to learn from other languages. But the huge amount of technical debt the language has, and the appearance of three successor languages, proves otherwise.

Comparison

Table 1 attempts to provide a comparison between the three languages; C++ is also included as a baseline.

All three C++ successor languages announced this year are considered to be experiments. We don't have good indicators whether they will actually

Metric	Val	Carbon	Cpp2	C++
Project status	experiment	experiment	experiment	mature
GitHub stars / active users	272	28.5k	2.4k	millions
Resemblance to C++	lower	medium	high	perfect
Safety	strong	unsafe+	unsafe+	unsafe
Decision coherence	high	medium	medium	low
Theory based	yes	no	no	no

Table 1

succeed in attracting a critical mass of coders/code bases that would use them in production environments.

Looking at the number of stars on GitHub, we see Carbon as the leader of the pack – by a long way, compared to the other two. Carbon has succeeded at creating more hype inside the community; the focus on inclusivity and the governance model might have contributed to this.

The three languages also differentiate themselves in terms of how they resemble C++. As expected, Cpp2 is the closest of the three to C++. Carbon seems further away from C++, but uses the same fundamental building blocks as C++; the user fundamentally thinks in the same terms in Carbon as they used to in C++. Because of Mutable Value Semantics, Val programmers need to have a slightly different mental model when programming, which may present Val as a language further away from C++. On the other hand, if we look at the *fast by definition mantra* of Val, especially in the context of *safe by default* and *simple*, the principles of the language seem to translate well to a C++ audience.

Out of the three new languages, Val is the only one that can back up its promise of safety. The other two try to change some of the defaults for the most unsafe operations; it's unclear if that makes a large difference yet. If Carbon and Cpp2 don't feel like languages that you can easily shoot yourself in the foot with, they probably feel like languages that you can easily inflict knife cuts on your legs with.

All three languages seem to improve on C++ in terms of language feature coherence. But changing the defaults doesn't get you that far in terms of language coherency. Here, Val's approach seems slightly more cohesive compared to Carbon and Cpp2.

Finally, the point which I believe is important in an engineering discipline like ours: how many of the language design decisions are backup by some sort of science? In this respect, Val seems to be the only one that has some theoretical foundation. This can provide real guarantees to its users.

Personal take

Herb started his keynote with a plea not to abandon C++. It's a testament, from C++ leadership, that people are considering abandoning C++. The appearance of three C++ successor languages in a single year just confirms the same idea. Whether C++ is starting to lose popularity or not is not yet known, but we can probably assume that this year is an inflection point for the future of C++.

Currently, it's too early to tell whether any of these experiments will succeed or not. All languages have strengths, and all of them have weak points. If at least one of them succeeds, I believe we advance the practice in programming languages; that probably means a positive impact in the software industry overall.

As much as possible, I have tried to be objective in this comparison, but I do have my biases. I hope that they didn't prevent me doing a decent job of comparing these languages.

Speaking of biases, I do need to confess: in my spare time, I have started working with the Val team to push the core ideas of the language forward. To me, the ideas, if they can be perfected and adopted successfully in practice, are more important than particular languages. If Val dies as a programming language but all its ideas are incorporated in C++, then I will be delighted.

I have been captivated by the ideas of *mutable value semantics* since I saw the recordings of Dave and Dimitri's talks from C++ Now [Abrahams22a, Abrahams22b]. I distinctly remember at that point that I contemplated writing an *Overload* article on the subject; well, here we are. Meeting Dave and Dimitri at *CppCon* 2022 and spending time with them walking through the details, convinced me that the ideas behind Val are profound, well thought through, and that they deserve close attention.

Looking at the popularity numbers, Val doesn't do that well. Probably one of the reasons for this is the fact that good ideas take time to settle in. To paraphrase a famous speech, I chose to work on Val, not because it's easy, but because it's hard; because Val's goals are worthwhile. ■

References

[ADSP22] Connor Hoekstra, Bryce Adelstein Lelbach, Connor, Sean Baxter, *ADSP: The Podcast*, Episode 97: 'C++ vs Carbon vs Circle vs CppFront with Sean Baxter', 2022, https://adspthepodcast.com/2022/09/30/Episode-97.html

[Abrahams22a] Dave Abrahams, A Future of Value Semantics and Generic Programming (part 1), C++ Now 2022, https://www.youtube.com/watch?v=4Ri8bly-dJs

[Abrahams22b] Dave Abrahams, Dimitri Racordon, A Future of Value Semantics and Generic Programming (part 2), C++ Now 2022, https://www.youtube.com/watch?v=GsxYnEAZoNI&list=WL

[Abrahams22c] Dave Abrahams, 'Values: Safety, Regularity, Independence, and the Future of Programming', *CppCon* 2022

[Carbon] GitHub, Carbon Language: An experimental successor to C++, https://github.com/carbon-language/carbon-lang

[Carruth22] Chandler Carruth, 'Carbon Language: An experimental successor to C++', *CppNorth* 2022, https://www.youtube.com/watch?v=omrY53kbVoA

[Kazakova21] Anastasia Kazakova, 'Code Analysis++', *CppNow*, 2021, https://www.youtube.com/watch?v=qUmG61aQyQE

[Parent22] Sean Parent, 'Exceptions the Other Way Around' https://www.youtube.com/watch?v=mkkaAWNE-Ig

[Racordon22a] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, Brennan Saeta, 'Implementation Strategies for Mutable Value Semantics' https://www.jot.fm/issues/issue_2022_02/article2.pdf

[Racordon22b] Dimitri Racordon, 'Val Wants To Be Your Friend: The design of a safe, fast, and simple programming language', *CppCon* 2022, https://www.youtube.com/watch?v=ELeZAKCN4tY&list=WL

[Stroustrup94] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley Professional, 1994

[Sutter22] Herb Sutter, 'Can C++ be 10× simpler & safer … ?', *CppCon* 2022, https://www.youtube.com/watch?v=ELeZAKCN4tY&list=WL

[TIOBE22] TIOBE, TIOBE Index for October 2022, October 2022, https://www.tiobe.com/tiobe-index/ (last accessed October 2022)

[Wikipedia] Wikipedia, C++, https://en.wikipedia.org/wiki/C%2B%2B#Criticism

[Val] The Val Programming Language, https://www.val-lang.dev/

[Zverovich22] Victor Zverovich, 'Google will soon have…', Twitter, 2022, https://twitter.com/vzverovich/

Advertise in C Vu & Overload

80% of readers make purchasing decisions, or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for info.

An Introduction to Go for C++ Programmers

Learning another language is always interesting. Arun Saha walks us through Go as a C++ programmer.

Go is a statically typed, compiled programming language with memory safety, garbage collection, and CSP-style concurrency [Go] [Wikipedia]. It was designed at Google in 2007, publicly announced in 2009, and version 1.0 was released in 2012. It is open-sourced under BSD-3-Clause license and developed at github [Github].

You might wonder why we are talking about the Go programming language. While most of the other top programming languages are much older, Go has achieved significant usage and popularity within just ten years of its existence [TIOBE] [Stackoverflow]. I believe that this is not accidental but a result of different language design decisions. On one hand, it has almost C- and C++-like efficiency, while on the other hand, it has Python-like brevity and a batteries-included approach.

I have been a long-time C++ and C programmer. I started learning and using Go last year. During this (ongoing) journey, I have noticed a lot of elements in Go that are similar to C++ and many elements that are different. In this article, I would like to share that learning with you. (The concurrency aspects are part of a future article.)

Variable declaration

A variable declaration in C++ has the type specified to the left of the identifier. For example,

```
int result = 42;
```

In a variable declaration in Go, the order is reversed – the type is specified to the right of the identifier. The equivalent in Go is the following.

```
var result int = 42
```

This is perhaps the biggest habit change necessary for reading and writing Go. The designers have chosen this deliberately [Pike10]. It took me a while to get used to this.

Semicolons

Unlike C++, semicolons are optional to terminate statements in Go. The lexer insert semicolons automatically, so the source code is mostly free of them. If only multiple statements are written on a line, then semicolons are necessary to separate them.

