OVERLOAD 166

December 2021

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Yi ZhU
on Unsplash.

Copy deadlines

All articles intended for publication in Overload 167 should be submitted by 1st January 2022 and those for Overload 168 by 1st March 2022.

The ACCU

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

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

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

It's not normal

We live in strange times. Frances Buontempo asks if everything's OK.

As 2021 draws to a close, a suitable editorial topic would be reflection on the year or making predictions for the following year, but I am in denial that 2022 is nearly upon us, so we'll have none of that. Such an approach may be traditional, but why would I let what's perceived to be normal drive my actions? My Dad always used to remind me that normal people don't have two legs. Why? No-one has more than two, some have fewer, and so the 'average' or normal number isn't two. If you were normal in every way, you would be very different indeed. Besides, an aim of mediocrity is a strange target. Trying to be better than average seems more aspirational. However, if you are no good at something, in my case some exercises in a gym class, then simply managing to get better than you were last time you tried is great. Improvement matters: comparing the actual outcome if you are far worse than average may tempt you to give up. The measurements you choose can impact the outcome, so in order to gamify any activity, you need to pick the right game.

Now, there are circumstances where knowing an average is helpful. We know the average human's body temperature, within a range. If yours is outside that range, you might be in serious trouble. Knowing what's normal can help in some circumstances, therefore we need to know how to spot what's normal. Frequently this requires data gathering, involving measuring or counting, though sometimes we notice things are amiss before gathering numbers to analyse. If my PC starts making an odd noise, something may be wrong. I suspect we unconsciously register what is ordinary, so that even without numbers or metrics, you can spot when something is off. This spidey sense of something being up comes from noticing anything out of the ordinary. Experience can help you spot reasons for apparently weird behavior, like fractions in code giving what are often termed "floating point 'errors'" or a recursive function causing a stack overflow. In contrast, if you are trying something completely different, like a new gym exercise or a new baking recipe, you have no prior experience to go on. Even without having previous attempts to remember, a bone going crack or a burning smell usually indicates a problem. You can still hazard a guess about what to expect based on different, but related, experiences. Trying to vocalise your concerns can be difficult, since learning precise language to describe a novel situation takes practice. Having an experienced person on hand to say "Looks good!" or "No, stop!" is useful. The feedback may not tell you how to improve, but can be enough to nudge you somewhere better. Now, supervised machine learning and AI uses feedback functions as similar nudges: we tell the machine if the generated solution is good or not. The algorithm uses this feedback to inform further attempts, so that the AI may appear to learn something. I suspect this is similar to the way many of us learn, but we have the advantage of being able to question instructions and ask for help.

Whether AI can ever think like a human is an unanswered question. Turing invented the so-called Turing test to avoid needing a definition of thinking and AI is a big topic, so let's change our focus. I talked about measuring and counting to decide what's normal. Some basic arithmetic, for example a back of the envelope calculation, will indicate ball park figures. If you want to know how long a program might run for, then knowing your clock speed and number of calculations to perform gives a good enough guestimate to determine whether your program has got stuck or not. Of course, you could debug and find out what it's actually up to, unless it's running on a machine you can't access. If your program runs regularly as a batch job, keeping stats on how long it usually takes is useful. Knowing the average time taken is one thing, but the variance is useful too. Running the same program with the same inputs is unlikely to take exactly the same amount of time, or even RAM, or whatever else you want to track. If anyone tells you an average, ask them for the variance or standard deviation too. And also ask how many data points were used. If I measure once, and find my code takes 120 seconds to run, the average is 120s, the variance is 0, and though this is a data point, it is only one data point. Ask how many samples were used. If you learn enough statistics, you can decide if the sample size is large enough using maths, but even without the detailed knowledge, very few data points might not be enough to go on. It has been said "There are lies, damn lies and statistics." Whether Mark Twain, Benjamin Disreali or someone else first said this, we'll probably never know; however, ensure you are told the variance and sample size and you might spot untruths.

What kind of things do you measure? Aside from the performance of the programs you create, you might be tracking compile and link times as you code. Perhaps these seem relatively stable, but after a year or so, a tiny but creeping slow-down may have moved from almost imperceptible to totally unacceptable. We all have different levels of patience, but I once worked on a test suite which ran in about five seconds. Five seconds is just under the limit of my attention span, before I get distracted and forget what I was doing. Initially, there were no tests, so they took no time at all. Gradually, we added more tests and they took a little time, but not too much. I was keeping an eye on coverage to check I'd hit a lot of the code, but the time taken mattered to my mind. If a test suite takes a long time, people might not bother running the tests before committing code, and no one wants a broken build. Because I was tracking the time, even though it did gradually creep up, I spotted when a test went from running in microseconds to taking a second or so. Many might mock, but the implementation of a function had been changed, and if this had run in a tight loop in production our batch would have missed service level

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.

agreements and much trouble would have ensued. Catching the small change early saved the day.

A dynamic system, such as a growing test suite, might have a 'moving average'. As you measure, things change. Though I said earlier to check the variance and sample size as well as the average, life is, as ever, more complicated. Ask yourself if the average is changing over time. Is it trending up? Then your build times, test suite or batch job might breach some limits in the long run. Maybe the average cycles, going up then down, following what is known as a seasonal pattern [Kenton20]. Do you know what's driving the change? Maybe many people do a code commit a Friday, forming a backlog on build servers, so things slow down at the end of the week. A batch processing job may be longer at the weekend or the end of the month, because more reports are generated then. Alternatively, you may have a mystery slow down once in a while. Keeping stats and plotting graphs can give you new viewpoints on problems. An initial gut feeling or back of the envelope calculation as a starting point is ok, but numbers give you so much more. This may help you track down mystery abnormalities and find the root cause.

As averages can change in cycles or follow trend lines, data in general can fall into one of many statistical distributions. The so-called normal distribution is commonly assumed. This is the famous bell curve also known as the Gaussian distribution. Gauss used the word normal to describe the distribution, but in the sense of orthogonal or at right angles. As a blog puts it, "The term comes from a detail in a proof by Gauss … where he showed that two things were perpendicular in a sense." [Cook08] If you want to know which two things and in what sense, you'll need to go research. Wikipedia tells me,

> by the end of the 19th century some authors had started using the name normal distribution, where the word 'normal' was used as an adjective – the term now being seen as a reflection of the fact that this distribution was seen as typical, common – and thus 'normal'. [Wikipedia]

Collecting data and fitting regression lines to spot patterns and trends has a long history. Recently we have turned this up to the max. It's very difficult to go anywhere near the internet without leaving ripples or a digital footprint. You will gradually collect cookies, unless you are very careful. Claims that this enables targetted marketing are used. Given a few of the items certain social media sites try to sell me, I'm not certain of the reasoning for the targets, to be honest. Maybe the internet has gotten out of hand. I noticed a recent MIT press book claiming,

> this has not always been the case: In the mid-to-late 1990s, when the web was still in its infancy, 'cyberspace' was largely celebrated as public, non-tracked space which afforded users freedom of anonymity. How then did the individual tracking of users come to dominate the web as a market practice? [Kant21]

How did this happen? By coders writing code. Why did it happen? Because some people think collecting as much data as possible might help make money. Don't get me wrong, the internet is a tool that can be used for good or harm, like many things. Being tracked online has become normalized now, though you can maintain some anonymity, sometimes.

This brings us to another use of the word 'normal'. In order to do data science or statistics, you often need to scale data so it lies in the same approximate range. We call this normalization. If you have two features that differ by orders of magnitude, maybe height and shoe size, the larger quantities can overshadow the smaller ones. Scaling so our numerical values are similar, levels the playing field, as it were, makes it easier to spot trends. Being programmers, we also normalize strings to deal with various diacritic marks and similar. Furthermore, we can do this in several different ways: canonical decompositions, compatibility decomposition and one of these combined with composition afterwards [MDN]. This means we also define canonical equivalence, which is not to be confused with normalization. I thought numbers could be hard work until I discovered strings.

So, what have we learnt so far? Normal doesn't mean normal and we are still none the wiser as to what normal really means. We do talk about conventions as normal, like running tests before a commit, warm up before exercising etc. Many habits we are encouraged into, like brushing our teeth before bed, do have obvious benefits. Running tests locally as you write code, or even checking it compiles can save you a world of pain in the long run. However, conventions do vary and when this happens accusing someone of not being normal because they do things differently is unacceptable. I tend to use a bookmark as I read down a page if I'm using a real life book or paper print out. Some people mock me for this. It helps me concentrate and stops the letters moving about all over the place. Some may say I have dyslexic tendencies, so first I apologise if my writing is littered with typos and second, using a bookmark helps me, so don't judge.

We are all different, and that's a good thing. Doing things differently can lead to innovations. Some programmers are told there are not normal – accused of being geeks or nerds [Buontempo21]. I say hooray for geeks, people who myopically collect details, measure and figure out what's going on. Without us, the world would be very hard to navigate. Don't strive to be normal, it's far too mediocre. As I draw to a close, I notice a relevant tweet to end on [Kazum93]:

> Normal people on their weekend: Chill, Netflix
>
> Simon: Let's create a memory allocator in C++

Thank you @kazum93, I couldn't have put it better myself.

References

[Buontempo] Frances Buontempo (2021) 'Geek, Nerd or Neither?' *Overload* 163, June 2021, available at https://accu.org/journals/overload/overload163

[Cook08] John D. Cook, 'Four characterizations of the normal distribution', published 13 March 2008 on https://www.johndcook.com/blog/2008/03/13/four-characterizations-of-the-normal-distribution/

[Kant21] Tanya Kant, 'A history of the data-tracked user', published 8 October 2021 at https://thereader.mitpress.mit.edu/a-history-of-the-data-tracked-user

[Kazum93] Kazum93 on Twitter, available at https://twitter.com/kazum93/status/1454387344031817732

[Kenton20] Will Kenton, 'Seasonality', updated 30 November 2020, available from https://www.investopedia.com/terms/s/seasonality.asp

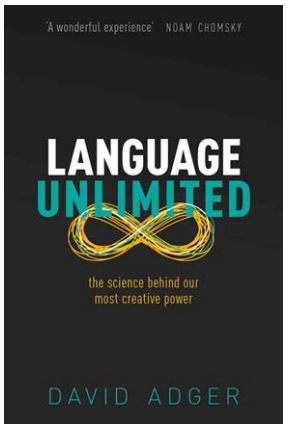
[MDN] `string.prototype.normalize()` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/normalize

[Wikipedia] 'Normal distribution', section 7.2 ('Naming'), available from https://en.wikipedia.org/wiki/Normal_distribution#Naming

Programming Language Unlimited

How programmer-friendly are programming languages? Lucian Radu Teodorescu considers how principles from linguistics might allow us to read code with ease.

You walk into your favourite bookstore. Your attention is drawn by a book with good cover design, the title sounds intriguing, you flick through it and find the content interesting, you smell the book. You decide to buy it. This is precisely what happened to me at the beginning of this year when I found *Language Unlimited* by David Adger [Adger19] (actually, I abstained from smelling the book because of the Covid-19 pandemic).



For years, I have been interested in proper linguistics, but have never taken the time to dive into it. I still remember the times my wife shared the great ideas from the General Linguistics courses that she was attending while in college with me. One of the ideas that profoundly fascinated me was that language structures our thinking (i.e., Linguistic relativity principle [Wikipedia]), as first put forward by Wilhelm von Humboldt. Later, for a few years, I worked on a text-to-speech project and then on an automatic-speech-recognition project, and they gave me an opportunity of seeing language from a different perspective. It was during these years that I started to look with awe at the complexity of human language; I began to think that if we ever fully understood language, we would fully understand the human brain (making computers understand language equates to making them think); since then, I have relaxed my beliefs on the subject, but I still feel that is largely true.

Therefore, buying this book, was a perfect opportunity for me to brush up my knowledge on general linguistics. Besides offering a lot of linguistics information, the book also contains some gems that can be used in Software Engineering.

Main ideas from Language Unlimited

The book starts by explaining that virtually every statement (especially the more complex ones) is novel, i.e., we haven't heard it before. Think about it; what is the probability that you heard the previous sentence before? Close to zero. And yet, it is easy for us to understand it. This suggests that our mind has a *structure* that allows it to understand (and produce) language; language is not learned directly from experience. Humans are born with some ability that Noam Chomsky calls *Universal Grammar*, which allows us to easily process language.

The book revolves around three main ideas: First, that human language is organised in a special way and cannot transgress some boundaries (imposed by human biology). Second, that language is organised hierarchically. And finally, that macrostructures in the language echo the smaller structures that they are built from (i.e., language has some fractal properties).

