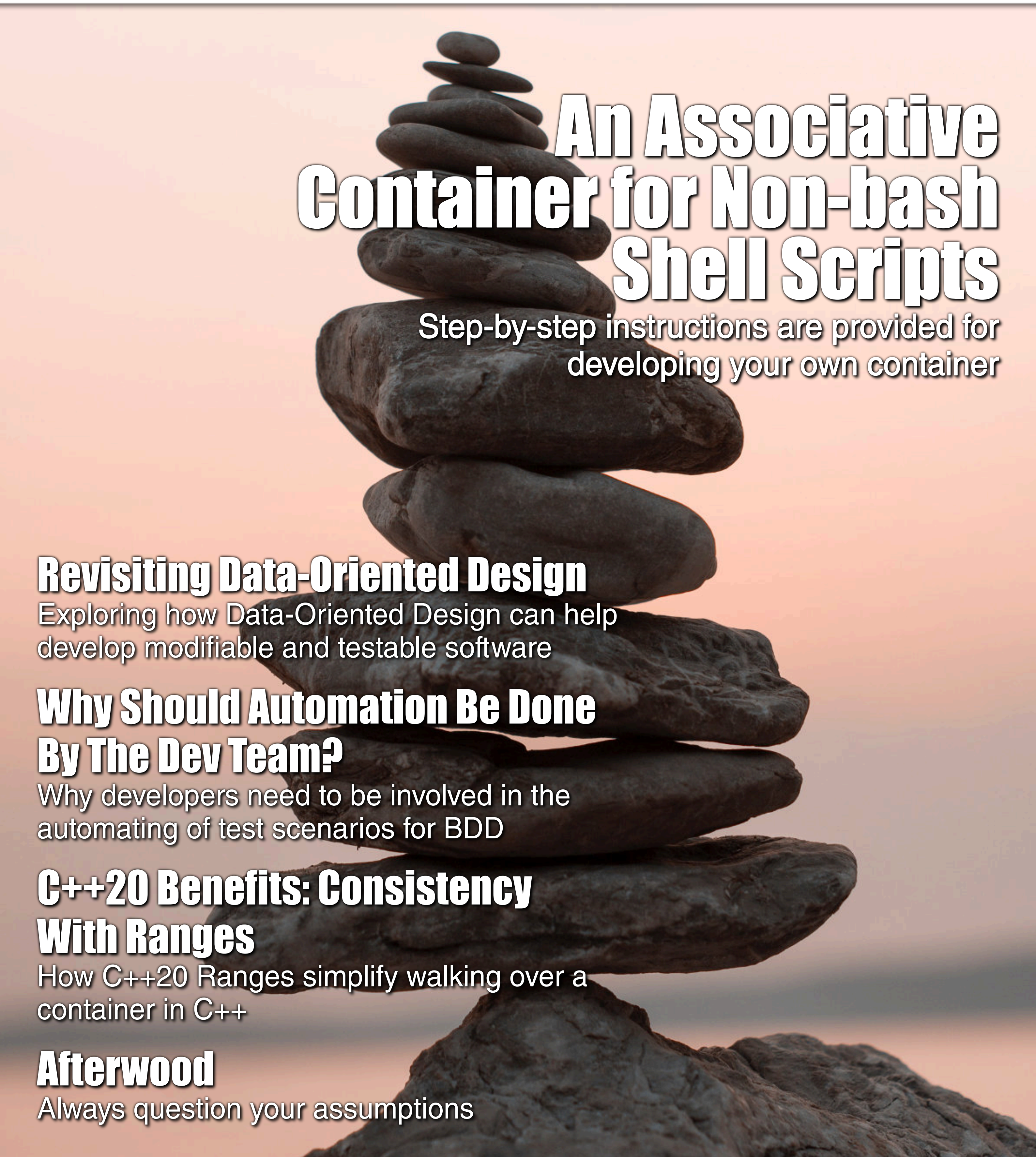
overload 167

FEBRUARY 2022 £4.50



An Associative Container for Non-bash Shell Scripts

Step-by-step instructions are provided for developing your own container

Revisiting Data-Oriented Design

Exploring how Data-Oriented Design can help develop modifiable and testable software

Why Should Automation Be Done By The Dev Team?

Why developers need to be involved in the automating of test scenarios for BDD

C++20 Benefits: Consistency With Ranges

How C++20 Ranges simplify walking over a container in C++

Afterwood

Always question your assumptions

A magazine of ACCU

ISSN: 1354-3172

Join ACCU

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

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

Join now!
Visit the website

professionalism in programming

www.accu.org

OVERLOAD 167

February 2022

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Bekir Dönmez
on Unsplash.

Copy deadlines

All articles intended for publication in Overload 168 should be submitted by 1st March 2022 and those for Overload 169 by 1st May 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

What are you optimizing for?

Sometimes attempts to improve things make it worse. Frances Buontempo encourages you to think about what you're doing when you try to optimise, and to check it really is working.

.Welcome to the first issue of *Overload*, 2022. Before you ask, I haven't written an editorial. I have been fighting a battle with the number of tabs open in my PC's browser. I might be winning, with only 67, after a steady 69. I did try closing several, but unfortunately opened other links while tidying up and kept ending back at 69. Clearly, for me 69 is the optimal number of tabs and 67 is just showing off. That only wasted a few hours of my life. Advent of Code [Advent] took even more time, and is hogging a couple of tabs as we speak.