Declaration versus assignment

Go chose `:=` (colon equals) as a shorthand notation to define and initialize a variable within the scope of a function or a loop.

```
attempt := 1 // Shorthand declaration and
             // assignment
```

A variable declaration needs the `var` keyword outside of a function. It can be used inside a function as well. The following notation first defines a variable and later assigns to it.

```
var attempt int // Declaration
…
attempt = 1 // Assignment
```

Obviously, the above two approaches can be combined to have an explicit type declaration and assignment, as shown in the following.

```
var attempt int = 1 // Long declaration and
                    // assignment
```

While using the new shorthand notation, a common beginner confusion is the following.

```
sum := 0
…
  sum := newsum // Error: Multiple declaration
                // of 'sum'
  sum = newsum  // OK. Assignment
```

Zero initialization

In Go, any declared but not explicitly initialized variable would be automatically zero-initialized. There are well-defined zero values for each type, for example, `0` for numeric types, `false` for boolean, `""` (empty string) for strings. Thus, the following statement not only declares but also initializes the variable.

```
var result int
```

I love this feature!

In C++ (and some other languages), a lot of bugs boil down to uninitialized variables as they do not have any automatic or implicit initialization. This required introduction of compiler flags like `-Wunintialized`, `-Wmaybe-uninitialized` [GCC] to detect uninitialized variables that the programmers must remember to enable and enforce. Go eliminates all those hassles and errors through this simple language specification.

Type declaration

A type declaration defines a new named type that has the same underlying type as an existing type. The following example declares Miles as a new type with float64 as the underlying type.

```
type Miles float64
```

Two named types with the same underlying type cannot be assigned or compared as shown in the example below.

```
type Kilometers float64
var m Miles = 26.2
var k Kilometers = 42
k = m // compilation error
equal := k == m // compilation error
```

Arun Saha Arun is a software engineer and works in different areas of software-defined data centers including networking and storage systems. Arun is passionate about building robust software infrastructure, engineering high quality software, and improving productivity. Arun holds a B.S. and Ph.D. in Computer Science. He can be reached at arunksaha@gmail.com

There is a strong and widely used convention for generating and propagating errors. Any function where something can go wrong usually returns an error along with its usual return value(s).

Functions

A function is defined with the `func` keyword as shown below.

```
func add(a int, b int) int {
  return a + b
}
```

Go allows multiple return values from a function. The following function returns both the sum and the difference of two values.

```
func sumdiff(a int, b int) (int, int) {
  sum := a + b
  diff := a - b
  return sum, diff
}
```

It can be called and used in the following way.

```
func multipleReturn() {
  sum, diff := sumdiff(2, 3)
}
```

The return values could be named. It helps disambiguate between multiple return values of the same type.

```
func sumdiff2(a int, b int) (sum int, diff int) {
  sum = a + b
  diff = a - b
  return
}
```

The returned variables (`sum`, `diff`) are defined in the `return` statement and assigned in the body of the function. The final `return` statement is required.

Go does not support function overloading.

Constructor and destructor

In C++, the name of the constructor is the same as the class name. A class may have one or more constructors.

Go does not have constructors. Instead, the following convention is followed. A package provides public functions with names starting with New to (1) allocate an object, (2) initialize it per the package's needs, and (3) return the allocated object. The following is an example of creating a new list from the "container/list" package in the Go standard library [Go].

```
// Create a new list and put some numbers in it.
l := list.New()
e4 := l.PushBack(4)
```

In absence of such `New` functions, instantiating a struct performs zero initialization to all its members.

Go does not have destructors.

Error handling

Go does not have exceptions. However, there is a strong and widely used convention for generating and propagating errors. Any function where something can go wrong usually returns an error along with its usual return value(s). The returned error is part of the function signature, it is usually the last of the returned values. If a function can return an error, then the caller is expected to check that; it can handle it or pass it up to its caller.

The following example is from the Go standard library [Go]; `Open()` opens the named file for reading. On successful opening, it returns a File object and `nil error`. If it fails to open, it returns a `nil File` object and an error object to capture the cause.

```
func Open(name string) (*File, error)
```

It can be used as follows.

```
f, err := os.Open("notes.txt")
if err != nil {
  log.Fatal(err)
}
```

In Go, `nil` is the zero value for pointers, interfaces, maps, slices, functions, etc. It is equivalent to `nullptr` in C++.

Go represents a potential error state with the built-in interface type, `error`. A `nil error` represents no error.

Go has a built-in function `panic()` that stops the ordinary flow of control and begins panicking. It can be initiated by invoking `panic()` directly. They can also be caused by runtime errors, such as division by zero.

Defer

Go provides a defer mechanism to specify a function that will be called at the exit of the current scope. It is similar to ScopeGuard or `std::experimental::scope_exit` in C++. Defer is used as a regular pattern for unlocking mutexes, closing files, etc. The example below uses defer for closing a file when the function returns.

```
// Contents returns the file's contents as a
// string.
func Contents(filename string) (string, error) {
  f, err := os.Open(filename)
  if err != nil {
    return "", err
  }
  defer f.Close() // f.Close will run when we're
                  // finished.
<truncated>
```

`defer` is not a substitute for a destructor since there is no way to use it when a heap-allocated object is deconstructed.

The built-in function `recover()` regains control of a panicking situation. It is only useful inside deferred functions. If the current flow of control is panicking, a call to recover will capture the value given to panic and resume normal execution.

Visibility

Unlike C++, Go does not have class member visibility qualifiers like `public`, `protected`, or `private`. In Go, any variable, constant, function

A significant difference from C++ is that the object of the member function can be named anything

or struct data member starting with an upper-case character is public; others (starting with a lower-case character) are private. For example,

```
type Person struct {
  Name string // public data member
  Phone string // public data member
  creditCardNumber string // private data member
}
```

Methods

Go allows defining methods on types. A method is a function with a special receiver argument. The receiver appears in its own argument list between the func keyword and the method name.

In this example, the `Distance` method has a receiver of type Point named point.

```
type Point struct {
  X, Y float64
}
func (point Point) Distance() float64 {
  return math.Sqrt(point.X*point.X
                 + point.Y*point.Y)
}
```

Like C++, Go has pointers. A pointer holds the memory address of a variable. (Go does not allow pointer arithmetic though.)

```
point := Point{X:3, Y:4}
ptr := &point
```

If the method needs to change any of the data members, then the method needs a pointer receiver as the following.

```
func (point *Point) Move(dx, dy float64) {
  point.X += dx
  point.Y += dy
}
```

Like C++, methods can be invoked either on the variable type or the pointer type.

```
dist1 := ptr.Distance()
dist2 := point.Distance()
```

A significant difference from C++ is that the object of the member function can be named anything, as opposed to the reserved keyword `this`.

Const

A Go program can define compile-time constants as const.

```
const separator = ","
```

But a variable cannot be qualified as const at its declaration and initialization. I.e., there is no equivalent of the following C++ expression.

```
int const result = ComputeResult(…);
```

There is no mechanism for const pointers or pointers to const data. Methods with non-pointer receivers behave as const member functions.

The following member function uses a non-pointer receiver (i.e., `Person` instead of `*Person`) and is equivalent to a const-member function in C++.

```
func (p Person) GetName() string {
  return p.Name
}
```

On the contrary, the following member function uses a pointer-receiver (`*Person`) and is equivalent to a non-const-member function in C++.

```
func (p *Person) SetPhoneNumber(ph string) {
  p.Phone = ph
}
```

Loop

There is only one kind of loop available in Go, the `for` loop.

The following is a traditional init-condition-post style `for` loop. There are no parentheses to enclose the init-condition-post portion. Note that, the only kind of increment that Go offers is post-increment (i.e., `i++`).

```
func Sum(n int) int {
  sum := 0
  for i := 1; i <= n; i++ {
    sum += i
  }
  return sum
}
```

The following is a range-based `for` loop iterating over a sequence of `int`s.

```
func SumIntSequence(nums []int) int {
  var sum int
  for _, elem := range nums {
    sum += elem
  }
  return sum
}
```

The `range` returns two values for each iteration, the index and the element. The _ is a placeholder for a return value that is not used subsequently in the code. In the above code, _ is used to ignore the returned index value.

Common data structures

The two most widely used data structures in Go are slices and maps. Both are built into the language.

Array

Like almost all other languages, an array is a sequence of contiguous mutable elements of fixed length. The following is an array of four strings.

```
suits := [4]string{"clubs", "diamonds", "hearts",
  "spades"}
```

Slices, described below, are based on arrays. Most of the time, instead of using arrays directly, Go programs use slices.

the placement of a variable in stack versus heap is up to the compiler...if the lifetime of a variable exists beyond the scope of a function – based on escape analysis – then the compiler places it on the heap

Slice

Slice is a non-owning view of a subsequence of contiguously stored mutable elements in an underlying array. It is written as `[]T` where the elements are of type `T`. A slice has three components: a pointer, a length, and a capacity.