Languages are built hierarchically. There is no language in which grammar rules are based on sequentiality (e.g., next noun, next verb, 3 words to the right, etc.); all languages are built out of hierarchical structures. Grammars for natural languages are context-free grammars, a term that we study in Computer Science[1].

The book describes numerous experiments that show how the language is inherently hierarchical, and deeply rooted in our human nature. From examples of deaf people who create structure in their own invented sign language, to examples of MRI scans performed on the newly born babies that reveal how we have innate structures for processing language. Although other animal species have better abilities than humans to listen to sounds and good abilities in statistical learning, they can't structure language like we do. The human mind is hardwired for a particular kind of language acquisition; this is what Chomsky calls *Universal Grammar*; different languages appear as particularisations of this Universal Grammar.

The human languages seem to revolve around the distinction between verb and noun. Moreover, the concept of grammatical Subject, which seem to be an imperceptible property of languages, is central to the construction of a language. And, the distinction seems to never be based on meaning. For some languages, classifiers seem to be important in the distinction between nouns and verbs.

The book also briefly covers the *Merge* process, introduced by Chomsky in the 90s [Chomsky95]. This process appears to be used in all human languages. If we have two (compatible) units of language, then we can group them together and form another language unit. This grouping can be done hierarchically until the entire phrase becomes a language unit. If this process is deeply embedded in our brain, then this will explain why all the languages are hierarchical. This seems to provide the upper-boundary for the limits of the languages that we can have in practice.

As an example of Merge, one can look at the classic *Subject Verb Object* sentences. In the sentence *Alice sends a message*, *Alice* is the Subject, *sends* is the Verb and *a message* is the Object. Here, *a message* is a noun phrase created using Merge from determiner *a* and noun *message*; this phrase is created by Merge from its two constituents. Thereafter, the phrase *sends a message* can be formed by Merge from the verb and the last phrase. Adding Subject to the newly formed phrase, we obtain the complete sentence.

This Merge process makes the larger structures of the language similar to the smaller structures of the language (like a fractal). For programmers, this means that each sentence can be represented as a binary tree. The binary tree might be slightly complicated in some cases (i.e., with what we can call symbolic links between the nodes), but nevertheless, a relatively simple structure.

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

1. Interestingly enough, the concept of context-free grammars was invented by Noam Chomsky, the prominent linguist behind most of the main theories of languages advocated in the book. Chomsky didn't just make major contributions to linguistics, but to Computer Science as well.

if there is one trait that we want our code to have in common with natural language, that would be the ability to process it easily and very fast

Thinking fast language

The entire book talks about how the human brain is hardwired to process language. We are born with *Universal Grammar*, and, based on the language we hear (or see) around us in early childhood, we constrain this into an actual language. It's similar to how we are born with the ability to move our body, and then we learn different types of movements (standing, walking, running, various athletic movements, etc.).

Similar to how we can walk without thinking about it, we can process language without having our attention to the structure of the language itself. This is an automatism in on brain. This leads us to the distinction found in Daniel Kahneman's famous book *Thinking, fast and slow* [Kahneman11]: our brain can be thought of as operating in two modes: one fast, called *System 1*, and one slow, called *System 2*. In the first mode of operation, our brain has the ability to respond quickly to inputs, and it does this in an automatic manner, without requiring our attention. In the second mode of operation, our brain acts only when we pay attention to a particular situation, and typically is associated with complex computations.

Language processing is fast, it's performed by *System 1*. On the other hand, processing mathematical sentences is typically handled in *System 2*, it's slow, and it requires our attention.

If we think about language as the means of communicating (not necessarily between humans), then the code that we have in Software Engineering is also language. It is a special type of language, but can still be language[2].

And, if there is one trait that we want our code to have in common with natural language, that would be the ability to process it easily and very fast. That is, read code without focusing our attention on it, just like reading a novel. This might be a goal that we cannot achieve, but still, the closer we get to it, the better. After all, programming is a knowledge acquisition process (see [Henney19]); the better we are at *reading* the code, the more attention we can divert to *understanding* and *reasoning* about the code.

In this context, it makes sense to look at natural language for inspiration when we design programming languages. The more we can create the structures in programming languages to be compatible with our brain's ability to process language, the better. Thus, we arrive at the following goal:

> Programming languages should be designed in such a way that the code written can be easily processed by our innate language abilities – the programming language should be a specialisation of the Universal Grammar.

It then makes sense to look at how natural languages are constructed and draw some conclusions that might be applied to programming languages.

2. Throughout the article I'll use the term *programming language* to mean generically code, i.e., the information encoded according to some predefined rules. This is similar to how we use the term *language* to express a set of sentences that convey some information (e.g., a text written in English), without referring to a particular set of syntactic and semantic rules (for example of English).

The different levels of language

Let us think for a bit what the English language means. If we think about the spoken language, then there is a phonetic aspect of the English language that is exercised in speech. If we think about the written text, then there are lexical rules in the language. Both oral and written English have some syntactic rules that constrain how words can form sentences. On top of the syntactic rules, we have semantics, which tell us what different words mean. The semantics and the syntax of a language overlap, but, for simplicity, let's consider them as being completely distinct.

We also have the distinction between lexical rules, syntactical rules and semantic rules in programming languages as well. We are going to briefly look at some aspects of these levels for natural languages so that maybe we benefit in programming languages.

Let us consider the phrase: *Winston Churchill was the best president that Unitd Kingdom had.*

Most of the readers can easily understand all the words in this sentence; even if one word contains a typo. We usually don't focus on the lexical part; we just read the symbols and fill in the gaps. We are using the fast part of our brain (*System 1*, to use Kahneman's terms).

Similarly, most of the readers will be able to process the structure of the sentence with no problem. We unconsciously identify what is the Subject of the sentence, what is the Verb and how different words are connecting with other words, forming the structure of the sentence. And most probably, nobody thinks about the Subject when reading this sentence. The parsing of the sentence is done automatically by our brain, without needing our attention. Again, syntax processing for natural languages are done in the *System 1* mode.

Things become more interesting when we look at the semantic level. Here, a lot of the readers will probably treat the sentence with increased attention, i.e., using the slow and analytic *System 2*. Part of the reason for doing that is that UK doesn't have a president, and part of the reason is that claiming that somebody is the best PM of UK is highly debatable. In general, for sentences containing some novelty, we tend to involve the analytic part of the brain.

Looking back at programming languages, it would be nice if we could design them in such a way that lexical and syntactic processing is always handled by our fast brain, while leaving the semantics to the analytical, slow brain. The *understanding* and the *reasoning* about the code are typically much more demanding than understanding regular English texts. Thus, it's important that our entire attention goes to these processes, which means we should not require the programmer to divert attention to lexical and syntactical processing of the code.

We have yet another argument that syntactic processing needs to be done with the fast brain, and thus we should try to create programming languages to model the *Universal Grammar*.

The infamous for-loop

I've written before how the classic 3-clause `for` loop is not programmer-friendly [Teodorescu21]. But, at the time, I'd argued that using this `for`

Translating this to programming languages, we cannot naturally have lists of elements in which some elements (i.e., the first) has a special meaning compared to the rest

loop is bad because of reasoning complexity. The idea is that, to fully understand what the `for` loop does, one needs to perform a significant amount of reasoning; significantly more than the reasoning one needs to perform for a ranged `for` loop (20 steps versus 8).

Here, I will move the argument to the next level. This structure is not user-friendly because it doesn't follow the structure of a natural language. This means that, most probably, our brain needs to focus when reading the `for` structure. It is precisely what we said above we want to avoid.

For example, let's look at the follow classic `for` loop:

```
for (int i=0; i<100; i++) console.print(i);
```

It's hard to fit this into our patterns of reading natural language. Let's try it out: *for int i equals 0 <pause> i is less than 100 <pause> i plus plus <pause> console print i*. This is hard to process by humans. It is like reading a mathematical statement, thus it most probably requires *System 2* to process it. It cannot be similar to natural language.

On the other hand, let's take a look at the ranged `for` loop (written in an imaginary programming language):

```
for (i: 1..100 ) console.print(i);
```

This can be processed easier by humans; one can say this out loud: *for i in range 1 to 100 <pause> console print i*. This starts to sound like regular English (not quite there, but close). It means that it can be processed by the fast part of our brain, without requiring our attention.

A simple grammar test

OK, so we have decided that we shall try to create the programming languages to be processed by the fast part of our brain. Do we have a measure to see how close we are to our goal?

Approaching this from a linguistic perspective is the right way to go, but probably that is an overkill for most of us in Software Engineering. But, there might be a trick to make things much easier.

Similar to how in linguistics we use various tests to check the constituency of a phrase (probably the most known is the substitution test), we can have a test that will easily tell us if the structure of the programming language corresponds to our innate structure of the brain responsible for processing language. And that test is simply reading the code aloud.

> *One should be able to read aloud a piece of code and have it sound like natural language.*

This is what we've done above for the classic `for` loop to prove that it doesn't quite sound like the languages humans are accustomed to; and we've also applied the same test to show that the ranged `for` loop can be made to sound like natural language.

On sequentiality

Language Unlimited reveals us a fact about natural language organisation, that for us, software engineers, might sound surprising.

Apparently, there isn't a single language found on Earth in which sequentiality plays a structural role. Languages do not have rules of the kind 'rule *X* applies when a word or type of word is followed by another word or type of word'. For example, one might imagine a rule in which grammatical *Agreement* works between a noun and the following verb; but this is never the case; one can always insert a structure containing nouns and verbs in between the two words for which we apply the *Agreement*. Similarly, there are no rules that rely on counting (three words to the right, etc.)

Language is always hierarchical.

Even when we have enumerations, language is hierarchical. Let's take the following phrase as an example: "Alice, Bob and Carol talk loudly". We have a hierarchical structure that looks like Figure 1.

Here, the first "and" is mute, while the second one is audible. Even if we naively think that we have a sequence of words in the enumeration, our innate abilities process that as a (binary) tree structure.

The point is not that our brain cannot process enumerations fast, but that the elements of the enumeration are processed as being homogeneous. In the enumerations that we find in natural language, we can't find the first term to have a special meaning compared to the rest of the terms.

Moreover, because of *Merge*, our brain can assimilate the whole enumeration as one language unit. Thus, we can easily substitute the enumeration by one word or one phrase. For example, the following sentence is equivalent: "*They* talk loudly"; we've replaced the enumeration "Alice, Bob and Carol" with the pronoun "they".

Translating this to programming languages, we cannot naturally have lists of elements in which some elements (i.e., the first) has a special meaning compared to the rest. This means that a Lisp command line `(write a b c d e)` is not necessarily easily parsed by our brains. Similarly, Haskell's

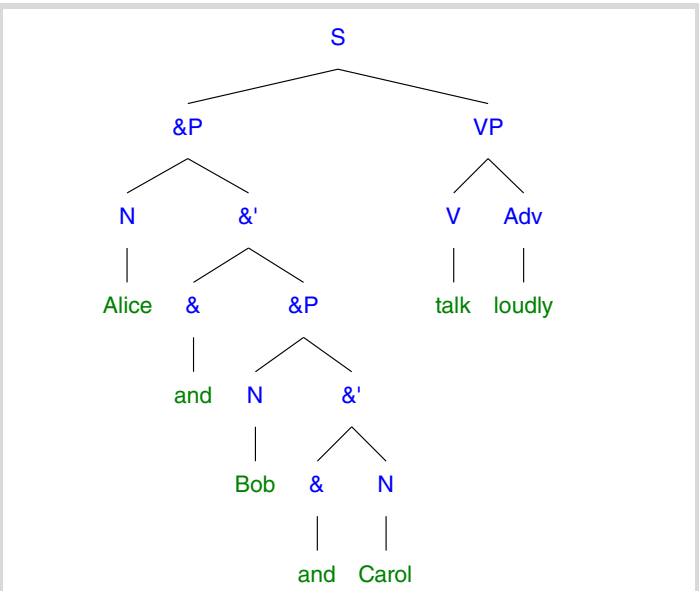


Figure 1

functional languages seem to be further away from natural language statements; they are closer to mathematical formulation

syntax `func arg1 arg2 arg3 ...` doesn't necessarily play nice with the fast part of our brain (especially when having more than 2 arguments).