If you've not come across Advent of Code before, I did a short write up for our members' magazine *CVu* [Buontempo22]. Puzzles are set during December, each having two parts. The first part tends to nudge you towards a simple way to solve the problem and the second part then slaps you in the face by blowing out RAM or similar. There are often warning signs in the description, such as 'exponential' getting a mention, or simply a gut feeling that this might get really big, really quick. Knuth told us "premature optimisation is the root of all evil." Though this quote is about how long something might take to run, rather than memory, it can apply to both. However, speed and RAM tend to tug in opposite directions – it can be hard to do things quickly and use little memory. Caching requires somewhere to store the cache, but can avoid duplicate calculations. It may even slow down your code if it introduces cache misses. Many attempts to solve problems tug in opposing directions. To find out if you have covid, you could take a lateral flow test. The results are available quickly, but may not be so accurate. Instead, you could take a PCR test, but have to wait longer for the results. The PCR – a polymerase chain reaction – requires a few iterations, splitting up DNA and making copies of the target of interest if present, making it much more sensitive than the lateral flow test, which relies on cells in the sample hitting detector molecules leading to a colour change. (Forgive me if my summary of this year's Royal Institute Christmas Lecture is a bit mis-remembered and only vaguely correct. There are some helpful links on their website [Royal Institute21]). Speed versus accuracy is a common optimization tradeoff.

If you are optimizing, you need to decide your requirements. RAM, speed and accuracy might matter, but comprehensibility and even happiness might come into play as well. Loop unrolling tricks may speed up your code, but confuse your colleagues (or your future self) and cause unhappiness. You could experiment with `-funroll-loops` if you are using GCC, rather than try to do this by hand, however the manual says "This option makes code larger, and may or may not make it run faster." [GCC]. The question lurking in the background for any optimization is always, "Has this worked?" For a one-off task, it's all too common to spend more time attempting to automate it than doing the task manually. XKCD [XKCD] reminds us that writing code to automate something might not be the time saver we hoped for. More than that, if you are trying to speed up your code, measure and see what happens. Your instincts may be wrong.

Machine learning frequently involves optimization in one form or another. A fundamental part of the process involves checking the algorithm is working by calculating a score in a so-called *fitness* or *cost* function. Bigger scores are better if you want to maximize something, like profit or happiness, and smaller scores are better for minimization problems, such as travel time or fuel used. The algorithm makes course corrections to get a better score in the fitness function, usually by selecting different, randomly chosen inputs. Using randomness to solve a problem might seem odd on the face of it. If there is time to brute force a solution that might be better. However, some situations have so many possible combinations of inputs it's quicker to try a few and see what happens. One selection might be good enough, job done. Otherwise, try, try, try again. Now, pure randomness might never work. First, the same inputs may be selected several times over, wasting time. Many machine learning algorithms actually do this, though not deliberately. For example, genetic algorithms don't conventionally track what they have tried before. If the algorithm gets stuck in a rut, trying the same thing over and over again, you can try again from the top with different parameters or a new fitness function. Second, a pure random algorithm might never solve a problem: the so-called *bogosort* springs to mind [Wikipedia]. This algorithm checks if the inputs are sorted. If they are great, return them. Otherwise randomly permute and try again. Potentially forever. The internet [Morgan-Mar] assures me there are other even worse variants of this algorithm. In order to optimize my tab count, I have closed that link, so I'll leave you to find out about the sorting algorithm that violates the second law of thermodynamics and report back. Randomness can work, but might not.

Machine learning can also be used to make predictions. There are many ways to do this, but each tries to minimize the difference between predicted values and actual values, again giving another optimisation problem. Often this requires a gradient calculation, in order to move 'downhill' or closer to the required outputs. This number crunching can take a while, so you can speed up the optimisation by using *stochastic gradient descent*. This "inexact but powerful technique" [Stojiljković21] finds the gradient on a small subset of the training data, clearly an attempt to optimise the optimisation. Where will this meta-optimising end? That randomness can be used to solve problems may be a surprise. It shouldn't; trial and error is a sensible way to explore a problem. Furthermore, Stochastic (random) processes underpin many machine learning algorithms. In fact they also form much of finance and possibly some rocket science too. The c2 wiki does question the connection [c2-09] but quickly devolves into flippancy, quoting words allegedly overheard at

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.

NASA: "Come on! You make things sound so difficult. Sending probes to Mars isn't rocket science, you know!"

Now, in order to optimize, we may need to think beyond our initial requirements. Nobert Wiener, a mathematician who made some fundamental steps towards modern artificial intelligence, warned us to be very thoughtful about the "purpose put into the machine." [Wiener60] He reminds us of the tale of the sorcerer's apprentice, who automates a broom to carry water, which it dutifully does, nearly drowning the apprentice in the process. He then says,

> If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere once we have started it, because the action is so fast and irrevocable that we have not the data to intervene before the action is complete, then we had better be quite sure that the purpose put into the machine is the purpose which we really desire and not merely a colorful imitation of it.

There are many similar thought experiments concerning potential problems with automation, AI and optimization. The paperclip maximizer springs to mind. If we task an AI to collect paperclips, and leave it to its purpose, it might put "first all of earth and then increasing portions of space into paperclip manufacturing facilities". [LessWrong]. Fear not: not all attempts to optimize pose an existential risk to the human race, the planet or the universe. 2021's Reith Lectures were given by Stuart Russell and explored 'Living with Artificial Intelligence' [Russell21]. He mentioned Wiener's concerns about fully specifying our objectives. Asimov's 3 Laws of Robotics are an attempt to constrain AI, covering us when we fail to give clear purpose to a robot, and allowing many interesting stories to develop. They are interesting, but fictional. Russell suggested a better solution involves assistance games [Shah19], where a human provides feedback while the AI learns rather than giving a fixed objective up-front. I suspect an analogy about waterfall versus agile is lurking in there somewhere. I'll leave that for you to think about.

When we optimize, we need to keep an eye on our solution, to ensure it is solving our problem and not causing other complications in the process. We may not find the best possible approach and that's OK. Trial and error experiments might find an acceptable algorithm or set of inputs. Acceptable might be good enough. If you need your code to run faster, quick enough is fine. Spending weeks trying to find the quickest possible code might take longer than running a slightly slower version that works. As we try to improve our code's performance, we might need to consider its comprehensibility and the happiness, or otherwise, of future developers. Many clever techniques can speed up code, but might make your head hurt each time you encounter them. Duff's device, a loop unrolling technique, always makes my eyes glaze over. It's also important to realise that a clever technique that works in one situation may not work in another. If your target architecture changes or new floating point operations become available, or you change compiler, old optimization techniques may in fact pessimise your performance. Things change.