A slice can be defined using a new underlying array or specifying a half-open range of the subsequence in an existing array or another slice.

```
myCards := []string{"CA", "D9"} // slice based on
                                // a new underlying array
redSuits := suits[1:3]  // slice based on
                        // an existing array
trump := redSuits[:1]   // slice based on
                        // another existing slice
```

Multiple slices may refer to the same underlying storage, and those slices' views may overlap.

```
majorSuits := suits[2:] // overlaps with redSuits
```

For slices with overlapping contents, mutating an element through one slice is visible to the other slices.

```
redSuits[1] = "xxx"
fmt.Printf("%q\n", majorSuits)
  // prints: ["xxx" "spades"]
```

Unlike arrays, slices are growable using the built-in function `append()`. If the underlying array has reached its capacity, then `append()` allocates a new underlying array, copies the previous contents, and appends the new ones.

```
myCards = append(myCards, "CQ")
  // myCards: ["CA" "D9" "CQ"]
```

If other slices were sharing the original array, those slices and the original array stay untouched.

Erasing elements from a slice is achieved by concatenating the slice before and the slice after. For erasing the *i*th element, we concatenate the (*i*-1) elements on the left, i.e., `[:i]`, to all the elements on the right, i.e., `[i+1:]`. The following example erases the element at index 1.

```
myCards = append(myCards[:1], myCards[2:]...)
  // myCards: ["CA" "CQ"]
```

From a C++ viewpoint, the slice has some similarities to `std::vector` from the storage management aspect and it has some other similarities to `std::span`, and `std::string_view` from the view sharing aspect.

Map

The map is a reference to a hash table, an unordered collection of key-value pairs, in which all the keys are distinct, and the value associated with a key can be retrieved, updated, or removed in constant time. It is written as `map[K]V`, where `K` and `V` are the types of its keys and values. The following map associates `string`s to `int`s.

```
var rgbMap map[string]int
```

A map needs to be initialized with the built-in make function.

```
rgbMap = make(map[string]int)
```

The following shows insertion and retrieval.

```
rgbMap["red"] = 1           // insert or update
redCode := rgbMap["red"]    // retrieve
```

A Go map is equivalent to `std::unordered_map` in C++.

Generics

Ten years after the initial release, Go started supporting Generics in 2022. It is equivalent to templates in C++.

Go allows expressing type constraints. The following example composes (union) the standard library provided `Integer` and `Float` constraints to define the `Numeric` constraint.

```
type Numeric interface {
  constraints.Integer | constraints.Float
}
```

The generic function `SumSequence()` accepts a slice of type `T` where `T` satisfies the `Numeric` constraint. The generic type `T` and its constraint `Numeric` are enclosed in a pair of square brackets after the function name. The return type is also the generic type `T`.

```
func SumSequence[T Numeric](nums []T) T {
  var sum T
  for _, elem := range nums {
    sum += elem
  }
  return sum
}
```

The statement `var sum T` performs default zero initialization for the actual type. Like C++, you can build generic data structures. The following example shows building a generic set data structure.

```
type Set[K comparable] struct {
  elems map[K]bool
}
func NewSet[K comparable]() *Set[K] {
  var set Set[K]
  set.elems = make(map[K]bool)
  return &set
}
func (set *Set[K]) Add(elem K) {
  set.elems[elem] = true
}
```

A sample user code is the following.

```
seti := NewSet[int]()
seti.Add(42)
```

Stack versus heap allocation and garbage collection

In C++, the local or automatic variables in a function are allocated in the stack. They are deallocated when the function returns. Thus, returning the address of such a variable is a recipe for disaster.

An interface in Go is an abstract type; it is a collection of one or more behaviors (methods) that are offered as part of this interface

In Go, however, the placement of a variable in stack versus heap is up to the compiler. If the lifetime of a variable exists beyond the scope of a function – based on escape analysis – then the compiler places it on the heap. Based on this principle, in the `NewSet()` function above it is okay to return the address of its local variable.

Go has automatic memory management or garbage collection. If there is no path to reach a heap variable from any other package level variable or any currently active functions, then the variable is unreachable and can be deallocated.

Interface

An interface in Go is an abstract type; it is a collection of one or more behaviors (methods) that are offered as part of this interface. This way it is like Pure Abstract Virtual Classes (PABC) in C++. One or more structs can satisfy an interface by implementing all the methods of the interface. Such structs are known as instances of that interface.

The methods are named as verbs (e.g., `Read`, `Write`, `Close`) and the interfaces are named as nouns that perform those verbs (e.g., `Reader`, `Writer`, `Closer`).

The `Reader` interface offers a `Read` method to read from some source, outside the scope of this function, into the byte buffer `buf`, returning the number of bytes read `n` (where 0 <= `n` <= `len(buf)`) and any error encountered `err`.

```
type Reader interface {
  Read(buf []byte) (n int, err error)
}
```

The `Writer` interface offers a `Write` method to write `len(buf)` bytes from the buffer `buf` to the underlying data stream, returning the number of bytes written `n` (where 0 <= `n` <= `len(buf)`) and any error encountered `err` that caused the write to stop early.

```
type Writer interface {
  Write(buf []byte) (n int, err error)
}
```

A user-defined type can implement such standard interfaces and avail the standard library methods. The following example shows how a user-defined type `Gadget` implements the `Writer` interface.

```
type Gadget struct {
  serial []byte
}
func (gadget *Gadget) Write(data []byte)
     (n int, err error) {
  gadget.serial = make([]byte, len(data))
  copy(gadget.serial, data)
  return len(data), nil
}
```

A client of Gadget can use it like the following.

```
serial := []byte("123456789")
gadget := Gadget{}
fmt.Fprintf(&gadget, "%s", serial)
```

Interfaces can be composed to make bigger interfaces.

The interface `ReadWriter` is an interface that combines the `Reader` and `Writer` interfaces.

```
type ReadWriter interface {
  Reader
  Writer
}
```

An expression may be assigned to an interface if and only if its type satisfies the interface.

```
// Declaration: w is a variable of interface
// type io.Writer
var w io.Writer
w = os.Stdout
// OK: os.Stdout is of type *os.File which
// has Write method
w = time.Second
// compile error: time.Second is of type
// time.Duration lacking Write method
```

The empty interface, `interface{}`, also known as `any`, is satisfied by any value.

A struct can satisfy more than one interface. When a struct implements an interface, then it may or may not explicitly specify the interface. When it is not explicitly specified, the compiler uses structural typing to determine if a struct is implicitly satisfying an interface and allows substitution.

An interface value can be converted to its concrete value or a different kind of interface value by an operation known as type assertion. The following example shows how an interface value `w` may be converted to a variable `f` of its concrete type.

```
var w io.Writer
w = os.Stdout
f := w.(*os.File) // success: f == os.Stdout
c := w.(*bytes.Buffer) // runtime panic:
  // interface holds *os.File, not *bytes.Buffer
```

The following example shows how the interface value `w` of interface type `io.Writer` (from above) is converted to interface value `rw` of interface type `io.ReadWriter`.

```
rw := w.(io.ReadWriter)
// success: *os.File has both Read and Write
```

Inheritance

Go does not offer inheritance. A struct cannot inherit another struct. However, inheritance-like behavior can be achieved by designing interface(s), and struct(s) satisfying those interface(s) [Saha21].

Packages and modules

Go source files are bundled into packages, and packages are bundled into modules.

A package groups files of similar functionalities together. The source code for a package resides in one or more `.go` files, usually in a directory

whose name is the same as the package name. All such files list the name of the package at the beginning of the file, e.g., `package fmt`. Files outside the package can refer to or use a package by importing it, e.g., **`import "fmt"`**. Each package serves as a separate namespace. From a C++ point of view, it is similar to a library from the file organization and build aspect, and namespace from the naming scope aspect.

A module is a collection of Go packages stored in a file tree with a `go.mod` file at its root. The `go.mod` file defines the module's dependency requirements.

Eco system

Go is not just a language; it comes with a rich toolchain [Edwards19] ecosystem around it. Following are some frequently used tools in the ecosystem:

1. **`go build`** to build,
2. **`go run`** to build and run an executable,
3. **`go test`** to build and run the tests and benchmarks, and
4. **`go doc`** to build the documentation from comments and examples.

Beyond the basics, there are

5. **`go get`** to download a package from the Internet
6. **`go fmt`** to format the source code uniformly, and so on.