The syntax of a sentence

Most phrases in English contain a *Subject* (e.g., doing some action), and a *Verb* (e.g., expressing the action done by the *Subject*). There is often another term, called *Object*, which typically represents the object acted upon. For example, in the sentence "Bob drinks wine", "Bob" is the *Subject*, "drinks" is the *Verb* and "wine" is the *Object.*

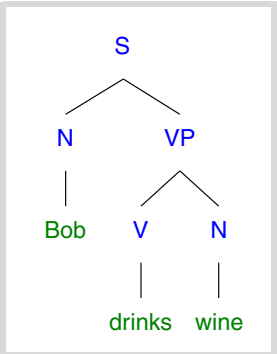


Figure 2

The difference between the *Subject* and the *Object* is their position in the hierarchy; the *Object* is always merged with the *Verb* (see Figure 2).

English is a *Subject Verb Object* language; but not all the languages are like this. For example, Japanese is a *Subject Object Verb* language. There are also some languages that have *Verb Object Subject* ordering, and recently linguists also found some languages that have *Object Verb Subject* ordering. While there are some languages that have the ordering of *Verb Subject Object*, these can be explained by a more complex set of rules that involve duplicating the verb and silencing one occurrence (something similar to what English does with auxiliaries).

When designing a programming language, we have to pick a convention. For example, let us pick the *Subject Verb Object* order as in English. In the following paragraphs, we will be focusing only on the structure of expressions, ignoring other control structures (`if` clauses, loops, etc.).

In the most common case, if we ignore any punctuation, our programming language sentences should be of the following form:

```
subject operation argument
```

This looks extremely similar to the notation in Object-Oriented languages. The only difference is that we typically have a dot between `subject` and `operation` and, depending on how we look at it, parentheses around the `argument`. That is: `subject.operation(argument)`. Now, if we have multiple arguments, they can be similar to an enumeration. That is, one can think of the whole notation being `subject.operation argument`, where `argument` can be written as `(arg1, arg2, ...)`.

I'm not actually advocating here for intense use of OO languages, but in terms of syntax, the common practice for expressing a basic statement seems to be close to statements in natural languages. On the other hand, functional languages seem to be further away from natural language statements; they are closer to mathematical formulation, which tends to make our brain use the slower *System 2* part of our brain.

An OO statement like `plane.fly()` makes sense syntactically, and it can be easily spoken, too. Similarly, the following statement makes sense syntactically and again can be spoken relatively easily too: `rectangle.draw(context)`.

However, I'm not necessarily satisfied by the OO solution either. While a plane can fly, a rectangle cannot draw. Especially, a rectangle doesn't draw a context. A rectangle can only be drawn by somebody else, i.e., the *Subject*. But what can be this subject? I think the only reasonable assumption is that the *Subject* is the actual system that executes the program. Thus, a more appropriate statement would be `System.draw(rectangle, context)`, read as "System, draw the rectangle in the context".

The more we think about this assumption, the more it makes sense for most of the actions in an OO program to be of the form `System.action(argument)`. But, if we repeat `System` all over the place, it is maybe better to just remove it and have it implicitly there, only when we speak. Thus, we transform our statements into `action(argument)`. This is similar to how this is spelled in most of the functional programming languages: `doSomething arg1 arg2 arg3`; we would read this as "System do something with arg1, arg2 and arg3". This sounds a bit better.

Starting from this, we can develop a large analysis of the possible conjugations for verbs in natural languages and how they can translate to programming languages. We could also discuss the grammatical agreement. But, unfortunately, that falls outside the scope of this article.

In natural languages *Subject*, and the distinction between *Subject* and *Object*, seem to be an important part of syntax. In programming languages, the notion of *Subject* doesn't seem to be too well-defined. I hope that we can make some progress in programming languages for better isolating the *Subject* from possible *Objects*, and with that make the programming language more natural.

Coming back to the structure of phrase, most of the phrases in natural languages are much more complicated than just the 3 terms *Subject Verb Object*. Each of these terms can have sub-structures. As David Adger explained, the high-level structure of a phrase is similar to the low-level structure of the phrase; it's *Merge* all the way down.

Instead of a noun we can put structures that contain determiners (e.g., "the chair"), adjectives (e.g., "red chair"), or we can put entire phrases (e.g., "*The man who sold the world* was caught by the authorities"). Similarly, we can have adverbs near verbs ("he spoke quickly and loudly"), and entire phrases instead of verbs (e.g., "She read a book in bed before going to sleep"). One interesting case of nesting is the possessive chains, which can go on forever (at least in English); consider for example "My mother's brother's wife's book was lost".

We often have this nesting in programming languages too. The possessive chains are often used in OO languages (ex: `a.b.c.fun()`), but they can be found in other types of languages too. Replacing objects with expressions is almost universal in programming languages. Replacing verbs (i.e., functions) with complex statements is also very frequent in programming languages (ex: using lambdas instead of functions).

Sentences in Sparrow

For years, I worked on a programming language called Sparrow [SparrowRepo] [Teodorescu15]. I wanted to create a language that integrates efficiency, flexibility and naturalness, and the central feature of the language was (paradoxically) static metaprogramming. For a long time, Sparrow was mainly an OO imperative language, but later on started to migrate towards being more functional (the transition was not complete).

Borrowing from Scala, Sparrow has an interesting syntax for expressions. Coupled with the use of ranges, it creates some nice possibilities for expressing some algorithms. Expressions in Sparrow have two forms:

- **`subject operation`** – postfix notation for unary operations
- **`subject operation object`** – infix notation for binary operations

In both cases, no actual punctuation is needed. The operation can be an operator or a simple function name. Moreover, the operation has name lookup rules that will search near the given **`subject`**.

Let us take an example (actually taken from [Teodorescu15]). Let's compute the sum of squares for all the odd numbers belonging to the first *n* Fibonacci numbers. This is achieved in Sparrow by the following one-liner (assuming functions **`fib`**, **`isOdd`** and **`sqr`** are already present):

```
1...n map \fib filter \isOdd map \sqr sum
```

No other punctuation is actually needed (previous versions of Sparrow required a semicolon at the end of the sentence, but the last one doesn't). The backslash is used to transform a function name into an object. This line contains the following operations: **`...`**, **`map`**, **`filter`** and **`sum`**.

Reading this line from left to right, it sounds like: "the inclusive range from 1 to **`n`**, mapped through **`fib`**, filtered by **`isOdd`**, mapped through **`sqr`**, then summed". Reading this from right to left, it sounds like: "the sum of the squares of all odd Fibonacci numbers generated from range 1 to *n* (inclusive)". In both cases, it sounds relatively well in English.

Let us take another example of the same kind. Let's compute the root-mean-square of the lengths of all the Collatz sequences up to the first one that has a length greater than or equal to 500. Given a natural number, a Collatz sequence is a sequence of numbers starting with the given number, and repeatedly applying a transformation until we reach 1; although for all known starting numbers the Collatz sequence is always finite, the computer cannot know that, which makes the problem especially interesting. This problem can be solved in Sparrow by the following one-liner:

```
(1..) map \collatzSeq map \rangeSize takeWhile
(fun s = s<500) rootMeanSquare
```

Here, the structure **`(fun s = s<500)`** is a lambda function. This can be read from left to right in the following way: "the infinite range starting from **`1`**, mapped through **`collatzSeq`**, mapped through **`rangeSize`**, taking elements while the number is less than 500, and apply **`rootMeanSquare`** for all these elements. Again, this can be read relatively easy.

Looking at this problem in more depth, we start with an infinite range. For each element in that range, we generate a range that is potentially infinite. We reduce these to a finite sequence of numbers by mapping through **`rangeSize`** and by calling **`takeWhile`** with an appropriate predicate. This one liner is pretty complex for its succinctness. Moreover, the tests I've done on this form proved that this can be as efficient as writing imperative code (actually, it was faster than traditional code, probably because it exposed some optimisations to the compiler). So, this one-liner is the most efficient implementation of the (no trivial) problem, very succinctly (shorter than the actual program description) and with a syntax that can be easily be read by humans.

I was pretty happy with the results after finishing the design of the syntax in Sparrow. Now, after some time in which I haven't worked in Sparrow, with the new focus on linguistic structure, I find the results to be even better.

I am not arguing that the syntax of expressions in Sparrow is the best one, and all programming languages should use something similar. For that, we do need a more in-depth analysis. The point I'm trying to make is that we can find syntactic forms that will be easier to read (and to process) by humans. There are ways in which written code sounds natural, and the programmer can process the syntax with the fast part of the brain, while focusing the analytic part on the code semantics.

What can be done next

A collaboration between programming language designers and experienced linguists, and possible neuroscientists would be highly beneficial in order to design programming languages that require only *System 1* for processing the syntax of the code.

David Adger brings in his book a lot of evidence that originates in experiments involving MRI scans for humans exposed to language. And, of course, that leaves us pondering whether we can do the same thing for programmers.

If we can have MRI experiments that would show how different sentences expressed in different programming languages are read by programmers, then we can compare different programming languages and different syntactic rules from a naturalness perspective. We could probably find a ranking between different types of syntactic structures. Having that, we can create programming languages that fully exploit the innate structures in our brain and allow us to read code with ease, similar to reading natural language.

Conclusions

Inspired by David Adger's book, *Language Unlimited*, this article tried to question how programming languages should be designed to be as close as possible to natural languages. The article doesn't attempt to provide any answer, but just explores different aspects of language, with the hope of having a first attempt at drawing the space of the problem.

Besides this, the article tries to argue that all programming languages should have a (new?) goal: to make the syntax similar to human language, with the same structure, so that the human mind processes the syntax in the fast mode, leaving the programmer to direct their attention on semantics.

If we achieve this goal, then maybe programmers might start to immerse in the programming language, similar to how people are plunging into language since the day they are born. We can then speak of *programming language* as the totality of code that can be written and easily understood, as the sea of structures that shapes how we think programming; similar to the way we use the term *language* to mean a fundamental part of the human existence.

Only then we can fully unleash our creativity in programming. Only then we can have *programming language unlimited*. ■

References

[Adger19] David Adger, *Language unlimited: The science behind our most creative power*, Oxford University Press, 2019

[Chomsky95] Noam Chomsky, *The Minimalist Program*, MIT Press, 1995

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

[Kahneman11] Daniel Kahneman. *Thinking, fast and slow*, Macmillan, 2011.

[SparrowRepo] Lucian Radu Teodorescu, 'The Sparrow programming language', https://github.com/Sparrow-lang/sparrow

[Teodorescu21] Lucian Radu Teodorescu, 'How We (Don't) Reason About Code', *Overload* 163, June 2021

[Teodorescu15] Lucian Radu Teodorescu, *Improving Flexibility and Efficiency in Programming Languages: a natural approach*, PhD Thesis, 2015, https://github.com/Sparrow-lang/sparrow-materials/raw/master/PhD/ThesisLucTeo.pdf

[Wikipedia] Wikipedia, 'Linguistic relativity', https://en.wikipedia.org/wiki/Linguistic_relativity

C++20 Text Formatting: An Introduction

C++20 has brought in many changes. Spencer Collyer gives an introduction to the new formatting library.

Much of the talk about the C++20 standard has focused on the 'big four' items, i.e. modules, concepts, coroutines, and ranges. This tends to obscure other improvements and additions that may have a bigger impact on the general programmer who spends their lives working on application code.

One such addition is the new text formatting library, **`std::format`**. This brings a more modern approach to text formatting, akin to Python's **`str.format`**. This article is intended as a brief introduction to the library, outlining the main items that allow you to produce formatted text.

The original proposal for the library [P0645] was written by Victor Zverovich and was based on his **`{fmt}`** library [fmtlib]. It was subsequently extended and modified by further proposals. A brief history can be found in a blog post [Zverovich19].

This article deals with the formatting of fundamental types and strings. A later article will describe how you can write formatters for your own types.

Current implementation status

At the time of writing (September 2021), support for **`std::format`** in major compilers is patchy.

The C++ library support page for GCC [GCClib] indicates that support is not yet available. A query to the libstdc++ mailing list received the response that no work on implementing it was currently known.

For Clang, work is being carried out on an implementation, and the progress can be found at [ClangFormat]. It is expected that full support will be available in Clang 14, due for release in 2022.

For MSVC, the C++ library support page [MSVClib] indicates that support is available for **`std::format`**, but with a caveat that to use it you currently need to pass the **`/std:c++latest`** flag, because of ongoing work on the standard.

Given the above, the code samples in this article were compiled using the **`{fmt}`** library, version 8.0.1. This version provides **`std::format`** compatible output. Versions of **`{fmt}`** before 8.0.0 had some differences, especially regarding some floating-point formatting and locale handling.

To convert the listings to use the standard library when available, replace the **`#include <fmt/format.h>`** with **`#include <format>`**, and remove the **`using namespace fmt`** line. One small wrinkle is that **`{fmt}`** has its own **`string_view`** class, so on the rare occasions when we use **`string_view`** in the examples, it is always qualified with the **`std`** namespace.