What happens if we abandon our quest to optimize and try to pessimise instead? There are various ways to achieve this, for example the 'Multiply and Surrender' strategy, which

> consists in replacing the problem at hand by two or more subproblems, each slightly simpler than the original, and continue multiplying subproblems and subsubproblems recursively in this fashion as long as possible. At some point the subproblems will all become so simple that their solution can no longer be postponed, and we will have to surrender. [c2-14]

This may seem like a case of chronic procrastination, but sometimes switching one problem for another works. However this can lead to analysis paralysis, where you freeze because every possible route forward appears to cause further problems. Multiply and surrender falls into a category known as reluctant algorithms. 'Pessimal Algorithms and Simplexity Analysis' by Broder and Stolfi [Broder84] explores other ways to solve various problems slowly. They open by posing the following problem: "Consider the following problem: we are given a table of *n* integer keys A1, A2, ..., A*n* and a query integer *X*. We want to locate *X* in the table, but we are in no particular hurry to succeed; in fact, we would like to delay success as much as possible." They also consider the sloppiest path problem for graphs, a slowsort and many other problems. Deliberately writing slow algorithms may seem foolish, but it's a great read and might get you thinking.

Sometimes trying to speed up makes things worse. How often have you watched a driver on a crowded motorway switching lanes to get one car ahead? Usually you pass them several times over if you stick to one lane. They have a greater speed than you, because they have travelled a larger distance in the same period of time, but have failed to make better progress towards their final destination. Slow is fast as the saying goes. In fact, traffic flow models are fascinating, which reminds me I should draw to a close because I have an ACCU conference talk to prepare. On a final note, don't forget Goldratt's theory of constraints. All processes have constraints or bottlenecks. If you try to optimise other parts of the process, you are unlikely to make any difference. Counterintuitively, you might find slowing down can give a steady input to the bottleneck which might speed things up overall. The Theory of Constraints "shifts the focus of management from optimizing separate assets, functions and resources to increasing the flow of throughput generated by the entire system." [Constraints]. Here's to a slow and steady 2022.

References

[Advent] https://adventofcode.com/

[Broder84] Andrei Broder and Jorge Stolfi (1984) 'Pessimal Algorithms and Simplexity Analysis', available at https://www.mipmip.org/tidbits/pasa.pdf

[Buontempo22] Frances Buontempo (2022) 'Advent of Code', *CVu* 33.6, available from: https://accu.org/journals/cvu/33/6/buontempo/

[c2-09] 'Rocket Scientist' (last updated 18 October 2009): https://wiki.c2.com/?RocketScientist

[c2-14] 'Multiply and Surrender' (last updated 26 October 2014): http://wiki.c2.com/?MultiplyAndSurrender

[Constraints] 'Theory of Constraints (TOC) of Dr. Eliyahu Goldratt' available at: https://www.tocinstitute.org/theory-of-constraints.html

[GCC] Using the GNU Compiler Collection (GCC) 'Options that Control Optimization' available at https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[LessWrong] 'Paperclip Maximiser', Less Wrong, available at https://www.lesswrong.com/tag/paperclip-maximizer

[Morgan-Mar] David Morgan-Mar, 'DM's Esoteric Programming Languages' available at https://www.dangermouse.net/esoteric/

[Royal Institute21] Royal Institute Christmas Lectures 2021, 'Going viral: How Covid changed science forever', available at https://www.rigb.org/christmas-lectures/2021-going-viral-how-covid-changed-science-forever

[Russell21] Stuart Russell (2021) *Living with Artificial Intelligence* (a set of 4 talks), The Reith Lectures, available at https://www.bbc.co.uk/programmes/m001216k/episodes/player

[Shah19] Rohin Shah (2019) 'Human-AI Collaboration', published on Less Wrong, 22 Oct 2019, available at https://www.lesswrong.com/posts/dBMC63hjkc5wPqTC7/human-ai-collaboration

[Stojiljković21] Mirko Stojiljković (2021) 'Stochastic Gradient Descent Algorithm With Python and NumPy' on *Real Python*, available at https://realpython.com/gradient-descent-algorithm-python/

[Wiener60] Wiener, Norbert. 'Some Moral and Technical Consequences of Automation' *Science*, vol. 131, no. 3410, American Association for the Advancement of Science, 1960, pp. 1355–58, Available here https://nissenbaum.tech.cornell.edu/papers/Wiener.pdf

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

[XKCD] 'Automation', available at https://xkcd.com/1319/

Revisiting Data-Oriented Design

Modifiable and testable software makes life easier. Lucian Radu Teodorescu explores how Data-oriented Design can help here.

If one made a list of the top of the most outstanding C++ talks in the last 10 years, Mike Acton's talk 'Data-Oriented Design and C++' [Acton14] would probably rank very high on it. A relatively recent tweet chain started by Victor Ciura [Ciura21] is just a small confirmation of this. The talk covers the fundamental principles for building software, and shows with multiple examples how to get 10x improvements in performance. For a C++ programmer, such an improvement is close to the holy grail.

The common belief these days is that Data-Oriented Design (for short, DOD) is an approach that focuses on program optimisation, an approach brought into the light by Mike Acton.

This article aims at revisiting Acton's main ideas on Data-Oriented Design and seeing its general applicability for software systems that don't have hard performance constraints. That is, ignoring performance, we are going to focus on the *design* part of Data-Oriented Design.

A recap of Acton's Data-Oriented Design

If you haven't watched Mike Acton's presentation [Acton14], it's best for you to pause reading this article and watch the recording first. The text is waiting patiently.

The 'Data-Oriented Design and C++' talk can be divided into four parts. In the first part, Mike describes the constraints that the game industry typically faces, giving a context and a justification for some problems exposed in the talk. In the second part, the talk focuses on the principles of Data-Oriented Design, rules of thumb and a few myths in Software Engineering – this is, more or less, the theoretical framework of Data-Oriented Design. He then goes to give examples from the game industry on how performance can be improved by using DOD; improvements of 10x can be seen by making some relatively simple transformations, having data usage in mind – this is probably regarded as the most powerful part of the presentation. At the end, the talk comes back to reinforce some principles by drawing some conclusions from the presented examples.