Go offers a flag **`-race`** that can be passed to **`go build`** or **`go test`** to instrument the code for race detection.

Conclusion and further reading

This article is a quick introduction to Go from a C++ perspective. It is by no means a tutorial on Go. For that, please refer to the resources below.

Effective Go [Go-1] and *The Go Programming Language* [Donovan15] are excellent sources for starting to learn Go. The Go Playground [Go-2] is an excellent tool to write and execute Go programs from the comfort of a browser. ■

Acknowledgments

Many thanks to Prakash Jalan, Frances Buontempo, and the *Overload* reviewers for their feedback on the earlier versions of this article.

Note: The opinions expressed in this article are solely the author's.

References

[Donovan15] Alan A. A. Donovan and Brian W. Kernighan (2015) *The Go Programming Language*, Addison-Wesley Professional Computing Series, ISBN: 978-0134190440

[Edwards19] Alex Edwards 'An Overview of Go's Tooling', published 15 April 2019 at https://www.alexedwards.net/blog/an-overview-of-go-tooling

[GCC] 'Options to request or suppress warnings' at https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

[Github] Go: The Go Programming Language – https://github.com/golang

[Go] The Go website: https://go.dev/

[Go-1] Effective Go, https://go.dev/doc/effective_go

[Go-2] The Go Playground: https://go.dev/play/

[Pike10] Rob Pike 'Go's declaration syntax' published 7 Jul 2020 at https://go.dev/blog/declaration-syntax

[Saha21] Arun Saha 'Inheritance in golang' published 27 Oct 2021 at https://medium.com/@arunksaha/inheritance-in-golang-44680461cbcf

[Stackoverflow] 'Most poular technologies' at https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

[TIOBE] 'TIOBE Index for November 2022': https://www.tiobe.com/tiobe-index/

[Wikipedia] 'Go (programming language)': https://en.wikipedia.org/wiki/Go_(programming_language)

Best Articles 2022

Vote for your favourites:

- Best in *CVu*
- Best in *Overload*

Select up to 3 favourites from each journal.

Voting open online at:

https://www.surveymonkey.co.uk/r/ZJ3TF9P

The Testing Iceberg

Many of us are aware of the Testing Pyramid. Seb Rose introduces the Testing Iceberg to explain when we should invest effort in making a test readable to non-technical team members.

Almost 10 years ago, I had a conversation with @mattwynne, which led to the hastily sketched piece of paper (Figure 1). The diagram on the left (since redrawn and blogged about by @tooky [Tooke13]) shows the relationship between end-to-end tests and business-readable tests. Not all business-readable tests need to be end-to-end and not all end-to-end tests need to be business readable.

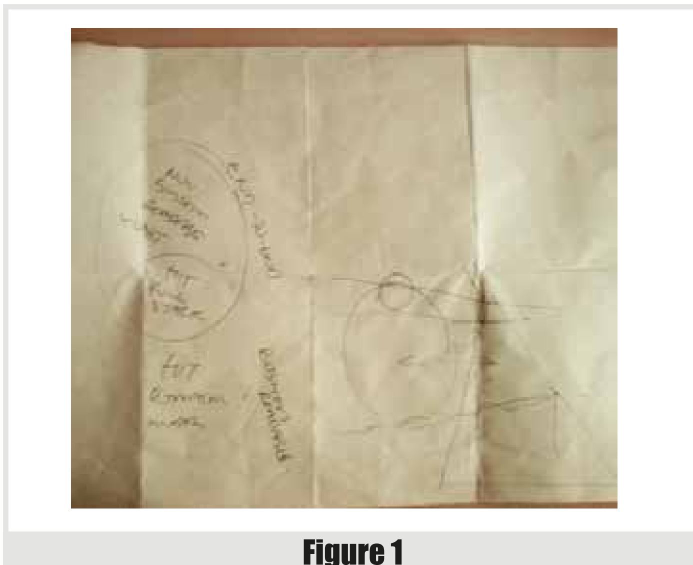


Figure 1

The middle part of the sketch is my attempt to show the relative size of these sets of tests. There should be far more business-readable tests than end-to-end tests. It also shows that most end-to-end tests are business-readable because there are very few situations where purely technical concerns require anything broader than integration tests. The point is, where possible, test the domain model directly and only use end-to-end tests to verify correct 'wiring up' of the entire system.

The far right of the sketch attempts to relate the Venn diagram to the well known Testing Pyramid [Vocke18]. Business-readable tests that hit the domain model directly map to the middle section of the pyramid – integration/component tests. Business-readable tests that hit the full stack map to the top of the pyramid. Not shown is where non-business-readable end-to-end tests should map.

At this point I'm going to re-imagine the Testing Pyramid as a Testing Iceberg (Figure 2: another product of conversations with @mattwynne).

Those portions of the iceberg above the waterline are business-readable, while those below are not. As you can see, in this diagram there are examples of all test types both above and below the readability waterline.

Now I can map non-business-readable end-to-end tests to the submerged system test portion of the iceberg, which is very small because most

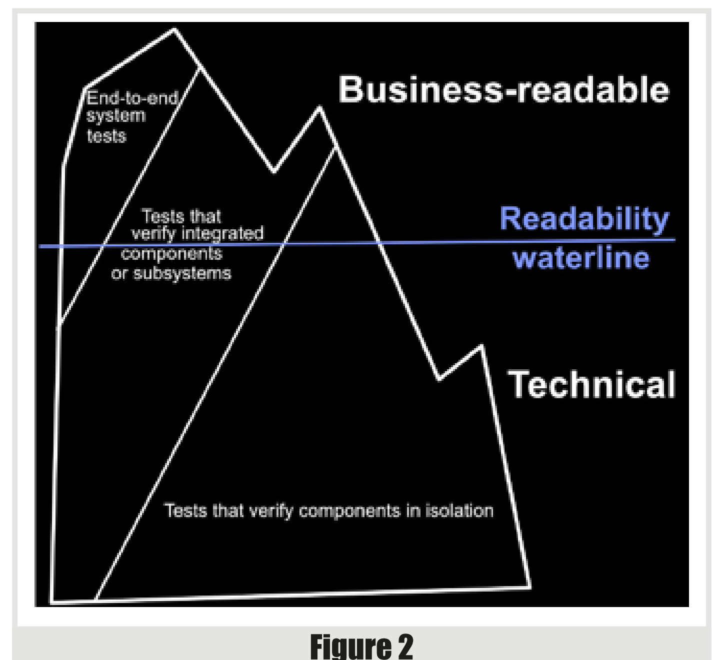


Figure 2

end-to-end tests should be business-readable. Some projects may have specific technical concerns that can only be validated using a fully deployed system, and that are of no interest to business people, but these will be few and far between.

I often get asked how I decide whether a test should be written in a business-readable format (such as Gherkin) rather than a programmer-readable format (such as xUnit). A common anti-pattern is to assume that all end-to-end tests should be business-readable, while all unit tests should be programmer-readable. The Testing Iceberg demonstrates that the question we should actually ask is 'which tests will benefit from being business readable?' If your Product Owner or Business Analyst could give useful feedback on the accuracy of the behaviour a test is verifying, then you will get value from writing that test using a business-readable format. ■

Reference

[Tooke13] Steve Tooke (2013) 'Cucumber and Full Stack Testing' published 18 January 2013 at https://tooky.co.uk/cucumber-and-full-stack-testing/

[Vocke18] 'The Practical Test Pyramid', published 26 February 2018 at https://martinfowler.com/articles/practical-test-pyramid.html

This article is based on a post published on Seb's blog on 14 February 2013: http://claysnow.co.uk/the-testing-iceberg/

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. Co-author of the BDD Books series Discovery and Formulation (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

The Model Student: The Regular Travelling Salesman – Part 2

Richard Harris explores more of the mathematics of modelling problems with computers.

Last time I described the regular travelling salesman problem and we discovered that whilst the shortest tour was trivial to determine, the distribution of tour lengths was a little more difficult. Specifically, the factorial growth of the number of tours as the number of cities increased limited us to tours of no more than 14 cities.

So, how should we go about reducing the computational expense? Well, if we can spot any more symmetries we might be able to exploit them. Taking a look at every 5-city tour, fixing the first city as usual, might give a hint as to whether any more symmetries exist.

Figure 1 shows the complete set of tours for 5-city fixed-start regular TSP. Clearly there's a symmetry we've not yet taken into account since only 4 of the 24 possible tours are distinct from one another!