Text formatting functions

This section describes the main **`std::format`** functions. These are all you need if you just want to produce formatted text.

format

The first function provided by **`std::format`** is called **`format`**. Listing 1 gives an example of how you would use it, along with lines that produce the same output using **`printf`** and iostreams. Output of this program is given in Figure 1.

```
#include <fmt/format.h>
#include <iostream>
#include <cstdio>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  int i = 10;
  double f = 1.234;
  string s = "Hello World!";
  printf("Using printf   : %d %g %s\n", i, f,
    s.c_str());
  cout << "Using iostreams: " << i << " " << f
    << " " << s << "\n";
  cout << format("Using format   : {} {} {}\n", i,
    f, s);
}
```

Listing 1

```
Using printf   : 10 1.234 Hello World!
Using iostreams: 10 1.234 Hello World!
Using format   : 10 1.234 Hello World!
```

Figure 1

As can be seen from the listing, the interface to the **`format`** function is similar to the one for **`printf`**. It takes a *format string* specifying the format of the data to be output, followed by a list of values to use to replace fields defined in the string. In **`printf`** the fields to be replaced are indicated by preceding the format instructions with **`%`**, while in **`format`** they are delimited by **`{`** and **`}`** characters.

Looking at the strings passed to **`format`** in the listing, it is obvious that there is nothing in the replacement fields that indicates the types of values to be output. Unlike **`printf`**, **`format`** knows what types of arguments have been passed. This is because it is defined as a template function with the following signature:

```
template<class... Args>
string format(string_view fmt,
  const Args&... args);
```

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

Because format knows what the types of each argument are, if you try to use incompatible formatting with a value it will throw an exception

```
#include <fmt/format.h>
#include <iostream>
using namespace std;
using namespace fmt;
int main()
{
  int i = 10;
  try
  {
    cout << format("Using format: {:s}\n", i);
  }
  catch (const format_error& fe)
  {
    cout << "Caught format_error: " << fe.what()
      << "\n";
  }
}
```

Listing 2

The **fmt** argument is the *format string* specifying what the output should look like. The **args** arguments are the values we want to output. Note that **format** returns a string, so to output it you need to write the string somewhere – in the example, we simply send it to **cout**.

The *format string* syntax will be described in more detail later, but for now, it is sufficient to know that the **{}** items output the corresponding value from **args** using its default formatting.

Because **format** knows what the types of each argument are, if you try to use incompatible formatting with a value it will throw an exception. Listing 2 demonstrates this, where we give an integer argument, but the format type is a string one. This function produces the following output:

```
Caught format_error: invalid type specifier
```

This contrasts with **printf**, which in all likelihood will at best output garbage with no indication why, and at worst can crash your program.

format_to and format_to_n

The **format** function always returns a new string on each call. This is a problem if you want your output to be built up in several stages, as you would have to store each string produced and then stitch them all together at the end when outputting them.

To avoid this, you can use the **format_to** function. This appends the formatted text to the given output. The signature for this function is as follows:

```
template<class Out, class... Args>
Out format_to(Out out, string_view fmt,
  const Args&... args);
```

The first parameter, **out**, is an output iterator, which has to model **OutputIterator<const char&>**. The formatted output is sent to this output iterator. The function returns the iterator past the end of the written text.

```
#include <fmt/format.h>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
using namespace fmt;
string VecOut(const vector<int>& v)
{
  string retval;
  back_insert_iterator<string> out(retval);
  for (const auto& i: v)
  {
    out = format_to(out, "{} ", i);
  }
  return retval;
}
int main()
{
  vector<int> v1{2, 3, 5};
  cout << VecOut(v1) << "\n";
  vector<int> v2{1, 2, 4, 8, 16, 32};
  cout << VecOut(v2) << "\n";
  vector<int> v3{1, 4, 9, 16, 25, 36, 49, 64, 81,
    100};
  cout << VecOut(v3) << "\n";
}
```

Listing 3

```
2 3 5
1 2 4 8 16 32
1 4 9 16 25 36 49 64 81 100
```

Figure 2

Listing 3 shows how you might use **format_to** to output all the values in a vector. The output is a **back_insert_iterator<string>**, which matches the constraint, and appends the formatted values to the end of the string. Output from this program is in Figure 2.

If you also need to limit the number of characters written, use the **format_to_n** function. The signature for this function is similar to that for **format_to**, as follows:

```
template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out,
  iter_difference_t<Out> n,
  string_view fmt, const Args&...  args);
```

This takes the maximum number of characters to write in parameter **n**. The return value of this function is a **format_to_n_result** structure, which contains the following members:

- **out** – Holds the output iterator past the text written.

If you need to know how many characters would be output for a particular format string and set of arguments, you can call formatted_size

```
#include <fmt/format.h>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
using namespace fmt;
string VecOut(const vector<int>& v)
{
  string retval;
  back_insert_iterator<string> out(retval);
  for (const auto& i: v)
  {
    auto res = format_to_n(out, 5, "{}", i);
    retval += ' ';
  }
  return retval;
}
int main()
{
  vector<int> v1{1, 100, 10000};
  cout << VecOut(v1) << "\n";
  vector<int> v2{1, 1000, 1000000};
  cout << VecOut(v2) << "\n";
}
```

Listing 4

```
1 100 10000
1 1000 10000
```

Figure 3

- **size** – Holds the size that the formatted string would have had, before any potential truncation to a length of **n**. This can be used to detect if the output has been truncated, by checking if the value is greater than the **n** passed in.

The **VecOut** function in Listing 4 is similar to the one in Listing 3, but this time it limits the number of characters written for each value to 5. As can be seen from the output in Figure 3, the third value in **v2** is truncated from 1000000 to 10000 – probably something you'd only want to do if you were putting together a toy program to illustrate how the **format_to_n** function works.

formatted_size

If you need to know how many characters would be output for a particular format string and set of arguments, you can call **formatted_size**. This can be used if you want to create a buffer of the right size to accept the output. The function has the following signature:

```
template<class... Args>
size_t formatted_size(string_view fmt,
  const Args&... args);
```

```
#include <fmt/format.h>
#include <iostream>
#include <cstdio>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  int i = 10;
  double f = 1.234;
  string s = "Hello World!";
  string fmt_str{"{} {} {}\n"};
  cout << "Length of formatted data: "
    << formatted_size(fmt_str, i, f, s) << "\n";
  cout << format(fmt_str, i, f, s);
  cout << "123456789|123456789|12\n";
}
```

Listing 5

```
Length of formatted data: 22
10 1.234 Hello World!
123456789|123456789|12
```

Figure 4

The **size_t** value returned gives the length that the formatted string would have with the given arguments. If you are using this to create a buffer to write a C-style string to, remember that the value returned would not include any terminating **'\0'** character unless you include it in the *format string*.

Listing 5 illustrates the use of **formatted_size**. The output is in Figure 4. It may appear that the length output is incorrect but remember that the terminating newline character is included in the *format string*.

Wide-character support

The functions described above all deal with classes (**string**, **string_view**, the output iterators) that use **char** to represent the characters being handled. If you need to use **wchar_t** characters, there is an overload for each of the functions which take or return the appropriate class using **wchar_t**. For instance, the **format** function that uses **wchar_t** has the following signature:

```
template<class... Args>
wstring format(wstring_view fmt,
  const Args&... args);
```

Note that as of the C++20 standard, **std::format** does not handle any of the **charN_t** types (e.g. **char16_t**, **char32_t**).

Error reporting

Any errors detected by **std::format** are reported by throwing objects of the class **format_error**. This is derived from the

Any text that isn't part of a replacement field or an escape sequence is output literally

`std::runtime_error` class so it has a `what` function that returns the error string passed when the exception is created. Listing 2, presented previously, shows an example of catching a `format_error`.

Format string

The *format strings* used by the `std::format` functions consist of *escape sequences*, *replacement fields*, and other text. They are based on the style used by the Python `str.format`, for anyone familiar with that. A similar style is also used in the .NET family of languages, and in Rust.

Escape sequences

The two *escape sequences* recognised are `{{` and `}}`, which are replaced by `{` and `}` respectively. You would use them if you need a literal `{` or `}` in the output.

Obviously this is distinct from the normal string escapes that the compiler requires if you want to insert special characters in the string, such as `\n`. By the time the `std::format` functions see the string, these will have already been replaced by the compiler.

Replacement fields

A *replacement field* controls how the values passed to the `std::format` function are formatted. A replacement field has the following general format:

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

where:

- *arg-id*

  If given, this specifies the index of the argument in the value list that is output by the *replacement field*. Argument indexes start at 0 for the first argument after the *format string*.

- *format-spec*

  Gives the format specification to be applied to the value being handled. Note that if you give a *format-spec*, you have to precede it with a `:`, even if you do not give an *arg-id*.

Later sections will give more details on *arg-id*s and *format-spec*s. Examples of valid *replacement fields* are `{}`, `{0}`, `{:10d}`, `{1:s}`.

Other text

Any text that isn't part of a replacement field or an escape sequence is output literally as it appears in the *format string*.

Argument IDs

The first item in a *replacement field* is an optional *arg-id*. This specifies the index of the value in the argument list that you want to use for that *replacement field*. Argument index values start at 0.

If not specified, the arguments are simply used in the order that they appear in the function call. This is known as automatic argument numbering. For instance, in Listing 1 the `format` call has no *arg-ids*, so the arguments are just used in the order `i`, `f`, `s`.

```
#include <fmt/format.h>
#include <iostream>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  int i = 10;
  printf("%d %o %x\n", i, i, i);
  cout << i << " " << std::oct << i << " "
    << std::hex << i << std::dec << "\n";
  cout << format("{0} {0:o} {0:x}\n", i);
}
```

Listing 6

```
10 12 a
10 12 a
10 12 a
```

Figure 5

A given *format string* cannot have a mix of manual and automatic argument numbering. If you use an *arg-id* for one *replacement field* you have to use *arg-ids* for all *replacement fields* in the *format string*.

A simple use for this argument numbering can be seen in Listing 6, where it is used to output the same value in three different bases, along with lines that do the same thing for both `printf` and iostreams. The output from this is in Figure 5.

Another important use for this facility will be described later in the section 'Internationalization'.

Note that the *format string* does not have to specify *arg-ids* for all the arguments passed to the function. Any that are not given will simply be ignored. An example of this is shown in Listing 7, with the output in Figure 6.

Format specifications

The standard *format-spec* has the following general format[1]:

[[*fill*]*align*][*sign*][`#`][`0`][*width*][*prec*][`L`][*type*]

There should be no spaces between each item in the *format-spec*. Also, every item is optional, except that if *fill* is specified, it must be immediately followed by *align*. If *align* is given, any '`0`' will be ignored.

Anyone familiar with `printf` format strings will see that `std::format` uses a very similar style. However, there are some

1. This section describes the standard *format-spec* defined by `std::format` for formatting fundamental types, `string`s, and `string_view`s. Other types, like `std::chrono`, have their own *format-spec* definitions, and user-defined types can also define their own.

like printf format specifiers, but unlike many iostreams manipulators,the values given in a format-spec only apply to the current field and don't affect any later fields

significant differences, so the following sections describe each item in the above format in detail, except for the '`L`' character, which will be left until the section on internationalization.

Note that like `printf` format specifiers, but unlike many iostreams manipulators, the values given in a *format-spec* only apply to the current field and don't affect any later fields.

```
#include <fmt/format.h>
#include <iostream>
#include <string>
using namespace std;
using namespace fmt;
void write_success(int warnings, int errors)
{
  string fmtspec = "Compilation ";
  if (errors == 0)
  {
    fmtspec += "succeeded";
    if (warnings != 0)
    {
      fmtspec += " with {0} Warning(s)";
    }
  }
  else
  {
    fmtspec += "failed with {1} Error(s)";
    if (warnings != 0)
    {
      fmtspec += " and {0} Warning(s)";
    }
  }
  fmtspec += "\n";
  cout << format(fmtspec, warnings, errors);
}
int main()
{
  write_success(0, 0);
  write_success(10, 0);
  write_success(0, 10);
  write_success(10, 10);
}
```

Listing 7

```
Compilation succeeded
Compilation succeeded with 10 Warning(s)
Compilation failed with 10 Error(s)
Compilation failed with 10 Error(s) and 10
Warning(s)
```

Figure 6

The *type* option is called the presentation type. The valid values for each fundamental type are given below, along with a description of what effect they have. Remember that, unlike `printf`, `std::format` knows the type of value being output, so if you just want the default format for that value, you can omit the *type* option.