Principles and rules of thumb

We will list here the principles and the rules of thumb that Acton exposed in his CppCon 2014 talk [Acton14], in the second part of the presentation. We will separate out the principles from the rules of thumb, and we will number them so that we can refer to them.

> [DOD-P1] The purpose of all programs, and all parts of those programs, is to transform data from one form to another.
>
> [DOD-P2] If you don't understand the data, you don't understand the problem.
>
> [DOD-P3] Conversely, understand the problem by understanding the data.
>
> [DOD-P4] Different problems require different solutions.
>
> [DOD-P5] If you have different data, you have a different problem.
>
> [DOD-P6] If you don't understand the cost of solving the problem, you don't understand the problem.
>
> [DOD-P7] If you don't understand the hardware, you can't reason about the cost of solving the problem.
>
> [DOD-P8] Everything is a data problem. Including usability, maintenance, debug-ability, etc. Everything.
>
> [DOD-P9] Solving problems you probably don't have creates more problems you definitely do.
>
> [DOD-P10] Latency and throughput are only the same in sequential systems.
>
> [DOD-ROT1] Rule of thumb: Where there is one, there are many. Try looking on the time axis.
>
> [DOD-ROT2] Rule of thumb: The more context you have, the better you can make the solution. Don't throw away data you need.
>
> [DOD-ROT3] Rule of thumb: NUMA extends to I/O and pre-built data all the way back through time to original source creation.
>
> [DOD-P11] Software does not run in a magic fairy aether powered by the fevered dreams of CS PhDs.

These are old principles, but they are even more important these days with the uneven growth of performance for different hardware parts and the ever-increasing demanding needs of the industry.

The three big lies in the software industry

According to Acton, in the software industry we have three big lies that made us move away from these principles:

- *Software is a platform.* He argues, obviously, that the end hardware is the actual platform. Those who believe that software can be the platform are just ignoring the reality.
- *Code designed around model of the world.* Here the argument is a bit more complicated; it involves the confusion between the needs for data maintainability and an understanding of the properties of the data, and also the confusion about what IS-A means in the real world and what it means in software. That is, what can be easily done in the real world cannot be simply translated to software. In a previous Overload article, I also argued that IS-A is far too confusing to be used as a basis for software construction [Teodorescu20].
- *Code is more important than data.* The reasoning against this fallacy starts with [DOD-P1]. If the purpose of any code is to transform data, then data must be more important than code.

An analysis of the DOD principles

While the actual phrasing for some principles may create confusion and may lead to false interpretation, we believe that these are generally true. One reason for accepting them so quickly is because they echo what Brooks had to say in a section called 'Representation Is the Essence of Programming' from his *The Mythical Man-Month* book [Brooks95]:

> Much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a

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

the final problem that we need to solve is typically much bigger than our focus

program lies. Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious. […] Representation is the essence of programming.

Having Brooks backing it up gives DOD a considerable boost in confidence.

We said that the aim of this article is to not-focus on performance, so let's split up Acton's principles and rules of thumb into two categories:

- general software design principles: [DOD-P1] to [DOD-P5], [DOD-P8], [DOD-P9], [DOD-ROT2], and [DOD-ROT3]
- performance-related principles: [DOD-P6], [DOD-P7], [DOD-P10], [DOD-ROT1], and [DOD-P11]

We will entirely ignore the performance-related principles.

The first principle is probably the most important one. It echoes what Brooks also said, and puts the data and data transformation in the centre of all our programming activities. This is obviously true. The important part to notice is that this applies not to the whole program, but also to program parts. That is, regardless of what size a unit of code is, its purpose is to transform data. This applies from small code snippets, functions, and program components to entire programs. This is an important fact, which we will explore in more detail later. Please also note that it's hard to fit classes on the same bill; it's hard to say that the purpose of classes is to transform data – we will cover this later, too.

The principles [DOD-P2]-[DOD-P5] are somehow expressing the same idea: a software problem is a data transformation problem, and the solution needs to be closely related to the data. This can be viewed as a direct consequence of the first principle.

Principle [DOD-P8] extends the data-centric view from just programming to related activities: maintenance, debugging, and even usability. Properly justifying this principle is a more complex endeavour, so we will skip it. What is interesting here is the idea that we often need to look at a larger context when designing a solution. It's the same idea that is expressed in [DOD-ROT2] and [DOD-ROT3]. And this perfectly aligns with a quote from Eliel Saarinen that Kevlin Henney often brings up in his talks:

Always design a thing by considering it in its next larger context

In Software Engineering, we frequently focus on a narrow part of the universe and try to *solve* that part. But that is just part of the problem; the final problem that we need to solve is typically much bigger than our focus. Acton also reminds us that we often have additional duties, not just the coding part. [DOD-ROT3] is especially intriguing as it makes us think outside the box. For example, some problems are better solved at compile-time, and not at run-time; this is also appropriate not just for performance, but for modifiability, testability, and other concerns that we might have.

Finally, [DOD-P9] is a truism, but something that we software engineers probably need to hear more often. For some strange reason, we tend to create problems for ourselves out of thin air, as if the problems that we already have are not enough.

To summarise, these principles state that data and data transformation are at the heart of software engineering, and that we should also be aware of the context of this data to fully meet our goals. We argued that all of these make a lot of sense as general principles for Software Engineering, not just for performance-focused development.

Building software with Data-Oriented Design

The name Data-Oriented Design indicates that we can use this approach to *design* applications. We try to use these data-centric principles, together with the modular decomposition, to sketch a process of designing an application. We will attempt a hierarchical decomposition of the problem, so we will apply the same techniques at multiple levels.

For our purposes, we assume that the process follows somehow an ideal (waterfall) model: we work on a level of abstraction, we fully perform our duties there, and we never invalidate our assumptions. This rarely happens in practice, but let us simplify the exposition here.