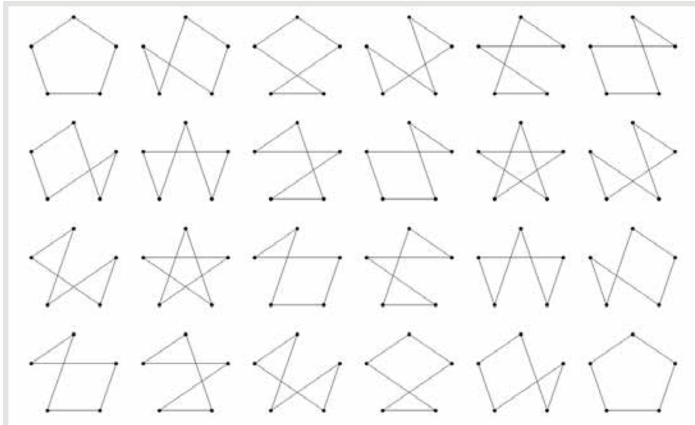


Figure 1

So where is it?

Well, perhaps surprisingly, it's the most obvious of them all. The fixed starting city and tour direction symmetries that we have already addressed exist for all TSPs. This final symmetry results from our tour being around a regular polygon. Specifically, it results from the fact that we can rotate and reflect the city labels on the polygon.

Trivially, reversing the city labels is equivalent to reversing the direction of the tour. More interestingly, rotating the city labels is not necessarily equivalent to rotating the starting city.

This is easily demonstrated by taking a tour that does not have rotational symmetry, say the second in Figure 1, rotating the labels and then checking whether rotating the starting point results in the same tour.

Figure 2 clearly shows that rotating the labels results in a tour that cannot be created by rotating the starting point.

Richard Harris When he wrote this article, Richard had been a professional programmer since 1996. He had a background in Artificial Intelligence and numerical computing and was employed writing software for financial regulation.

Before we embark on constructing an algorithm to efficiently generate the minimal set of symmetrically distinct tours, it's probably worth figuring out how many of them there are. The analysis is easiest for tours with a prime number of cities, p.

Rotating labels for a 5-city regular TSP	
Initial tour	0-1-2-4-3
Rotate labels:	1-2-3-0-4
Rotate starting point:	0-4-1-2-3

Figure 2

First of all, we should count the number of tours for which any rotation of the labels is equivalent to changing the starting city. Trivially, these tours must move the same number of vertices around the perimeter of the polygon at each step since if two consecutive steps were of different lengths, rotating the labels would mean that one of the cities would be followed by a different step, as illustrated in Figure 3.

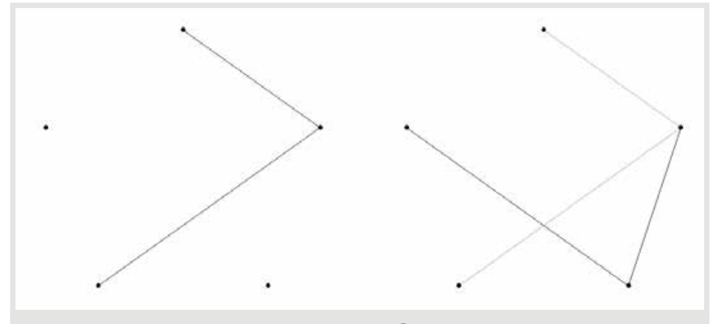


Figure 3

For odd, and hence prime, regular tours there are

$$c_1 = \frac{p-1}{2}$$

such tours (the factor of ½ resulting from the reflectional symmetry).

For prime regular TSPs, all remaining distinct tours must have a layout such that no rotation of the labels is equivalent to a rotation of the starting city.

To see why, assume that rotating the labels k times, where k is not equal to either 1 or p, is equivalent to the initial tour with a different starting city. Rotating it another k times must also be equivalent, as must rotating it any multiple of k times, since we return to an equivalent of the starting tour every time. We should also note that rotating the labels more than p times is equivalent to rotating them that number modulo p.

For each label, l, and any multiple of the k rotations, m, l will be mapped to

$$l \to (l + mk)(\text{mod } p)$$

Now, it is a property of prime numbers that repeatedly applying this mapping must result in every number between 0 and p-1. For p equal to 5 and k equal to 2, we can demonstrate this by enumerating every step

if we're willing to sacrifice a little accuracy, we can simply generate a random subset of the tours

$$0 \to 0+2=2 \to 2$$
$$2 \to 2+2=4 \to 4$$
$$4 \to 4+2=6 \to 1$$
$$1 \to 1+2=3 \to 3$$
$$3 \to 3+2=5 \to 0$$

Whilst this is a reasonable illustration of this fact, it is not remotely akin to a proof. To prove it, we first look for a multiple of the k rotations that maps every label to itself.

$$l = (l+mk)(\text{mod } p)$$

We can subtract the label value from both sides of the equation giving

$$0 = mk(\text{mod } p)$$

Since p is prime, mk can only be a multiple of p if either m or k is a multiple of p.

This demonstrates that if k is not equal to a multiple of p, repeatedly applying k rotations of the labels must generate all other rotations of the labels before returning to the initial layout. Therefore if k label rotations lead to a tour which is equivalent to the first, we simply keep repeating them to find that every possible rotation must also be equivalent.

So the remaining distinct tours must generate $2p^2$, rather than $2p$, tours since they have the extra rotational symmetry of the labels. The total number of tours must be equal to the sum of them both, giving

$$2p^2c_p + 2pc_1 = p!$$
$$c_p = \frac{p! - 2pc_1}{2p^2}$$
$$= \frac{p! - p(p-1)}{2p^2}$$
$$= \frac{(p-1)! - (p-1)}{2p}$$

Hence the total number of distinct tours is given by

$$c = c_p + c_1$$
$$= \frac{(p-1)! - (p-1)}{2p} + \frac{p-1}{2}$$

Whilst this does save us an extra order of magnitude, it's still factorial complexity so it doesn't really help us all that much.

For odd non-prime regular TSPs, the situation is even worse. This is because there will be some distinct tours for which there *is* a partial rotation of the labels that is equivalent to a rotation of the starting city. Since these will generate fewer tours, there must be more distinct tours.

For even regular TSPs, it is only the tour around the perimeter of the polygon for which label and starting city rotation are equivalent. This leads, by a similar argument, to a lower bound for the number of distinct tours being

$$c = \frac{(n-1)! - 2}{2n} + 1$$

The reason that this is only a lower bound is that, as for odd non-prime regular TSPs, there exist partial label rotations that are equivalent to starting city rotations which will each generate fewer tours.

I rather suspect that it's not therefore worth the effort it would require to develop an efficient algorithm for enumerating the symmetrically distinct tours.

So how should we proceed?

Well, if we're willing to sacrifice a little accuracy, we can simply generate a random subset of the tours. If the subset is large enough the resulting distribution of tour lengths should be approximately equal to that of the complete set of tours.

Fortunately for us, the standard library also includes a function for generating random permutations of sequences that we can use to generate our random tours; **`std::random_shuffle`**. Once again, we will ignore the reflectional symmetry for the sake of simplicity. We will still, however, exploit the rotational symmetry, although this time it's to distribute the samples as evenly as possible amongst the full set of tours. Listing 1 shows sampling the tour histogram.

Since we're no longer bound by the number of cities, but by the number of samples we might as well take a look at histograms for large numbers of cities.