Text alignment and fill

The *align* value is a single character that gives the alignment to use for the current field. It can have any of the values `<`, `>`, or `^`. The meaning of these is as follows:

- `<` – The value is left-justified in the field width. This is the default for string fields.
- `>` – The value is right-justified in the field width. This is the default for numeric fields.
- `^` – The value is centred in the field width. Any padding will be distributed evenly on the left and right sides of the value. If an odd number of padding characters is needed, the extra one will always be on the right.

If the first character in the *format-spec* is immediately followed by one of the alignment characters, that first character is treated as the fill character to use if the field needs padding. A *fill* character **must** be followed by a valid *align* character. You cannot use either of the characters `{` or `}` as fill characters.

Note: The *fill* and *align* values only make sense if you also specify a *width* value, although it is not an error to specify them without one.

Listing 8 shows the effect of the *align* and *fill* values. The output is in Figure 7.

Sign, #, and 0

The *sign* value specifies how the sign for an arithmetic type is to be output. It can take the following values:

- `+` – A sign should always be output for both negative and non-negative values.
- `-` – A sign should only be output for negative values. This is the default.
- (space) – A sign should be output for negative values, and a space for non-negative values.

The `#` character indicates that the alternative form should be used for output of the given value. The meaning of this is described under the appropriate section below.

The `0` character is only valid when also specifying a *width* value. If present it pads the field with `0` characters after any sign character and/or base indicator. If an *align* value is present, any `0` character is ignored.

Note that the *sign*, `#`, and `0` values are only valid for arithmetic types,and for `bool` or `char` (`wchar_t` in wide string functions) when an integer presentation type is specified for them (see later).

```
#include <fmt/format.h>
#include <iostream>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  string str;
  cout << "No fill character specified:\n";
  str = format("|{:^10}| |{:<10}| |{:^10}| | "
    "{:>10}|\n", "default", "left", "centre",
    "right");
  cout << str;
  string fmtstr = "|{0:10}| |{0:<10}| | "
    "{0:^10}| |{0:>10}|\n";
  str = format(fmtstr, 123);
  cout << str;
  str = format(fmtstr, 1.23);
  cout << str;
  str = format(fmtstr, "abcde");
  cout << str;

  cout << "\nFill character set to '*'\n";
  str = format("|{:*<10}| |{:*^10}| |{:*>10}|\n",
    "left", "centre", "right");
  cout << str;
  fmtstr = "|{0:*<10}| |{0:*^10}| |{0:*>10}|\n";
  str = format(fmtstr, 123);
  cout << str;
  str = format(fmtstr, 1.23);
  cout << str;
  str = format(fmtstr, "abcde");
  cout << str;
}
```

Listing 8

```
No fill character specified:
| default  | |left      | |  centre  | |     right|
|       123| |123       | |   123    | |       123|
|      1.23| |1.23      | |   1.23   | |      1.23|
|abcde     | |abcde     | |  abcde   | |     abcde|

Fill character set to '*'
|left******| |**centre**| |******right|
|123*******| |***123****| |*******123|
|1.23******| |***1.23***| |******1.23|
|abcde*****| |**abcde***| |*****abcde|
```

Figure 7

Listing 9 shows the effect of the *sign* and `0` values. Output is shown in Figure 8. The effect of the `#` value will be shown in examples in the arithmetic type sections.

Width and precision

The *width* value can be used to give the minimum width for a field. If the output value needs more characters than the specified width, it will be displayed in full, not truncated to the width. If you need the value to be truncated to a certain width you can use the `format_to_n` function to output the value, with the guarantee that only the given number of characters at most will be written.

The value given for the *width* field depends on whether you are hard-coding the width in the string, or need it to be specified dynamically at runtime. If it is to be hard-coded, it should be given as a literal positive decimal number. If you need to specify the width dynamically at runtime, you use a nested replacement field, which looks like `{}` or {*n*}.

Listing 10 demonstrates the use of the *width* value, using both literal values and nested replacement fields with automatic and manual numbering. As

```
#include <fmt/format.h>
#include <iostream>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  int ineg = -10;
  int ipos = 5;
  double fneg = -1.2;
  double fpos = 2.34;

  cout << format(
    "With sign '+' :|{:+}|{:+}|{:+}|{:+}|\n",
    ineg, fneg, ipos, fpos);
  cout << format(
    "With sign '-' :|{:-}|{:-}|{:-}|{:-}|\n",
    ineg, fneg, ipos, fpos);
  cout << format(
    "With sign ' ' :|{: }|{: }|{: }|{: }|\n",
    ineg, fneg, ipos, fpos);
  cout << format("With sign '+' and '0' "
    ":|{:+06}|{:+06}|{:+06}|{:+06}|\n",
    ineg, fneg, ipos, fpos);
  cout << format("With sign '-' and '0' "
    ":|{:-06}|{:-06}|{:-06}|{:-06}|\n",
    ineg, fneg, ipos, fpos);
  cout << format("With sign ' ' and '0' "
    ":|{: 06}|{: 06}|{: 06}|{: 06}|\n",
    ineg, fneg, ipos, fpos);
}
```

Listing 9

shown in Figure 9, if the value is wider than the given *width*, the *width* value is ignored and the field is wide enough to display the full value.

The *prec* value is formed of a decimal point followed by the precision, which like the *width* field can be a literal positive decimal number or a nested replacement field.

The *prec* value is only valid for floating-point or string fields. It has different meanings for the two types and will be described in the relevant section below.

If using a nested replacement field for either *width* or *prec*, you must use the same numbering type as for the *arg-ids*, e.g. if using manual numbering for *arg-ids* you must also use it for nested replacement fields.

If you use automatic numbering, the *arg-ids* are assigned based on the count of `{` characters up to that point, so the *width* and/or *prec* values come after the value they apply to. This contrasts with `printf`, where if using the `*` to indicate the value is read from the argument list, the values for *width* and *prec* appear before the value they apply to.

Integer presentation types

The available integer presentation types are given below. Where relevant, the effect of selecting the alternate form using the `#` flag is also listed. Note that any sign character will always precede the prefix added in alternate form.

- `d` – Decimal format. This is the default if no presentation type is given.

```
With sign '+' :|-10|-1.2|+5|+2.34|
With sign '-' :|-10|-1.2|5|2.34|
With sign ' ' :|-10|-1.2| 5| 2.34|
With sign '+' and '0' :|-00010|-001.2|+00005|+02.34|
With sign '-' and '0' :|-00010|-001.2|000005|002.34|
With sign ' ' and '0' :|-00010|-001.2| 00005| 02.34|
```

Figure 8

```
#include <fmt/format.h>
#include <iostream>
using namespace std;
using namespace fmt;
int main()
{
  int v1 = 10;
  int v2 = 10'000'000;
  cout << format("Specified width: |{0:4}| "
    "|{0:12}| |{1:4}| |{1:12}|\n", v1, v2);
  cout << format("Variable width, automatic "
    "numbering: |{:{}}| |{:{}}|\n", v1, 5, v2, "
    "12);
  for (int len = 7; len < 11; ++len)
  {
    cout << format("Variable width={0:>2},
      manual numbering: |{1:{0}}| |{2:{0}}|\n",
      len, v1, v2);
  }
}
```

Listing 10

```
Specified width: |  10| |          10| |10000000| |    10000000|
Variable width, automatic numbering: |   10| |    10000000|
Variable width= 7, manual numbering: |     10| |10000000|
Variable width= 8, manual numbering: |      10| |10000000|
Variable width= 9, manual numbering: |       10| | 10000000|
Variable width=10, manual numbering: |        10| |  10000000|
```

Figure 9

- **b**, **B** – Binary format. For alternate form, the value is prefixed with **0b** for **b**, and **0B** for **B**.
- **o** – Octal format. For alternate form, the value is prefixed with **0** as long as it is non-zero. For example, 7 outputs as **07**, but 0 outputs as **0**.
- **x**, **X** – Hecadecimal format. The case of digits above 9 matches the case of the presentation type. For alternate form, the value is prefixed with **0x** for **x**, or **0X** for **X**.

```
#include <fmt/format.h>
#include <iostream>
using namespace std;
using namespace fmt;
int main()
{
  int i1 = -10;
  int i2 = 10;
  cout << format("Default: {} {} {}\n", i1, 0,
    i2);
  cout << format("Decimal type: {:d} {:d} {:d}\n",
    i1, 0, i2);
  cout << format("Binary type: {0:b} {0:B} {1:b} "
    "{1:B} {2:b} {2:B}\n", i1, 0, i2);
  cout << format("Binary '#' : {0:#b} {0:#B} "
    "{1:#b} {1:#B} {2:#b} {2:#B}\n", i1, 0, i2);
  cout << format("Octal type: {0:o} {1:o} "
    "{2:o}\n", i1, 0, i2);
  cout << format("Octal '#' : {0:#o} {1:#o} "
    "{2:#o}\n", i1, 0, i2);
  cout << format("Hex type: {0:x} {0:X} {1:x} "
    "{1:X} {2:x} {2:X}\n", i1, 0, i2);
  cout << format("Hex '#' : {0:#x} {0:#X} {1:#x} "
    "{1:#X} {2:#x} {2:#X}\n", i1, 0, i2);
  cout << format("Char type: |{:c}| |{:c}|\n", 32,
    126);
}
```

Listing 11

- **c** – Outputs the character with the code value given by the integer. A **format_error** will be thrown if the value is not a valid code value for the character type of the *format string*.

Listing 11 gives examples of outputting using all the presentation types, with and without alternate form where that is relevant. The output is shown in Figure 10.

Floating-point presentation types

The available floating-point presentation types are given below.

- **e** – Outputs the value in scientific notation. If no *prec* value is given, it defaults to 6.
- **f** – Outputs the value in fixed-point notation. If no *prec* value is given, it defaults to 6.
- **g** – Outputs the value in general notation, which picks between **e** and **g** form. The rules are slightly arcane but are the same as used for **g** when used with **printf**. If no *prec* value is given, it defaults to 6.
- **a** – Outputs the value using scientific notation, but with the number represented in hexadecimal. Because **e** is a valid hex digit, the exponent is indicated with a **p** character.

If no presentation type is given, the output depends on whether a *prec* value is given or not. If *prec* is present the output is the same as using **g**. If *prec* is not present, the output is in either fixed-point or scientific notation, depending on which gives the shortest output that still guarantees that reading the value in again will give the same value as was written out.

If the floating-point value represents infinity or NaN, the values 'inf' and 'nan' will be output respectively. They will be preceded by the appropriate sign character if required. Specifying **#** does not cause a base prefix to be output for infinity or NaN.

The **e**, **f**, **g**, and **a** presentation types have equivalent **E**, **F**, **G**, and **A** types which perform the same, but output any alphabetic characters in uppercase rather than lowercase. For the **f** and **F** types this only affects output of infinity and NaN.

If the **#** character is used to select the alternate form, it causes a decimal point character to always be output, even if there are no digits after it. This does not apply to infinity and NaN values.

Listing 12 gives examples of outputting using the lowercase presentation types, and how *prec* and alternate form affects the output. Output is given in Figure 11.

Listing 13 gives examples of outputting infinity and NaN values. Because all presentation types give the same output for infinity and NaN, we only give the output for types **f** and **F**. Output is given in Figure 12.

Character presentation types

The default presentation type for **char** and **wchar_t** is **c**. It simply copies the character to the output.