Let us take an artificial problem as our running example. Let's assume that we are building a web service that can be used to do image processing. Some descriptions here are a bit fuzzy on the actual problem to be solved; in real-life, the engineer should fully understand the problem before attempting to solve it.

Top level

We start by considering the problem as a whole. Based on [DOD-P3], we need to understand all the data around this web service to understand the problem to solve. We must understand the inputs of the service (e.g., images from users, images from a database, data models, commands for processing the images, the parameters for image processing, etc.) but also the data produced by the system (e.g., other images, description of the image properties, etc.) – this is typically called the API of the service. But this is not all the data that needs to be transformed by the service. The service also has interactions with the cloud platform, with the DevOps team, with other nodes in the cloud, etc. For example, logging and monitoring are two important aspects of any web service. Moreover, adjacent concerns like how often the system will be restarted, what is the usage scenario, what are the peak hours/seasons, etc., contribute to the data exchange that the program needs to properly manage. And, even if performance is not a major concern, interaction with the actual physical hardware also needs to be considered (for example, for data reliability concerns, which can dramatically change the architecture of the solution).

These may seem a lot of concerns to be taken care of, but, they all deserve our attention. Applying proper engineering to the problem requires us to consider these aspects.

So far, we have just considered the data that our software needs to interact with. But that's not everything that we need to consider at this level. We need to look now at the next larger context to understand more about the problem at hand (see [DOD-P8]). We need to look at whether the system will be used by humans or other services, what the long-term plan is for the customer paying for this service, what are other services used by the same customer, what are the forces acting on the customer, etc. These all

There are multiple approaches to breaking down a system into sub-systems. It is generally accepted that good decompositions will minimise coupling between the smaller parts.

can influence how the customer demands new features or bug-fixes for the new service. Based on these, we can make decisions on how the application needs to be structured, and, more important for our discussion, what additional data we need to maintain to keep the system operational.

Only when we have all this information, when we know what data needs to be transformed by our system, what additional data we need to support, and how our service interacts with the platform, do we can fully understand the problem that we are trying to solve.

Breaking down into modules

After we fully understand the problem that we need to solve, i.e., we fully understand the data (see [DOD-P2]) we are ready to decompose the problem into smaller parts, i.e., modules. In this context, the term *module* may refer to larger application modules (e.g., libraries) or smaller parts of the application (e.g., translation units, functions)

There are multiple approaches to breaking down a system into sub-systems. It is generally accepted that good decompositions will minimise coupling between the smaller parts. Again, depending on your perspective, there are various approaches to ensure low coupling (by change rate, by functional role, by organisational structure, etc.). But we can also use Data-Oriented Design to offer us guidelines on how to perform the decoupling:

- group together data that is used together
- separate data that is not used together

The usage of the data gives us boundaries for our modules. If there are two completely independent data flows, then it probably makes sense that those two flows be in separate modules.

Passing data across modules is more expensive (modifiability, and typically performance) than passing data inside the same module. Thus, one can find the module boundary at the points in which data is stable enough to form an API. This is one important aspect to take into consideration when decomposing the problem.

Another significant aspect of performing a Data-Oriented decomposition is, following [DOD-P1], making sure that the submodule is a proper abstraction for transforming data. One should be able to think of the new submodule as a large function (not necessarily pure) that takes some input data and produces some output data.

With these in mind, we can identify the requirements for creating a module:

- properly identify the input data for the module
- the input data needs to be stable enough to act as the module API
- properly identify the output data for the module
- the output data needs to be stable enough to act as the module API
- analyse the next larger context to properly know the constraints that are imposed on the new module
- ensure that the module can be easily thought of as a large function that transforms input data into output data

Consider the system depicted in Figure 1. Traditionally, one would decompose the system by considering the nodes; the fewer arrows a node has, the less coupled the node is. In a Data-Oriented approach, we should view the links as the main elements.

The system depicted in Figure 1 can probably be decomposed into two parts: one that contains nodes N1-N7, and one that contains the nodes N8-N11. Looking at the data flow, one can easily see how the two modules would be assimilated as two large functions.

But maybe the data link d10 is not the most stable data channel. It may frequently change with d11 and be very coupled to it. In this case, perhaps it makes more sense to put n7 into the second module, and have the second module depend on two data inputs.

Ideally, each module would have only one data stream as input, but there are countless cases in which multiple input data streams are required. This is especially applicable to modules that maintain state, and interact with other modules in multiple ways.

We might find oftentimes that a data-centric decomposition leads to the same results as a functional decomposition. However, the focus is on the data, not on the code that just transforms the data.

In our imaginary application, we might decide to have a high-level module that deals with HTTP handling, and one module for each type of request handled by the service. These high-level modules can then be subdivided even further into smaller modules, until the decomposition makes sense. This follows because the data required for processing a request type is typically independent of the data required for processing another request type. All the modules created for handing different requests will interact

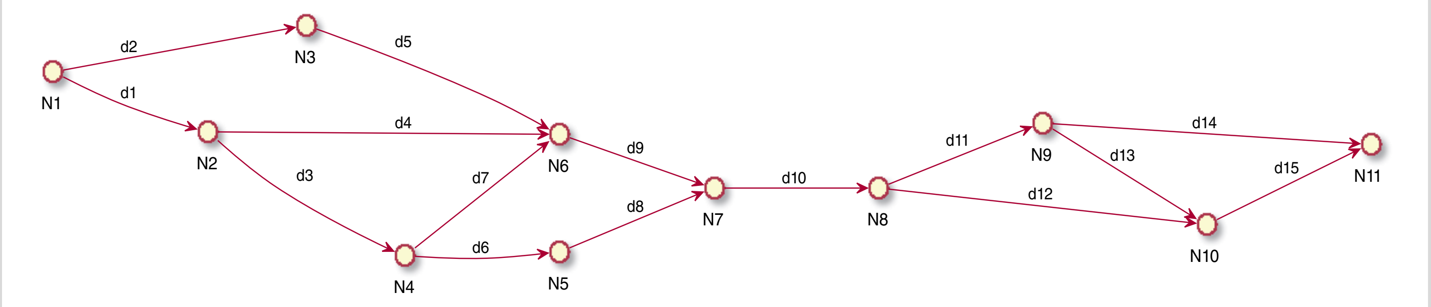


Figure 1

If the purpose of the code is to transform data, then it makes sense to make explicit the data we are transforming by making global data either an input or an output of the function

with the module for HTTP handling, but the data passed between these modules is relatively stable.