Figure 4 and Figure 5 (next page) record the results of 1,000 and 10,000 city regular TSPs, with 10,000,000 and 100,000,000 samples respectively. Table 1 shows the approximate average tour lengths for these histograms.

n	mean	mean/n
1,000	1,274.5	1.27
10,000	12,725.1	1.27

Table 1

```
void
tsp::sample_tour(tour_histogram &histogram,
                 size_t samples)
{
  distances dists(histogram.vertices());
  tour t(histogram.vertices());
  generate_tour(t.begin(), t.end());
  while(samples--)
  {
    std::random_shuffle(t.begin()+1, t.end());
    histogram.add(tour_length(t, dists));
  }
}
```

Listing 1

The central limit theorem states, for a very wide class of distributions, that the sum of a set of independently drawn random numbers is normally distributed

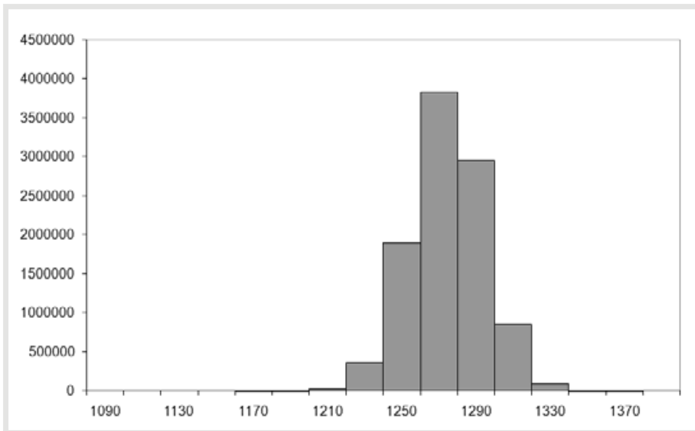


Figure 4

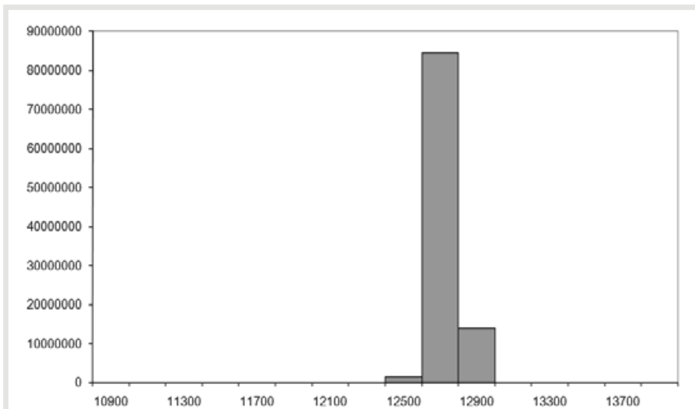


Figure 5

It seems reasonable that the limit of the average tour length is going to be approximately $1.27n$. The question that remains is why? Can we deduce a formula for the limit of the distribution of tour lengths for very large numbers of cities?

For extremely large numbers of cities, most steps in a regular TSP tour are more or less independent to those that have already been taken. It is only when the majority of cities have been visited that the choice of steps will be restricted to limited regions on the circumference of the polygon.

There is a statistical theorem called the law of large numbers which states that as n tends to infinity, the sum of n random numbers independently drawn from any single given distribution tends to n times the average of that distribution. If our assertion that the steps are more or less independent to each other is valid we should be able to approximate the average tour length with n times the average step length. For very large n, the average step length will be approximately equal to the average distance between two randomly selected points on the circumference of a circle of unit radius. In the same way that we can add up a finite set of step lengths and divide by the number of them to get the average, we can integrate the lengths of steps to cities separated by an angle of θ around the circumference and divide by 2π.

$$
\begin{aligned}
\mu &= \frac{1}{2\pi}\int_0^{2\pi} 2\sin\frac{\theta}{2}\,d\theta \\
&= \frac{1}{\pi}\int_0^{2\pi} \sin\frac{\theta}{2}\,d\theta \\
&= \frac{1}{\pi}\left[-2\cos\frac{\theta}{2}\right]_0^{2\pi} \\
&= \frac{1}{\pi}((-2\times -1)-(-2\times 1)) \\
&= \frac{4}{\pi} \approx 1.27
\end{aligned}
$$

This clearly confirms that our expectation of the average tour length was correct, but is not enough for us to completely determine how the tour lengths are distributed.

There is another statistical theorem we can use to help us; the central limit theorem. The central limit theorem states, for a very wide class of distributions, that the sum of a set of independently drawn random numbers is normally distributed. Because of this property, it shows up in a vast number of places.

The normal distribution is defined in terms of both the average, μ, and the standard deviation, σ, of the numbers drawn from it. The standard deviation is a measure of how different on average the numbers in a set are from their mean and it is calculated as follows

Note that in this context E means the expected, or average, value.

Given these values the normal distribution is defined by its cumulative density function, or cdf, which is the function in x that gives the probability that a random number will be less than x.

$$
E(x) = \mu = \frac{1}{n}\sum_i x_i
$$

$$
\begin{aligned}
E((x-\mu)^2) &= \sigma^2 \\
&= \frac{1}{n}\sum_i (x_i-\mu)^2 \\
&= \frac{1}{n}\sum_i (x_i^2 - 2\mu x + \mu^2) \\
&= \frac{1}{n}\sum_i x_i^2 - 2\mu\frac{1}{n}\sum_i x_i^2 + \mu^2\frac{1}{n}\sum_i 1 \\
&= \frac{1}{n}\sum_i x_i^2 - 2\mu^2 + \mu^2 \\
&= \frac{1}{n}\sum_i x_i^2 - \mu^2
\end{aligned}
$$

> **those of you for whom the word 'trigonometry' conjures images of sinister maths teachers intent on ruining your life ... might want to skip ahead and just trust me**

$$F(x;\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x} e^{\frac{-(t-\mu)^2}{2\sigma^2}}\,dt$$

Unfortunately this integral does not have a closed form, meaning a simple formulaic, solution. The derivative, known as the probability density function, or pdf, is simple to calculate, however, and its graph is shown in Figure 6 (the normal distribution pdf).

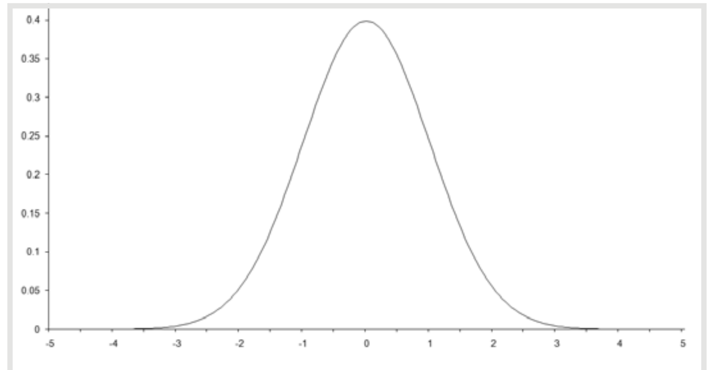


Figure 6

So the final piece of the puzzle is to calculate the average squared distance between two cities in a regular TSP, which we can use to determine which normal distribution is applicable. We could approximate it with an integral over the circle again, but there is an approximate formula for regular TSPs with a number of cities equal to a multiple of 4, so we may as well use it.

$$E(x^2)=\frac{1}{n}\sum_i l_i$$

$$=\frac{1}{n-1}\sum_i^{n-1}4\sin^2\frac{k\pi}{n}$$

$$=\frac{1}{n}\sum_i^{n}4\sin^2\frac{k\pi}{n}$$

This may not look very easy to solve, but appearances can be deceptive. The trick is to exploit some trigonometric identities. It does get a little bit fiddly though, so those of you for whom the word 'trigonometry' conjures images of sinister maths teachers intent on ruining your life (or at least that double period after lunch on Thursdays) might want to skip ahead and just trust me.

Now, the identities in question are

$$\sin\theta=\sin(\pi-\theta)$$

$$\cos\theta=\sin\left(\theta+\frac{\pi}{2}\right)=\sin\left(\frac{\pi}{2}-\theta\right)$$

$$\sin^2\theta+\cos^2\theta=1$$

We can use these by splitting the sum into four parts (Equation 1).

$$E(x^2)\approx\frac{4}{n}\sum_{k=1}^{n}\sin^2\frac{k\pi}{n}$$

$$=\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=\frac{n}{4}+1}^{\frac{n}{2}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=\frac{n}{2}+1}^{\frac{3n}{4}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=\frac{3n}{4}+1}^{n}\sin^2\frac{k\pi}{n}$$

Equation 1

Now since the last three terms are sums over ¼n steps offset by a constant factor, we can simply shift the constant factor from the index into the sum itself (Equation 2).

$$E(x^2)\approx\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{4}\right)+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{2}\right)+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{3\pi}{4}\right)$$

Equation 2

The next point to note is that we can perform the second and fourth sums backwards by subtracting from the last angle in each sum (Equation 3).

$$E(x^2)\approx\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=0}^{\frac{n}{4}-1}\sin^2\left(\frac{\pi}{2}-\frac{k\pi}{n}\right)+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{2}\right)+\frac{4}{n}\sum_{k=0}^{\frac{n}{4}-1}\sin^2\left(\pi-\frac{k\pi}{4}\right)$$

$$\approx\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{\pi}{2}-\frac{k\pi}{n}\right)+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{2}\right)+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\pi-\frac{k\pi}{n}\right)$$

Equation 3

Now we exploit the identity that equates the sine of the angle added to or subtracted from ½p to the cosine of the angle (Equation 4).

$$E(x^2)\approx\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\cos^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\cos^2\frac{k\pi}{n}+\frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}$$

$$=\frac{8}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\cos^2\frac{k\pi}{n}$$

Equation 4

picking the location of the next city in a TSP is equivalent to picking the next city in a tour

Finally, we exploit the identity that equates the sum of the squares of the sine and cosine of an angle to 1 to yield the result.

$$E(x^2) \approx \frac{8}{n}\sum_{k=1}^{\frac{n}{4}} 1 = \frac{8}{n} \times \frac{n}{4} = 2$$

Therefore, the standard deviation of the step length is given by

$$\sigma^2 = 2 - \frac{16}{\pi^2}$$

$$\sigma = \sqrt{2 - \frac{16}{\pi^2}}$$

In addition to stating that the sums of random numbers are normally distributed, the central limit theorem states that the specific normal distribution will have an average equal to n times that of their distribution and a standard deviation equal to the square root of n times that of their distribution.