You can also use the integer presentation types **b**, **B**, **d**, **o**, **x**, and **X**. They write the integer value of the character code to the output, and take account of the alternate form flag **#** if relevant.

```
Default: -10 0 10
Decimal type: -10 0 10
Binary type: -1010 -1010 0 0 1010 1010
Binary '#' : -0b1010 -0B1010 0b0 0B0 0b1010 0B1010
Octal type: -12 0 12
Octal '#' : -012 0 012
Hex type: -a -A 0 0 a A
Hex '#' : -0xa -0XA 0x0 0X0 0xa 0XA
Char type: | | |~|
```

Figure 10

```
#include <fmt/format.h>
#include <iostream>

using namespace std;
using namespace fmt;

int main()

{
  double small = 123.4567;
  double nodps = 34567.;
  double large = 1e10+12.345;
  double huge = 1e20;

  cout << "Default precision:\n";
  cout << format("Default: {} {} {} {} {}\n", 0.0,
    small, nodps, large, huge);
  cout << format("Type f : {:f} {:f} {:f} {:f} "
    "{:f}\n", 0.0, small, nodps, large, huge);
  cout << format("Type e : {:e} {:e} {:e} {:e} "
    "{:e}\n", 0.0, small, nodps, large, huge);
  cout << format("Type g : {:g} {:g} {:g} {:g} "
    "{:g}\n", 0.0, small, nodps, large, huge);
  cout << format("Type a : {:a} {:a} {:a} {:a} "
    "{:a}\n", 0.0, small, nodps, large, huge);
  cout << "\nAlternate form:\n";
  cout << format("Default: {:#} {:#} {:#} {:#} "
    "{:#}\n", 0.0, small, nodps, large, huge);
  cout << format("Type f : {:#f} {:#f} {:#f} "
     "{:#f}{:#f}\n", 0.0, small, nodps, large,
     huge);
  cout << format("Type e : {:#e} {:#e} {:#e} "
    "{:#e} {:#e}\n", 0.0, small, nodps, large,
    huge);
  cout << format("Type g : {:#g} {:#g} {:#g} "
    "{:#g} {:#g}\n", 0.0, small, nodps, large,
     huge);
```

Listing 12

```
  cout << format("Type a : {:#a} {:#a} {:#a} "
    {:#a} {:#a}\n", 0.0, small, nodps, large,
    huge);
  cout << "\nPrecision=3:\n";
  cout << format("Default: {:.3} {:.3} {:.3} "
    "{:.3} {:.3}\n", 0.0, small, nodps, large,
    huge);
  cout << format("Type f : {:.3f} {:.3f} {:.3f} "
    "{:.3f} {:.3f}\n", 0.0, small, nodps, large,
    huge);
  cout << format("Type e : {:.3e} {:.3e} {:.3e} "
    "{:.3e} {:.3e}\n", 0.0, small, nodps, large,
    huge);
  cout << format("Type g : {:.3g} {:.3g} {:.3g} "
    "{:.3g} {:.3g}\n", 0.0, small, nodps, large,
    huge);
  cout << format("Type a : {:.3a} {:.3a} {:.3a} "
    "{:.3a} {:.3a}\n", 0.0, small, nodps, large,
    huge);
  cout << "\nPrecision=3, alternate form:\n";
  cout << format("Default: {:#.3} {:#.3} {:#.3} "
    "{:#.3} {:#.3}\n", 0.0, small, nodps, large,
    huge);
  cout << format("Type f : {:#.3f} {:#.3f} "
    "{:#.3f} {:#.3f} {:#.3f}\n", 0.0, small,
    nodps, large, huge);
  cout << format("Type e : {:#.3e} {:#.3e} "
    "{:#.3e} {:#.3e} {:#.3e}\n", 0.0, small,
    nodps, large, huge);
  cout << format("Type g : {:#.3g} {:#.3g} "
    "{:#.3g} {:#.3g} {:#.3g}\n", 0.0, small,
    nodps, large, huge);
  cout << format("Type a : {:#.3a} {:#.3a} "
    "{:#.3a} {:#.3a} {:#.3a}\n", 0.0, small,
    nodps, large, huge);
}
```

Listing 12 (cont'd)

```
Default precision:
Default: 0 123.4567 34567 10000000012.345 1e+20
Type f : 0.000000 123.456700 34567.000000 10000000012.344999 10000000000000000000.000000
Type e : 0.000000e+00 1.234567e+02 3.456700e+04 1.000000e+10 1.000000e+20
Type g : 0 123.457 34567 1e+10 1e+20
Type a : 0x0p+0 0x1.edd3a92a30553p+6 0x1.0e0ep+15 0x1.2a05f2062c28fp+33 0x1.5af1d78b58c4p+66

Alternate form:
Default: 0.0 123.4567 34567.0 10000000012.345 1.e+20
Type f : 0.000000 123.456700 34567.000000 10000000012.344999 10000000000000000000.000000
Type e : 0.000000e+00 1.234567e+02 3.456700e+04 1.000000e+10 1.000000e+20
Type g : 0.00000 123.457 34567.0 1.00000e+10 1.00000e+20
Type a : 0x0.p+0 0x1.edd3a92a30553p+6 0x1.0e0ep+15 0x1.2a05f2062c28fp+33 0x1.5af1d78b58c4p+66

Precision=3:
Default: 0 123 3.46e+04 1e+10 1e+20
Type f : 0.000 123.457 34567.000 10000000012.345 10000000000000000000.000
Type e : 0.000e+00 1.235e+02 3.457e+04 1.000e+10 1.000e+20
Type g : 0 123 3.46e+04 1e+10 1e+20
Type a : 0x0.000p+0 0x1.eddp+6 0x1.0e1p+15 0x1.2a0p+33 0x1.5afp+66

Precision=3, alternate form:
Default: 0.00 123.0 3.46e+04 1.00e+10 1.00e+20
Type f : 0.000 123.457 34567.000 10000000012.345 10000000000000000000.000
Type e : 0.000e+00 1.235e+02 3.457e+04 1.000e+10 1.000e+20
Type g : 0.00 123.0 3.46e+04 1.00e+10 1.00e+20
Type a : 0x0.000p+0 0x1.eddp+6 0x1.0e1p+15 0x1.2a0p+33 0x1.5afp+66
```

Figure 11

```
#include <fmt/format.h>
#include <limits>
#include <iostream>
using namespace std;
using namespace fmt;
int main()
{
  auto pinf =
    std::numeric_limits<double>::infinity();
  auto ninf =
    -std::numeric_limits<double>::infinity();
  auto pnan =
    std::numeric_limits<double>::quiet_NaN();
  auto nnan =
    -std::numeric_limits<double>::quiet_NaN();
  cout << "Default:\n";
  cout << format("Default: {} {} {} {}\n", ninf,
    pinf, nnan, pnan);
  cout << format("Type f : {:f} {:f} {:f} {:f}\n",
    ninf, pinf, nnan, pnan);
  cout << format("Type F : {:F} {:F} {:F} {:F}\n",
    ninf, pinf, nnan, pnan);
  cout << "\nAlternate form:\n";
  cout << format("Default: {:#} {:#} {:#} {:#}\n",
    ninf, pinf, nnan, pnan);
  cout << format("Type f : {:#f} {:#f} {:#f} "
    "{:#f}\n", ninf, pinf, nnan, pnan);
  cout << format("Type F : {:#F} {:#F} {:#F} "
    "{:#F}\n", ninf, pinf, nnan, pnan);
  cout << "\nWidth=7:\n";
  cout << format("Default: |{:7}| |{:7}| |{:7}| "
    "|{:7}|\n", ninf, pinf, nnan, pnan);
  cout << "\nWidth=7, using '0':\n";
  cout << format("Default: |{:07}| |{:07}| "
    "|{:07}| |{:07}|\n", ninf, pinf, nnan, pnan);
}
```

Listing 13

```
Default:
Default: -inf inf -nan nan
Type f : -inf inf -nan nan
Type F : -INF INF -NAN NAN

Alternate form:
Default: -inf inf -nan nan
Type f : -inf inf -nan nan
Type F : -INF INF -NAN NAN

Width=7:
Default: |-inf   | |inf    | |-nan   | |nan    |

Width=7, using '0':
Default: |   -inf| |    inf| |   -nan| |    nan|
```

Figure 12

Listing 14 gives examples of outputting characters. Output is given in Figure 13.

String presentation types

String formatting works for **`std::string`** and **`std::string_view`** as well as the various **`char*`** types.

The only presentation type for strings is **`s`**, which is also the default if not given. The default alignment for string fields is left-justified.

If a precision value is specified with *prec*, and it is smaller than the string length, it causes only the first *prec* characters from the string to be output. This has the effect of reducing the effective length of the string when checking against any *width* parameter.

```
#include <fmt/format.h>
#include <iostream>
using namespace std;
using namespace fmt;
int main()
{
  char c = 'a';
  cout << format("Default: {}\n", c);
  cout << format("Char type: {:c}\n", c);
  cout << format("Decimal type: {:d}\n", c);
  cout << format("Binary type: {0:b} {0:B} "
    "{0:#b} {0:#B}\n", c);
  cout << format("Octal type: {0:o} {0:#o}\n", c);
  cout << format("Hex type: {0:x} {0:X} {0:#x} "
    "{0:#X}\n", c);
}
```

Listing 14

```
Default: a
Char type: a
Decimal type: 97
Binary type: 1100001 1100001 0b1100001 0B1100001
Octal type: 141 0141
Hex type: 61 61 0x61 0X61
```

Figure 13

Listing 15 shows examples of outputting various types of string, as well as the interaction between *width* and *prec* values. The output is shown in Figure 14.

Bool presentation types

The default **`bool`** presentation type is **`s`**, which outputs **true** or **false**.

You can also use the integer presentation types **`b`**, **`B`**, **`d`**, **`o`**, **`x`**, or **`X`**. These behave like the same types for integers, treating **`false`** as 0 and **`true`** as 1.

You can also use **`c`** as a **`bool`** presentation type. It will output the characters with values 0x1 and 0x0 for **`true`** and **`false`**, which may not be what you expect (or particularly useful).

Listing 16 is an example of formatting **`bool`**s, with the output in Figure 15.

The line that uses the **`c`** presentation type appears to print nothing, but if you send the output to a file and then examine the output using a program that displays the actual bytes in the file, you will see that the characters with codes 0x0 and 0x1 have been output. For instance, you can use **`od c`** on a Unix or Linux box.

Pointer presentation types

The only *type* value available for pointers is '**`p`**'. It can be omitted and **`std::format`** will deduce the type from the argument. Note that if the pointer is to one of the **`char`** types, it will be treated as a string, not as a pointer. If you want to output the actual pointer value you need to cast it to a **`void*`**.

Pointer values are output in hexadecimal, with the prefix '0x' added. and digits **`a`** to **`f`** in lowercase. The output is right-justified by default, just like arithmetic types.

Note that the C++20 standard specifies that only pointers for which **`std::is_void_t`** returns true can be output by **`std::format`**, which in practice means you need to cast any pointers to **`void\*`**. Listing 17 shows examples of pointer formatting, and does exactly that. Sample output is in Figure 16.

Internationalization

Internationalization, or i18n as it is commonly written, is the process of writing a program so its output can be used natively by people speaking different languages and with different conventions for writing things like numbers and dates.

```
#include <fmt/format.h>
#include <iostream>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  string s = "Hello World!";
  const char* cp = "Testing. Testing.";
  const char* cp2 = "Goodbye World!";
  std::string_view sv = cp2;
  cout << format("Default: {} {} {}\n", s, cp,
    sv);
  cout << format("Type   : {:s} {:s} {:s}\n", s,
    cp, sv);
  cout << "\nUsing width and precision:\n";
  cout << format("With width: w=7:|{0:7s}| "
    "w=20:|{0:20s}|\n",     s);
  cout << format("With precision: p=4:|{0:.4s}| "
    "p=15:|{0:.15s}|\n", s);
  cout << format("With width and precision: "
    "w=7,p=4:|{0:7.4s}| w=20,p=4:|{0:20.4s}|\n",
    s);
  cout << format("With width, precision, align: "
    "|{0:<8.4s}| |{0:^8.4s}| |{0:>8.4s}|\n", s);
}
```

Listing 15

```
Default: Hello World! Testing. Testing. Goodbye
World!
Type   : Hello World! Testing. Testing. Goodbye
World!

Using width and precision:
With width: w=7:|Hello World!| w=20:|Hello World!
|
With precision: p=4:|Hell| p=15:|Hello World!|
With width and precision: w=7,p=4:|Hell   |
w=20,p=4:|Hell                 |
With width, precision, align: |Hell    | |  Hell
| |    Hell|
```

Figure 14

By default, `std::format` takes no account of the current locale when outputting values. The reasons for this are described in the original proposal [P0645] in the section 'Locale support'. In contrast, iostreams takes account of the locale on all output, even if it is set to the default.

Format strings

The ability to use manual argument numbering in *format strings* to reorder arguments is useful when using translated output. Allowing arguments to appear in a different order in the output can make for grammatically correct output in a given language.

Rather than hard-coding the *format strings* in your code, you could use a mechanism that provides the correct translated string to use, with manual argument numbers inserted. Libraries exist that make it easier to use such translated strings – for instance, GNU's `gettext` library [gettext] takes a string and looks up the translated version of it, as long as the translations have been provided in the correct format.