And, speaking of HTTP services, maybe it's worth emphasising one important aspects of APIs. From a modifiability perspective, it's much better to have an API based on HTTP and standard conventions rather than a TCP-based user-defined protocol. The HTTP-based API tends to be more stable, while the TCP-based one may require many changes. Choosing an API based on stability is a consequence of using a data-centric approach.

On reusable libraries

One question that arises is whether Data-Oriented Design encourages or accepts reusable libraries.

The simplistic answer would be NO. A reusable library tends to focus on the code, and abstracts the problem away. In general, different problems require different data; different data means that it's hard to find a stable API for reusable code libraries. For example, an image can be represented in countless ways; over-generalising a library to process all possible types of image types is an effort that is not needed for most of the applications.

On the other hand, one can easily imagine that the same data structures and same algorithms appear in multiple places in the same application. Maybe the application supports only a limited set of image formats, and this needs to be supported for all the operations our service provides. Or perhaps the linear algebra to be used in the application can be easily abstracted out.

I want to avoid putting words in Acton's mouth, but I believe that a pragmatic approach to Data-Oriented Design would accept a limited set of reusable libraries. As long as we don't try to build reusable libraries out of everything; i.e., avoid creating problems that we don't have.

Use of classes

When we were analysing the DOD principles, we said that classes are a bit odd with respect to these principles. We agreed that the purpose of all code is to transform data, but, if viewed as code, classes don't typically transform data. Let us analyse a bit more the meaning of this.

When we say classes, in a language like C++ (or C#, or Java, or any language that supports OOP), we typically mean three things:

- a definition of the data layout in memory
- behaviour associated with the data stored in the class
- encapsulating the data of the class through behaviour

The data-layout part of the class is well aligned with DOD. After all, Data-Oriented Design deeply cares about how the data is arranged when solving problems. The problem comes from the other two properties of the class.

First, and more importantly, encapsulating the data inside the class means hiding the data. This is completely against Data-Oriented Design. We cannot have data-orientation when we make extra effort to hide the data. In the spirit of [DOD-P9], we are just creating problems for ourselves, without any real benefit.

Associating behaviour with data is also a bit awkward. This is because we move the focus from the data to the code near the data. We are privileging code inside the class to the detriment of code outside the class. All code should be equal, and inferior to data.

In an ideal world, code should be put in pure functions; impure functions are to be avoided, if possible. If the purpose of the code is to transform data, then it makes sense to make explicit the data we are transforming by making global data either an input or an output of the function.

Pure functions are not quite compatible with class methods, so this is another indication that we should not pack together code with data.

Instead of using classes for representing data, we should use plain structures.

As odd as it sounds, functional languages are more suited to Data-Oriented design than Object-Oriented languages, if we entirely ignore the performance aspect.

Completeness

Unlike other methods, applying Data-Oriented Design doesn't need to be complete. One can design various parts with a DOD mindset, while leaving other parts to use a code-centric approach (for whatever reason).

Similarly, one can refactor legacy systems and introduce DOD in parts where the system most benefits from it. One can hope to improve performance, or maybe maintainability and debug-ability, and can apply DOD just to those parts of the system.

As long as developers don't switch too often between DOD and non-DOD code, creating a lot of confusion, it's fine to have both approaches in the same system.

Therefore, if the ideas presented here are appealing, the reader might consider a gradual deployment of DOD practices in their codebase.

Modifiability

Software is not as hard as a rock. This is why it's called *soft*. Typically, software is as fluid as a river is, so perhaps *fluidware* is a better term.

So, how would Data-Oriented Design fare regarding modifiability? Let's look first at the costs of change. Here we have both good news and bad news.

The bad news is that certain parts of the code might change more often if DOD is strictly applied, and especially when performance is a major concern. Let us take an example from Acton's presentation. One technique the author uses to improve the performance and "make the code more data-oriented" is to pull out conditions from loops and from functions with a remark that sounds like "if one has a boolean flag then one probably has multiple types of objects; one should create different data structures and different execution paths". That is, each time one feels like adding a boolean flag, one should probably reimplement a whole chain of processing. That sounds a bit expensive in terms of programming time.

To generalise this, if the data changes, the problem changes, and we have to write new code. This is the true implication of [DOD-P4] and [DOD-P5].

But, if we turn this around, this is also good news: if the data doesn't change, then the code we might want to change is minimal. There is probably a bug in one small data transformation. As the data doesn't change, the flow cannot change; if the flow doesn't change, then probably the bug is in one small part of the flow. And the change required is pretty small, as there isn't much we can do to change the code without changing the data.

I view this as a trade-off in modifiability: the more data-oriented a system becomes, the harder it is to make a change when the data needs to change; the less data-oriented a system is, the more difficult it is to make a change when data remains the same.

Let us briefly cover debug-ability as part of modifiability.

A typical data-oriented application is harder to debug with breakpoint, as we need to look at larger volumes of data, and how the data changes over time. Any data-oriented program must have easy ways to extract data out of the system. One cannot properly apply DOD to a system in which it is hard to get data out once the code is running. One quick-and-dirty technique is to dump data to a file.

Once one extracts the data from the system, then, as the system is data-oriented, one can make many predictions for the running application. It should be relatively easy to figure out where the problem lies; the data tells it all.

Probably one shortcoming of this is that the developer must not be afraid of looking at potentially large volumes of data. Personally, I don't see this as a problem, but I know people who are not comfortable with this approach.