This means that the distribution of tour length of a regular TSP with n cities should tend, for large n, towards

$$N\left(\frac{4n}{n}, \sqrt{2\pi - \frac{16n}{\pi^2}}\right)$$

Figure 7 compares the histogram we'd expect from the normal distribution (at the bottom) to that we generated by sampling the 1,000 city tour (at the top). Under the assumption of normality, a bucket with mid point x and width w should contain the proportion of the samples given by

$$F\left(x + \frac{w}{2}; \frac{4n}{\pi}, \sqrt{2n - \frac{16n}{\pi^2}}\right) - F\left(\left(x - \frac{w}{2}\right); \frac{4n}{\pi}, \sqrt{2n - \frac{16n}{\pi^2}}\right)$$

Well, despite the fact that the assumption that the tour steps are independent is demonstrably false these look remarkably similar, a fact borne out by the histogram of the difference between them, plotted on the same scale in Figure 8.

In fact, there exists a mathematical technique for determining the likelihood that a sample histogram is consistent with a particular distribution. I strongly suspect that it would indicate that the sample histogram is not consistent with the normal distribution, but since we have already acknowledged that our assumptions are false we shouldn't find that surprising. Nevertheless, given that the maximum difference is of the order of 0.015, or 1½%, it's not too bad an approximation.

So can we perform a similar analysis on the usual type of TSP?

Well, let's assume that the cities are evenly randomly distributed on the unit square. If we're interested in the average tour length of all possible tours we should firstly note that we can take a tour of a random TSP by simply visiting each city in order. Furthermore, every possible tour can be generated by changing the labels and using the same scheme,

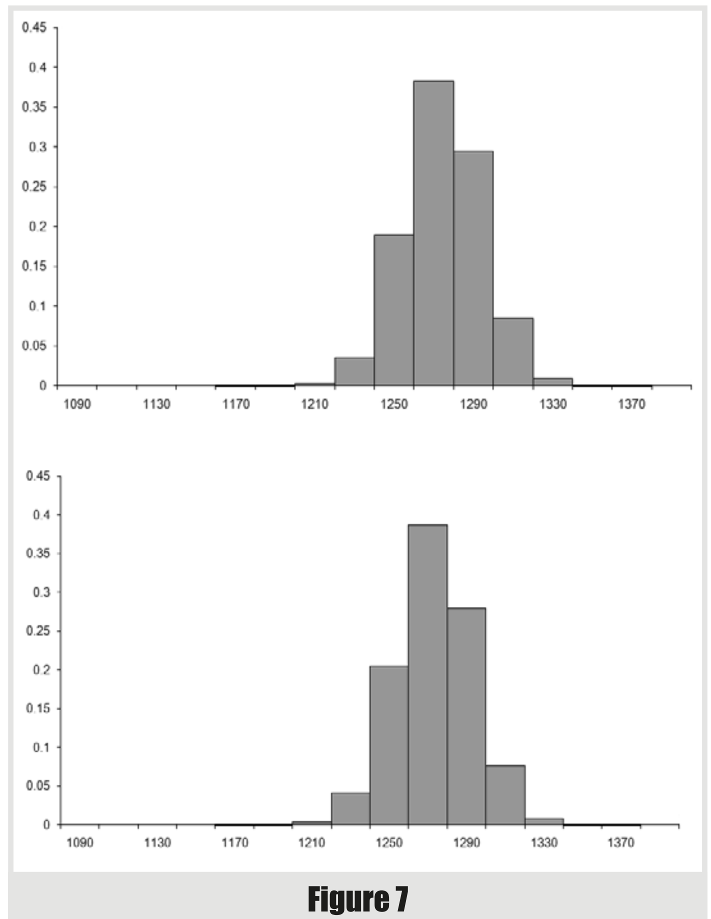


Figure 7

since we can view the labels as instructions as to the order in which we should visit them. This means that picking the location of the next city in

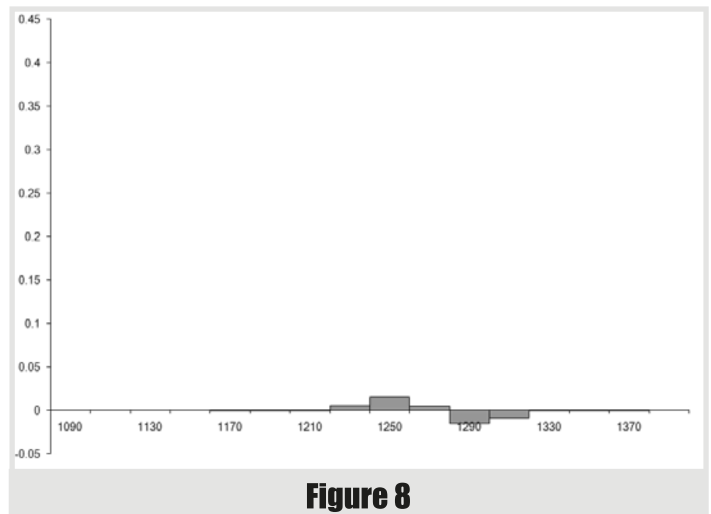


Figure 8

a TSP is equivalent to picking the next city in a tour. Since the former is independent of the cities already chosen, the latter must be independent the steps already taken, satisfying the independence requirement of the law of large numbers.

However, the distribution of step lengths is dependent on where in the square we are currently located, and this breaks the requirement that the step lengths are identically distributed. However, there is another version of the law of large numbers which states that the sum of independent random numbers from different distributions will tend to the sum of the averages of those distributions. Known as the strong law of large numbers, it requires that the standard deviations of those distributions have a particular property which happens to be satisfied if they do not grow without limit, or in other words are all less than some finite number. For cities in the unit square, this will be true for any reasonable definition of distance and so this approximation is actually more reasonable for normal TSPs than it is for regular TSPs.

Unfortunately, the expression for the average step length is a little bit more complicated this time. If we represent a pair of points by their coordinates on the unit square, (x, y) and (a, b), we have

$$E(l) = \frac{\int_0^1\int_0^1\int_0^1\int_0^1 dist((x,y),(a,b))\ dx\ dy\ da\ db}{\int_0^1\int_0^1\int_0^1\int_0^1 1\ dx\ dy\ da\ db}$$

$$= \int_0^1\int_0^1\int_0^1\int_0^1 dist((x,y),(a,b))\ dx\ dy\ da\ db$$

Once again, this is because the integral is the continuous limit of a sum. The fraction is the limit of the sum of the distances between all pairs of points in the unit square divided by the number of such pairs.

For the usual definition of distance, the integral becomes

$$E(l) = \int_0^1\int_0^1\int_0^1\int_0^1 ((x-a)^2 + (y-b)^2)^{\frac{1}{2}}\ dx\ dy\ da\ db$$

Whilst I'm not willing to assert that this does not have a closed form solution, it's too complicated for me to attempt. If we change the cost of travelling between cities to the square of the distance, it becomes a little easier, however.

$$E(l) = \int_0^1\int_0^1\int_0^1\int_0^1 (x-a)^2 + (y-b)^2\ dx\ dy\ da\ db$$

$$= \int_0^1\int_0^1\int_0^1\int_0^1 x^2 - 2ax + a^2 + y^2 - 2by + b^2\ dx\ dy\ da\ db$$

$$= \int_0^1\int_0^1\int_0^1 \left[bx^2 - 2abx + a^2b + by^2 - b^2y + \frac{1}{3}b^3 \right]_0^1 dx\ dy\ da$$

$$= \int_0^1\int_0^1\int_0^1 x^2 - 2ax + a^2 + y^2 - y + \frac{1}{3}\ dx\ dy\ da$$

Continuing in the same vein leads to the result

$$E(l) = \frac{1}{3} - \frac{1}{2} + \frac{1}{3} + \frac{1}{3} - \frac{1}{2} + \frac{1}{3}$$

$$= \frac{4}{3} - 1$$

$$= \frac{1}{3}$$

The average cost of a tour should therefore be approximately equal to $\frac{1}{3}n$.