Locale-aware formatting

In a *format-spec*, the `L` modifier can be used to specify that the field should be output in a locale-aware fashion. Without this modifier, the locale is ignored. You can use this for output of numeric values, and

```
#include <fmt/format.h>
#include <iostream>
using namespace std;
using namespace fmt;
int main()
{
  cout << "      b B d o x X\n";
  cout << format("{0}  {0:b} {0:B} {0:d} {0:o}
    {0:x} {0:X}\n", true);
  cout << format("{0} {0:b} {0:B} {0:d} {0:o} "
    "{0:x} {0:X}\n", false);
  cout << "\nUsing alternate form\n";
  cout << "#b  #B  #d #o #x  #X\n";
  cout << format("{0:#b} {0:#B} {0:#d}  {0:#o} "
    "{0:#x} {0:#X}\n", true);
  cout << format("{0:#b} {0:#B} {0:#d}  {0:#o} "
    "{0:#x} {0:#X}\n", false);
  cout << "\nUsing type=s\n";
  cout << format("|{:s}| |{:s}|\n", false, true);
  cout << "\nUsing type=c\n";
  cout << format("|{:c}| |{:c}|\n", false, true);
}
```

Listing 16

```
      b B d o x X
true  1 1 1 1 1 1
false 0 0 0 0 0 0

Using alternate form
#b  #B  #d #o #x  #X
0b1 0B1 1  01 0x1 0X1
0b0 0B0 0  0  0x0 0X0

Using type=s
|false| |true|

Using type=c
| | ||
```

Figure 15

```
#include <fmt/format.h>
#include <iostream>
#include <memory>
using namespace std;
using namespace fmt;
int main()
{
  int* pi = new int;
  void* vpi = static_cast<void*>(pi);
  double* pd = new double;
  void* vpd = static_cast<void*>(pd);
  void* pnull = nullptr;

  cout << format("Default: |{}| |{}| |{}|\n", vpi,
    vpd, pnull);
  cout << format("Type   : |{:p}| |{:p}| "
    "|{:p}|\n", vpi, vpd, pnull);
  cout << format("Width  : |{:20p}| |{:20p}| "
    "|{:20p}|\n", vpi, vpd, pnull);
}
```

Listing 17

```
Default: |0x55595cbc8eb0| |0x55595cbc8ed0| |0x0|
Type   : |0x55595cbc8eb0| |0x55595cbc8ed0| |0x0|
Width  : |      0x55595cbc8eb0| |      0x55595cbc8ed0| |                0x0|
```

Figure 16

```
#include <fmt/format.h>
#include <iostream>
#include <cstdio>
#include <string>
using namespace std;
using namespace fmt;
int main()
{
  double dval = 1.5;
  int ival = 1'000'000;
  cout << "Using default locale:\n";
  cout << format("format  : {:.2f} {:12d}\n",
    dval, ival);
  cout << format("format+L: {:.2Lf} {:12Ld}\n",
    dval, ival);
  cout << "\nUsing global locale de_DE:\n";
  locale::global(locale("de_DE"));
  cout << format("format  : {:.2f} {:12d}\n",
    dval, ival);
  cout << format("format+L: {:.2Lf} {:12Ld}\n",
    dval, ival);
  cout << "\nUsing function-specific locale:\n";
  cout << format(locale("en_US"),
    "en_US: {0:.2f} {0:.2Lf} {1:12d} {1:12Ld}\n",
    dval, ival);
  cout << format(locale("de_DE"),
    "de_DE: {0:.2f} {0:.2Lf} {1:12d} {1:12Ld}\n",
    dval, ival);
}
```

Listing 18

```
Using default locale:
format  : 1.50      1000000
format+L: 1.50      1000000

Using global locale de_DE:
format  : 1.50      1000000
format+L: 1,50    1.000.000

Using function-specific locale:
en_US: 1.50 1.50      1000000    1,000,000
de_DE: 1.50 1,50      1000000    1.000.000
```

Figure 17

also **bool** when doing string format output. When used with **bool** it changes the 'true' and 'false' values to the appropriate **numpunct::truename** and **numpunct::falsename** instead.

The various output functions described earlier use the global locale when doing locale-aware formatting. You can change the global locale with a function call like the following:

```
std::locale::global(std::locale("de_DE"));
```

This will set the global locale to the one for Germany, **de_DE**.

If you only want to change the locale for a single function call, there are overloads of the various output functions that take the locale as their first parameter. For instance, the **format** function has the following overload:

```
template<class... Args>
string format(const std::locale& loc,
  string_view fmt, const Args&... args);
```

Listing 18 shows examples of using locale-aware output, using both global locales and function-specific ones. Output from this program is shown in Figure 17.

Avoiding code bloat

The formatting functions are all template functions. This means that each time one of the functions is used with a new set of argument types, a new template instantiation will be generated. This could quickly lead to unacceptable code bloat if these functions did the actual work of generating the output values.

To avoid this problem, the **format** and **format_to** functions call helper functions to do the actual formatting work. These helper functions have names formed by adding '**v**' to the start of the name of the calling function. For instance, **format** calls **vformat**, which has the following signature:

```
string vformat(string_view fmt,
  format_args args);
```

The **format_args** argument is a container of type-erased values, the details of which are probably only interesting to library authors. They are certainly outside the scope of this article.

There will only be a single instantiation of the **vformat** function, and one **vformat_to** instantiation for each type of output iterator.

The actual work of doing the formatting is done by these helper functions, so the amount of code generated for each call to **format** or **format_to** is considerably reduced.

These functions are part of the **std::format** public API, so you can use them yourself if you want to. Listing 19, which is based on code presented in [P0645], shows one such use, with a logging function that takes any number of arguments. The output from this function is in Figure 18. The call to **vlog_error** uses the **make_format_args** function to generate the **format_args** structure.

```
#include <fmt/format.h>
#include <iostream>
#include <string>
using namespace std;
using namespace fmt;
void vlog_error(int code, std::string_view fmt,
format_args args)
{
  cout << "Error " << code << ": "
    << vformat(fmt, args) << "\n";
}
template<class... Args>
void log_error(int code, std::string_view fmt,
  const Args&... args)
{
  vlog_error(code, fmt,
    make_format_args(args...));
}
int main()
{
  int i = 10;
  double f = 1.234;
  string s = "Hello World!";
  log_error(1, "Bad input detected: {} is not "
    "an integer value", 10.1);
  log_error(10, "Oops - Type mismatch between {} "
    "and {}", "var1", 10);
  log_error(255, "Something went wrong!");
}
```

Listing 19

```
Error 1: Bad input detected: 10.1 is not an
integer value
Error 10: Oops - Type mismatch between var1 and 10
Error 255: Something went wrong!
```

Figure 18

Conclusion

Hopefully this article has given you a taster of what **std::format** can do. If you want to start trying it out you can use **{fmt}** as a good proxy for it until library authors catch up with the standard.

In my own projects I am already using the `std::format` compatible parts of `{fmt}`, and in general find it easier and clearer than the equivalent iostreams code.

In future articles I intend to explore how to create formatters for your own user defined types, and also how to convert from existing uses of iostreams and `printf`-family functions to `std::format`. ■

Acknowledgements

I'd like to extend my thanks to Victor Zverovich for responding quickly to my various queries whilst writing this article, and also for reviewing draft versions of the article, making many useful suggestions for improvement. I'd also like to thank the *Overload* reviewers for making useful suggestions for improvements to the article. Any errors and ambiguities that remain are solely my responsibility.

Appendix 1: Comparison of format and printf format specifications

Both `std::format` and `printf` use format specifications to define how a field is to be output. Although they are similar in many cases, there are enough differences to make it worth outlining them here. The following list has the `printf` items on the left and gives the `std::format` equivalent if there is one.

- `-` – Replaced by the `<`, `^`, and `>` flags to specify alignment. Repurposed as a sign specifier.
- `+`, space – These have the same meaning and have been joined as sign specifiers by `-`.
- `#` – Has the same meaning.
- `0` – Has the same meaning.
- `*` – Used in `printf` to say the width or precision is specified at run-time. Replaced by nested replacement fields. See Note 1 below on argument ordering differences when converting these.
- `d` – In `printf` it specifies a signed integer. In `std::format`, it specifies any integer, but as it is the default it can be omitted, except when outputting the integer value of a character or `bool`.
- `h`, `l`, `ll`, `z`, `j`, `t` – Not used. In `printf`, they specify the size of integer being output. `std::format` is type aware so these are not needed.
- `hh` – Not used. In `printf` it specifies the value is a char to be output as a numeric value. Use `d` in `std::format` to do the same thing.
- `i`, `u` – Not used. Replaced by `d`.
- `L` – Not used. In `printf` it specifies a `long double` value is being passed, but `std::format` is type aware. The `L` character has been repurposed to say the field should take account of the current locale when output.
- `n` – Not used. In `printf`, it saves the number of characters output so far to an integer pointed to by the argument. Use `formatted_size` as a replacement.
- `c`, `p`, `s` – Have the same meaning, but as they are the default output type for their argument type, they can be omitted.
- `a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`, `o`, `x`, `X` – All have the same meaning.

Note 1: If using dynamic values for width or precision, and you are using automatic parameter numbering in `std::format`, the order of assigning parameter numbers when parsing the string means the width and precision values come after the value being output, whereas in the `printf`-family functions they come before the value.

Note 2: POSIX adds positional parameters to the `printf` format specification. These are specified using `%n$` for value fields, or `*m$` for width and precision fields – e.g `%1$*2$,*3$f`. The `std::format` format specification already supports these using manual numbering mode.

Table 1 (overleaf) shows examples of `printf` formatting and the equivalent `std::format` version.

Appendix 2: std::format and {fmt}

As previously mentioned, `std::format` is based on the `{fmt}` library. However, `{fmt}` offers a number of extra facilities that are not in the C++20 version of `std::format`. This appendix gives a brief description of the main ones.

Direct output to a terminal

Both iostreams and the `printf`-family functions have the ability to write directly to the terminal, either using `std::cout` or the `printf` function itself. The only way to do this in `std::format` in C++20 is to use `format_to` with a `back_inserter` attached to `std::cout`, but this is not recommended as it leads to slow performance. This is why the examples in this article write the strings produced to `std::cout`.

The `{fmt}` library provides a `print` function to do this work. It can in fact write to any `std::FILE*` stream, defaulting to `stdout` if none is specified in the function call. A proposal to add this to C++23 has been made [P2093].

Named arguments

The `{fmt}` library supports named arguments, so you can specify an argument in the replacement field by name as well as position.

Output of value ranges

There are a number of utility functions provided in `{fmt}`. One of the most useful is `fmt::join`, which can be used to output ranges of values in a single operation, with a given separator string between each value. The ranges output can be tuples, initializer_lists, any container to which `std::begin` and `std::end` can be applied, or any range specified by begin and end iterators.

References

[ClangFormat] Libc++ Format Status, https://libcxx.llvm.org//Status/Format.html

[fmtlib] `{fmt}` library, https://github.com/fmtlib/fmt

[GCClib] GCC library support, https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.2020

[gettext] GNU gettext library, https://www.gnu.org/software/libc/manual/html_node/Translation-with-gettext.html

[MSVClib] MSVC C++ library support, https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-language-conformance

[P0645] *Text Formatting*, Victor Zverovich, 2019, http://wg21.link/P0645

[P2093] *Formatted Output*, Victor Zverovish, 2021, http://wg21.link/P2093

[Zverovich19] *std::format in C++20*, Victor Zverovich, https://www.zverovich.net/2019/07/23/std-format-cpp20.html

printf	std::format
Integer output	

`int iv = -1;`
`short sv = -2;`
`long lv = -3;`
`long long llv = -4;`
`printf("%i, %hi, %li, %lli\n", iv, sv, lv, llv);`
`unsigned int uiv = 1;`
`unsigned short usv = 2;`
`unsigned long ulv = 3;`
`unsigned long long ullv = 4;`
`printf("%u, %hu, %lu, %llu\n", uiv, usv, ulv, ullv);`
	`int iv = -1;`
	`short sv = -2;`
	`long lv = -3;`
	`long long llv = -4;`
	`cout << format("{}, {}, {}, {}\n", iv, sv, lv, llv);`
	`unsigned int uiv = 1;`
	`unsigned short usv = 2;`
	`unsigned long ulv = 3;`
	`unsigned long long ullv = 4;`
	`cout << format("{}, {}, {}, {}\n", uiv, usv, ulv, ullv);`

Floating-point output	

`float f = 1.234;`
`double d = 2.345;`
`long double ld = 3.456;`
`printf("%g %g %Lg\n", f, d, ld);`
	`float f = 1.234;`
	`double d = 2.345;`
	`long double ld = 3.456;`
	`cout << format("{} {} {}\n", f, d, ld);`

Character output	
`char c = 'A';` `printf("%c %hhd\n", c, c);`	`char c = 'A';` `cout << format("{} {:d}\n", c, c);`
String output	

`const char* cptr = "Mary had a little lamb";`
`std::string str(cptr);`
`std::string_view strview(str);`
`printf("%s\n", cptr);`
`printf("%s\n", str.c_str());`
`// printf cannot handle std::string_view`
	`const char* cptr = "Mary had a little lamb";`
	`std::string str(cptr);`
	`std::string_view strview(str);`
	`cout << format("{}\n", cptr);`
	`cout << format("{}\n", str);`
	`cout << format("{}\n", strview);`

Pointer output	
`printf("%p\n", static_cast<const void*>(cptr));`	`cout << format("{}\n", static_cast<const void*>(cptr));`
Field alignment	
`printf("[%-10d] [%10d]\n", iv, iv);` `// printf doesn't support centred alignment`	`cout << format("[{:<10}] [{:>10}]\n", iv, iv);` `cout << format("[{:^10}]\n", iv);`
Sign field	
`printf("[%+d] [% d] [%+d] [% d]\n", -1, -1, 1, 1);` `// printf does not support '-' as a sign specifier`	`cout << format("[{:+}] [{: }] [{:+}] [{: }]\n", -1, -1, 1, 1);` `cout << format("[{:-}] [{:-}]\n", -1, 1);`
Run-time width and precision	

`int width=10, prec=5;`
`double val=123.456;`
`printf("%*.*f\n", width, prec, val);`
	`int width=10, prec=5;`
	`double val=123.456;`
	`cout << format("{:{}.{}f}\n", val, width, prec);`

Positional parameters	
`float fv = 123.456;` `printf("%1$*2$.*3$f\n", fv, 10, 5);`	`float fv = 123.456;` `cout << format("{0:{1}.{2}f}\n", fv, 10, 5);`

Table 1

No Move vs Deleted Move Constructors

C++ allows you to mark constructors as deleted. Anders Knatten reveals what a deleted definition means in practice.