Testability

Being able to test the code written is almost as important as writing the main application code itself. So, Data-Oriented Design needs to behave well regarding testability if we were to consider it a viable technique.

First, the data-oriented code will be driven by data. There are no other code-dependencies. Thus, one can easily create tests that provide some input data and check whether that data is correctly transformed. I would argue that regular DOD code is more testable than regular OOP code.

Furthermore, as discussed above, the ideal DOD code would consist of pure functions. This makes it much easier to test compared to traditional OOP systems, in which the global and shared state is always a problem.

However, it's not just unit tests that benefit from DOD approaches. Integration tests also become easier to write. We argued that a module, at any level, can be thought of as one big function with some inputs and outputs. That is, it doesn't matter what size of module we are testing: the strategy applied to unit-tests can be applied to integration tests as well. The only inconvenience of having integration tests rather than unit tests is that the pairs of inputs and outputs become far more complex. However, this has to do with the inherent complexity of integration testing, not the method used to write our code.

As with modifiability, the main downside is that, whenever data changes in the code, a lot of the tests need to be updated too. The effort of updating the tests should be proportional to the effort of changing the code.

A refactoring story

I feel that this is the right time to share a personal example that is connected (at least partially) with Data-Oriented Design. A couple of years back, as an architect, I was tasked to help a (newly formed) team refactor a legacy module. The code was so bad that the team supporting it refused to fix bugs when they were shown the root cause by saying: "if I touch this part of the code, everything will break". The module contained more than 600 files generated by a huge state machine; the state machine was so complex that saying it was over-engineered seems like a compliment. One of the most important classes in the module contained about 70 pure virtual functions, and it was derived in strange and unexpected ways. On top of that, the threading was a complete mess.

After an initial assessment of all the problems of the module and the strategies that could be used to simplify it, I sat with the team to make a concrete plan for improving it.

The first thing we did was to define what data is being processed by the module. We analysed the data that must be stored by the module (it was a module that had responsibility for keeping some system state). Then we analysed how other modules would interact with this module, in terms of needed data. To our surprise, we found out that the data required by the module was much smaller than we initially thought.

That was the most important step of our effort. Once the data was clear to all of us, we were able to quickly identify the boundaries in which we operate. Without looking at the detail, we knew how the system could be dramatically improved – we knew all the possible ways in which the data could be transformed.

Of course, the new data structures were much better than the old ones. The crazy state machine, with all the generated code and the glue code, was reduced to something that was less than 3000 lines of code. The threading was fixed by using tasks and a clever thread-safe copy-on-write technique. In the end, the whole effort was a great success.

For this effort, we didn't fully utilise Data-Oriented Design techniques. And certainly, performance was not a major concern. However, the upfront discussion on the data structures was the biggest step forward in the entire process. It was at that time that I realised that Data-Oriented Design should be more about *design* than about performance.

Conclusions

If Mike Acton had named his approach Data-Oriented Optimisation[1], then I would probably have quickly dismissed it as something that is important in certain domains, and that maybe it doesn't have universal applicability. But the approach contains the term *design* in its name. And *design* has a more universal meaning in Software Engineering. Most of what we do is design. So, an analysis of the method focusing on the *design* part and ignoring the performance part was necessary.

In this article, we have analysed the principles of Data-Oriented Design. While they are a bit verbose, they do make sense – they echo well-accepted principles in our field. Thus, if the principles hold, then it follows that we should use DOD approaches more frequently.

We then went on to analyse the implications of using DOD to design complex applications. We found out that, even though the implications are different from what we are typically accustomed to in OOP, DOD can be used successfully. Assuming that the data doesn't change often, DOD can provide better modifiability, and, in general, it provides better testability.

All these make Data-Oriented Design a set of practices that should be used more often in Software Engineering. Maybe we don't get 10x improvement in performance, modifiability and testability. However, considering the state of our industry, any visible improvement is highly welcomed. ■

References

[Acton14] Mike Acton, 'Data-Oriented Design and C++', *CppCon 2014*, https://www.youtube.com/watch?v=rX0ItVEVjHc

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

[Ciura21] Victor Ciura, 'Unpopular opinion (in the C++ community)…', Twitter, https://twitter.com/ciura_victor/status/1463280526647865353?s=21

[Teodorescu20] Lucian Radu Teodorescu, 'Deconstructing Inheritance', *Overload* 156, April 2020, https://accu.org/journals/overload/28/156/overload156.pdf

1. Come to think of it, DOO doesn't sound that bad. And it forms a nice opposition to OOP. One is data-first, the other one focuses on programming, i.e., on code.

An Associative Container for Non-bash Shell Scripts

Basic shell facilities don’t provide associative containers. Ralph McArdell shows you what to do if you need one.

Have you ever found yourself stuck with having to write a *nix/ *nux shell script that cannot assume that *Bash* and system core utilites having GNU extensions are available and so specify the shell processor simply to be `/bin/sh` and then found you could really use some sort of container to store multiple values?

This happened to me a while ago and I thought I would attempt to create an associative container abstraction that could be used with only basic shell facilities.

Basic selection

Having decided to use ‘only basic shell facilities’, and not being a shell scripting guru, the next question was just what are ‘basic shell facilities’? What facilities are available for **`sh`**? In fact what shell is used when we use `/bin/sh`? It seems that these days `/bin/sh` may simply link to some other shell program such as *Dash*.

A little research lead me to the Open Group’s [OpenGroup] page on the POSIX (IEEE Std 1003.1-2017) ‘Shell Command Language’ [ShellLang]. It seemed that trying to use only the shell language facilities as described on this page would be the most portable solution, assuming as few facilities would be available as possible.

Not quite basic

After reading through the specification of the Shell Command Language, it became apparent that one feature that I would *really* like to use was not available as standard. The **`local`** utility used to declare variables local to a function call is not *required* to be implemented but is sort-of reserved in that the results of using **`local`** has unspecified results. It seems local variables within a function were considered but rejected, though the identifier **`local`** was reserved just in case local function variables make it into a future version of the POSIX standard [ShellLangRationale].