If you are interested, I invite you to investigate the accuracy of this approximation for different numbers of cities. You may be surprised as to just how accurate it actually is.

So is there anything more that can be said about the statistical properties of tours through TSPs? Well certainly, but not by me as I am afraid I have exhausted my mathematical toolbox. But this is an active area of research and a great many results have been found, of which just a few are described below.

Beardwood, Halton and Hammersley [Beardwood59] proved that the expected length of the shortest path through a random TSP tends to a value proportional to the square root of the number of cities.

Jaillet [Jaillet93] examined the probabilistic TSP in which each city has a probability that it may be skipped during the tour and provided bounds on the expected length of the shortest tour.

Agnihothri [Agnihothri98] examined the travelling repairman problem in which a repairman must travel to fix machines when they break down and developed a mathematical model with which expected travelling time, amongst other things, can be calculated.

And you, dear reader, may be able to shed further light on the properties of either the regular or normal TSP, and if you do please let me know. ■

Acknowledgements

With thanks to Larisa Khodarinova for a lively discussion on group theory that led to the correct count of distinct tours and to Astrid Osborn and John Paul Barjaktarevic for proofreading this article.

References

[Agnihothri98] Agnihothri, 'A Mean Value Analysis of the Travelling Repairman Problem', *IEE Transactions*, vol. 20, pp. 223-229, 1998.

[Beardwood59] Beardwood, Halton and Hammersley, 'The Shortest Path Through Many Points', *Proceedings of the Cambridge Philosophical Society*, vol. 55, pp. 299-327, 1959.

[Jaillet93] Jaillet, 'Analysis of Probabalistic Combinatorial Optimization Problems in Euclidean Spaces', *Mathematics of Operations Research*, vol. 18, pp. 51-71, 1993.

Further reading

Archimedes, *On the Measurement of the Circle*, c. 250-212BC.

Basel and Willemain, 'Random Tours in the Travelling Salesman Problem: Analysis and Application', *Computational Optimization and Applications*, vol. 20, pp. 211-217, 2001.

Clay Mathematics Institute 'Millennium Problems', http://www.claymath.org/millennium.

Hoffman and Padberg, 'Travelling Salesman Problem', *Encyclopedia of Operations Research and Management Science*, Gass and Harris (Eds.), Kluwer Academic, Norwell, MA, 1996.

We made a mistake...

In 'The Model Student: The Regular Travelling Salesman – Part 1' (*Overload* 171), we said that Archimedes proved that the ratio of the circumference of a circle to its diameter was between $^{31}/_{7}$ and $^{310}/_{71}$. He didn't. He proved it was between $3^{1}/_{7}$ and $3^{10}/_{71}$.

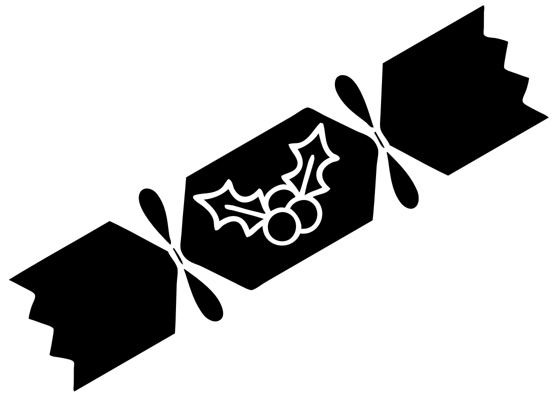
Thank you to Gary Taverner for pointing out the error – but unfortunately we were too late to correct the printed version of *Overload*.

Dr Richard Harris died over the summer, and this article has been republished as a tribute to him, following on from the article in *Overload* 171.

For more information about Richard, his work and his legacy to the software industry, see 'The Model Student: The Regular Travelling Salesman – Part 1' in *Overload* 171.

Afterwood

Pun and Dad jokes are lots of fun. Chris Oldwood git-pull's a cracker.



As the Earth closes in on another complete loop of the Sun and one more season of this journal comes to an end we first need to pass through the year's final major holiday – Christmas. This is a magical time of the year for both children and adults as that jolly, larger-than-life fellow in a red suit pays us a visit. Sadly, some of the mystique surrounding how he manages to circumnavigate the globe in such a short period of time has been dispelled due to the event being live streamed by NORAD. Santa's big mistake, like so many of us Internet users, was to accept cookies and now his every move is being tracked. When Google first revealed its 'map/reduce' technology I wondered if industrial espionage might have allowed them to expose one of Santa's biggest secrets, but his tech still remains safe for the time being, although I do wonder if their original motto of 'Don't be Evil' was simply a ploy to get on his good side. One thing's for sure, Santa must be a big fan of The Gang of Four as he takes the Visitor pattern very seriously.

For those of us in the UK, there is the annual disappointment of hoping for a 'white' Christmas despite knowing full well that the changing climate has probably put that out of reach for the foreseeable future. Maybe if you can find a couple of ageing mainframe programmers and can antagonise them with a fiendish text manipulation problem you might provoke a SNOBOL fight. Of course, baiting people is not going to earn you a place on Santa's more favourable list, and you can't take a leaf out of the Linux playbook and simply invoke yourself with 'nice' – you just have to get on with actually being nice. Some parents try to incentivise their children over the festive period, but we've always preferred the long game; the only ELF you'll find on our shelves lives in the library as a chapter in a book about binary file formats.

If you look closely enough, Christmas is a time of data structures: lists, maps, and those all-important trees. Santa's choice of a list for the containers of who's been naughty and who's been nice is certainly a curious one, although if there is one data structure that has wildly varying characteristics depending on which programming language you choose it's the humble list – it might be singly linked, doubly linked, or even array-like. With billions of people to manage, I can only imagine he uses Big HO notation to choose his implementation wisely. I suspect the reason he checks it twice is due to all those pointers and the need for an address sanitizer. Either way he must be storing our names using narrow strings because it's a time for no `L""`.

While lists might be the focus for Santa, us mere mortals have trees to contend with. Every year, December starts with the difficult task of choosing a tree, but then, even more importantly it needs to be decorated. If there is one thing you can never get agreement on it's how best to traverse it: pre-order, in-order, or post-order?! Being the impetuous sort, the kids like to visit the leaves too early by plastering them with tinsel meaning that the lights have to be surgically inserted later. Despite favouring a trunk-based approach, I'm not afraid to admit that feature branches have their place too.

Irrespective of how much effort we put into the upper regions of the tree, it's Santa who is responsible for most of what lies around the base. Much like the role of an Enterprise Architect, he does little of the work himself, preferring instead to farm it out to the little people. Also like an Enterprise Architect, you can always spot those presents he handled himself because of the excessive amount of wrapping. I've always felt elves would probably make good C# and Java programmers due to their expertise with boxing, although many are probably destined to work at Microsoft in the Office team as they also seem obsessed with ribbons. At least we haven't reached the point where requesting presents from Santa has degenerated into raising a JIRA ticket.

By the time we reach our Christmas lunch, Santa will be back home and resting after rushing around the globe grappling with time-zones. (Dates are a popular festive snack too though fortunately they only come in two formats – pitted or unpitted.) Lunch in the UK typically consists of turkey, although GOOS is popular with the TDD crowd, along with a varied selection of trimmings, such as credential stuffing for those in the infosec business. They aren't the only ones battling with crackers though, as everyone gets to partake in wearing a thin paper crown and reading out a pitiful Christmas themed joke. For those of you who have never had the (dis)pleasure of pulling crackers, let this episode of 'Afterwood' be my present to you.

Merry Christmas and a happy New Year! ■

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 emails and DMs to gort@cix.co.uk or @chrisoldwood

SEASON'S GREETINGS FROM ALL AT ACCU

Join ACCU

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

- A worldwide non-profit organisation
- Journals published alternate months:
  - *CVu* in January, March, May, July, September and November
  - *Overload* in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website

professionalism in programming

www.accu.org

To connect with like-minded people visit accu.org

accu