It's easy to think that deleting the move constructor means removing it. So if you do `MyClass(MyClass&&) = delete`, you make sure it doesn't get a `move` constructor. This is, however, not technically correct. It might seem like a nitpick, but it actually gives you a less useful mental model of what's going on.

First: When does this matter? It matters for understanding in which cases you're allowed to make a `copy`/`move` from an rvalue.

Listing 1 contains some examples of having to `copy`/`move` an object of type `MyClass`.

```
MyClass obj2(obj1);
MyClass obj3(std::move(obj1));

MyClass obj4 = obj1;
MyClass obj5 = std::move(obj1);

return obj1;
return std::move(obj1);
```

Listing 1

They are all examples of 'direct initialization' (the first two) and 'copy initialization' (the last four). Note that there is no concept of 'move initialization' in C++. Whether you end up using the `copy` or the `move` constructor to initialize the new object is just a detail.

For the rest of this article, let's just look at copy initialization; direct initialization works the same way for our purposes. In any case, you create a new copy of the object, and the implementation uses either the `copy` or the `move` constructor to do so.

Let's first look at a class `NoMove` (Listing 2).

This class has a user-declared copy constructor, so it doesn't automatically get a move constructor :

> If the definition of a class X does not explicitly declare a move constructor, a non-explicit one will be implicitly declared as defaulted if and only if
> - X does not have a user-declared copy constructor
> - (…)
>
> [C++Standard_1]

So this class doesn't have a move constructor at all. You didn't explicitly declare one, and none got implicitly declared for you.

On the other hand, let's see what happens if we explicitly delete the move constructor (Listing 3).

Anders Knatten Anders started programming in Turbo Pascal in 1995, and has been programming professionally in various languages since 2001. He's currently a senior developer at Zivid, making 3D cameras for robot vision. He's the author of the blog 'C++ on a Friday' (https://blog.knatten.org) and 'C++ Quiz' (https://cppquiz.org/).

```
struct NoMove
{
  NoMove();
  NoMove(const NoMove&);
};
```

Listing 2

```
struct DeletedMove
{
  DeletedMove();
  DeletedMove(const DeletedMove&);
  DeletedMove(DeletedMove&&) = delete;
};
```

Listing 3

This is called 'a deleted definition':

> A function definition of the form:
> `(…) = delete ;`
>
> is called a deleted definition. A function with a deleted definition is also called a deleted function. [C++Standard_2]

Importantly, that does not mean that its definition has been deleted/removed and is no longer there. It means that is has a definition, and that this particular kind of definition is called a 'deleted definition'. I like to read it as 'deleted-definition'.

So our `NoMove` class has no `move` constructor at all. Our `DeletedMove` class has a `move` constructor with a deleted definition.

Why does this matter?

Let's first look at a class with both a `copy` and a `move` constructor, and how to copy-initialize it (Listing 4).

When initializing `movable2`, we need to find a function to do that with. A `copy` constructor would do nicely. And since we do have a `copy` constructor, it indeed gets used for this.

What if we turn `movable` into an rvalue?

```
Movable movable2 = std::move(movable);
```

Now a `move` constructor would be great. And we do have one, and it indeed gets used.

```
struct Movable
{
  Movable();
  Movable(const Movable&);
  Movable(Movable&&);
};
Movable movable;
Movable movable2 = movable;
```

Listing 4

But what if we didn't have a move constructor? That's the case with our class **NoMove** in Listing 2.

This one has a **copy** constructor, so it doesn't get a **move** constructor. We can, of course, still make copies using the **copy** constructor:

```
NoMove noMove;
NoMove noMove2 = noMove;
```

But what happens now?

```
NoMove noMove;
NoMove noMove2 = std::move(noMove);
```

Are we now 'move initializing' **noMove2** and need the **move** constructor? Actually, we're not. We're still **copy**-initializing it, and need some function to do that task for us. A **move** constructor would be great, but a **copy** constructor would also do. It may be less efficient, but of course you're allowed to make a **copy** of an rvalue.

So this is fine, the code compiles, and the **copy** constructor is used to make a copy of the rvalue.

What happened behind the scenes in all the examples above, is overload resolution. Overload resolution looks at all the candidates to do the job, and picks the best one. In the cases where we initialize from an lvalue, the only candidate is the **copy** constructor. We're not allowed to move from an lvalue. In the cases where we initialize from an rvalue, both the **copy** and the **move** constructors are candidates. But the **move** constructor is a better match, as we don't have to convert the rvalue to an lvalue reference. For **Movable**, the **move** constructor got selected. For **NoMove**, there is no **move** constructor, so the only candidate is the **copy** constructor, which gets selected.

Now, let's look at what's different when instead of having no move constructor, we have a move constructor with a deleted definition (Listing 5).

```
struct DeletedMove
{
    DeletedMove();
    DeletedMove(const DeletedMove&);
    DeletedMove(DeletedMove&&) = delete;
};
```

Listing 5

We can of course still copy this one as well:

```
DeletedMove deletedMove2 = deletedMove;
```

But what happens if we try to copy-initialize from an rvalue?

```
DeletedMove deletedMove2 =
  std::move(deletedMove);
```

Remember, overload resolution tries to find all candidates to do the copy-initialization. And this class does in fact have both a copy and a move constructor, which are both candidates. The move constructor is picked as the best match, since again we avoid the conversion from an rvalue to an lvalue reference. But the move constructor has a deleted definition, and the program does not compile.

> A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [Note: This includes calling the function implicitly or explicitly (…) If a function is overloaded, it is referenced only if the function is selected by overload resolution.(…)] [C++Standard_2]

The function is being called implicitly here, we're not manually calling the move constructor. And we can see that this applies because overload resolution selected to use the move constructor with the deleted definition.

So the differences between not declaring a move constructor and defining one as deleted are:

- The first one does not have a move constructor, the second one has a move constructor with a deleted definition.
- The first one can be copy-initialized from an rvalue, the second cannot.

References

[C++Standard_1] C++ standard: 'Copying and moving class objects', available from https://timsong-cpp.github.io/cppwp/n4659/class.copy

[C++Standard_2] C++ standard: 'Delete definitions', available from https://timsong-cpp.github.io/cppwp/n4659/dcl.fct.def.delete

This article was previously published online at: https://blog.knatten.org/2021/10/

Afterwood

Players in Escape Rooms are set puzzles to be solved in order to win. Chris Oldwood reminisces on old childhood games as inspiration for various programming puzzles.

As I write this, the number one show on the popular streaming service Netflix is called *Squid Game*, a South Korean drama where hundreds of adult contestants compete in a deadly series of children's games for a life changing sum of money. Naturally, the meme generators are working overtime and I can't help but be seduced by thoughts of how you might apply the premise of the show to our own profession. The show has a somewhat macabre element, which I'd rather not dwell on in a family friendly publication such as this, so maybe *The Crystal Maze* would be considered a better format. However, if it's children's games from my era [Wikipedia-1] we're after then I'd need to regress a few more years back into the mid '80s BBC Archives where we meet its spiritual successor – *The Adventure Game*. This show was set on the fictional planet of Arg, which feels like the perfect place for programmers to battle it out.

This also feels like the ideal topic to start testing our contestant's mettle – passing arguments. The scenario I have in mind is one which I personally have spent the past few weeks tearing my hair out over, namely passing arguments with embedded quotes. Shells like Bash and PowerShell support more than one style of quotes (for string interpolation reasons) which makes the simple case undaunting. The show should require something more akin to my recent task which involved passing a non-trivial PowerShell command line to a remote Windows VM from a Bash shell, which in turn invoked a shell script that used SSH to talk to the VM's host and SSH from the host into the guest VM, which in turn had Bash configured as the default shell on the Windows side. That should keep them busy for a while just googling how to escape quote characters in different shells and tools!

The second game I have in mind taps into the *Squid Game* theme, but only in name – *Stuck in the Mud*. On this occasion we're not alluding to the Multi-User Dungeon, which is probably more befitting of the Adventure Game, but the term popularised by Brian Foote and Joseph Yoder. As a system grows over time, close attention must be paid to the architecture and design lest it wind up as the aforementioned Big Ball of Mud [Wikpedia-2] – "a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle". Hence the task our competitors must face would be to make sense of a legacy codebase and refactor it so the unnecessary duplication is removed and it becomes more amenable to change without taking its toll on the team's morale.

Much as I like the play on words in that title, I think there is a better playground game that lends itself more to this particular exercise. In 'Spot', players take it in turns to kick a football against a wall, and if they miss, they lose a life. (Each life is represented by one letter of S.P.O.T. and when all lives are lost the player is out). Not only does the title gel nicely with the idea of refactoring to reduce the duplication of concepts, i.e. a Single Point of Truth (aka SPOT), but the format also suggests the use of limiting the number of test failures allowed as a mechanism for 'lives'. I'm sure the ridiculousness of a scenario where the production code is in an absolute mess but also contains a full suite of tests won't be lost on the contestants. I'm going to have my work cut out for me trying to provide a plausible backstory. (As an aside I'm now beginning to wonder if Kent Beck took his inspiration for "Test && Commit || Revert" from the show as that also seems quite brutal…)

Given the amount of exposure the TV show has had in the media and on the socials in the last few weeks, I don't think it's a spoiler to say that one of the challenges they face in *Squid Game* is a variation of what we Brits called Statues, but they call Red Light/Green Light, where the players have to creep up on the person who's 'it' while their back is turned. I sense there is some additional testing related mileage here under the more 'punful' name Red Bar/Green Bar. Here you can't make progress by adding new features to the product unless the bar is green; when it's red you must fix the code first. That feels a little too much like real life but, in my game, you won't be able to just comment the test out, add an ignore attribute, or make the test run multiple times in the hope it succeeds eventually and manages to hide the lack of determinism.

For the finale I can think of no more difficult a topic for programmers to tackle than that of time. In 'What's the Time, Mr. Wolf?' our contestants will be faced with a series of challenges that require them to correctly calculate dates and times across the globe. For example, one task might be to use Outlook in one country to book a Zoom meeting where all the participants are situated in other countries. You lose a life if at least one person doesn't show up at the correct time. Another would see you execute a multi-currency, multi-month trade where the start and end dates straddle Christmas and Golden Week, during a leap year. I mustn't forget to include a daylight saving time puzzle in there somewhere too!

I'm sure if I cast my mind back far enough to my childhood, I can dredge up some more inspiration. The game of Sardines (a variant of Hide-and-Seek) where you work ever harder to overcome a lack of capacity has an eerily familiar tone to it. Likewise the practice of chaos engineering feels like it has its roots in a game of Knock Down Ginger where you try and switch off random hardware without getting caught by the customer, and I've definitely been involved in code reviews that have felt more like a game of British Bulldog! ■

References

[Wikipedia-1] List of children's games: https://en.wikipedia.org/wiki/List_of_children%27s_games

[Wikipedia-2] Big ball of mud: https://en.wikipedia.org/wiki/Big_ball_of_mud

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