I decided that I would use **`local`** – at least until it proved a problem, whereby most uses could probably be replaced with careful use of global variables. The exception would be cases where recursion was required. However, I would assume local variables were only in scope in the call of the declaring function, and not in functions called from the declaring function call – similar to most other languages’ local variable scoping rules.

The requirements

In the end it turned out that in addition to **`local`**, some basic use of core utilities were required. The full requirements to use the associative container shell script library are:

- the POSIX Shell Command Language, as described at [ShellLang]
- the **`local`** utility
- **`echo -n`** (the use of **`-n`** is not strictly portable)
- basic use of the **`tr`** character transform utility
- basic use of the **`sed`** stream editor utility to perform global subsitutions.

What’s in the box?

Having decided on a super-set of the standard POSIX Shell Command Language to use, the next thing was to look at the available facilities and work out how they could be used. Some of the facilities (or lack of) that are of note are mentioned below.

First off, flow control contructs – **`if`**, **`elif`**, **`else`**, **`case`**, **`while`**, **`until`**, **`for`** – are supported, as are user-defined functions. However, returning values from a direct function call (one that does not start a sub-command shell in a separate process) is limited to the numeric exit status – although the Boolean-like **`true`** and **`false`** can be used for predicate functions that can be called and tested in flow control constructs such as **`if`** or **`while`**. This is because **`true`** and **`false`** are built-in utilities that return with an exit code of 0 (success) or a non-zero value (failure) respectively.

To return a copy of an expression, as is more familiar, a user-defined function can be called using *Command Substitution*, which executes the function in a separate, presumably forked, sub-shell process. The output to **`stdout`** from a *Command Substitution* call is captured and effectively become the function call’s return value. Note that *Command Substitution* is also commonly used to capture the output of commands into variables. However, creating and destroying the sub-shell around the function call is expensive.

Variables store string values. Sometimes these can be interpreted as numbers. There are some limited operations that can be performed on a variable’s string value:

- Various error or default/alternative value substitution actions around values being null, set and not null or unset.
- Character length of the string value of a variable.
- Removal of a pattern matching the smallest or largest prefix or suffix of the variable’s value from the variable’s value.

Notably, as far as I could see, there is *no* facility to more directly reference variable values’ characters or substrings other than the prefix/suffix pattern match removal operations.

Arithmetic can be performed on numeric values using *Arithmetic Expansion*.

Facilities often used on the command line such as redirection (**`>`**, **`<`**, etc.), pipelines (**`|`**), and-or lists (**`&&`**, **`||`**) , asynchronous execution (**`&`**) and so on are supported.

Ralph McArdell Ralph started programming around 1980 in the 6th form in BASIC on a teletype over a 110bps dialup modem. Since the mid 1990s he has been a freelance developer using a variety of systems and programming languages, with an emphasis on C++. He has been an ACCU member since the turn of the millenium and helps organise ACCU London meetings. You can contact him at ralphmcardell@gmail.com

with no true random access operations into a string, the format of a dict was going to have to be sequential

There is a set of special variables. Of particularly interest are **`$@`** and **`$#`** for the list of parameters passed to the script from the command line or to a function call and the number of such parameters, along with the **`set`**, **`unset`** and **`shift`** built-in utilities that allow control and processing of script and function call parameter lists.

Script source files can be included in other script source files using the dot (`.`) built-in utility. The *bash* style **`source`** synonym for dot is not supported.

The shape of code to come

By now, I had an idea of the general shape of the project. The first order of business would be to decide on a name and use it, or a form of it, as the basis for file names and, given the lack of namespaces, as a prefix to function names and the like.

The implementation code would be contained in a shell script file intended for inclusion using `.`, the dot built-in utility, in client scripts. Such included files do not need to be executable themselves. Sometime later I found out that there is a convention that such included shell script library files should have a *.sh* suffix.

The public, client, API would consist of a set of functions, most of which would return string values. To make the use of these functions more familair they would be designed to be called using *Command Substitution*.

Any private internal implementation details – helper functions, global variables and the like – would have uglified names involving prefixed and suffixed double underscores.

Given that values are strings, the associative container instances would have to be represented as a string. Instance creation operations would have to return the string-instance value. Operations that do not modify an instance would take an instance as the first parameter to its implementation function. Operations that modify an instance would both take an intance as the first parameter and return the modified instance. Note that all parameter passing and value returning is effectively *by value*, so lots of potential copying.

What's in a name

The first point of action was to decide a name from which the script library file name and library script identifier names can be derived.

The name I chose, following in the footsteps of Python, was ***dict***.

The library file name was also originally just `dict` but later was changed to `dict.sh`, following the discovery of the previously mentioned convention.

What's a dict to do?

Obviously *dict* instances would be of a *dictionary* type – also known as *maps*. These are associative containers whose entries are *key:value* value pairs that allow *value*s to be looked up by their associated *key*. They can be *ordered* or *unordered*.

Given the limited options available with the facilities of the POSIX Shell Command Language, I decided it would be much simpler to opt for *dicts* being *unordered*.

The basic operations required would be:

- Create *dict* instances.
- Add and update entries in a *dict*.
- Lookup and return a value in a *dict* given its associated key.
- Remove an entry from a *dict* given the associated key.

Some additional operations also proved useful:

- A function to return the size of the *dict*, being the number of entries.
- A predicate to check whether some value, which of course is a string, appears to be a string formatted as a *dict*.
- A for-each operation that calls a function for each entry in a *dict*.
- A customisable pretty-print function to pretty print a *dict* in a user-controlled fashion.
- To aid debugging a function to print a *dict* string with special unprintable characters translated to printable characters.

One capability that needed to be added explicitly is the ability to nest one *dict* as a value of another, preferrably to an arbitrary depth.

The dict data format

As the only data type to play with is the string and with no true random access operations into a string, the format of a *dict* was going to have to be sequential, with an appearence of random access only – think tape access rather than disk. This sort of thinking lead to vague memories of some of the lesser used control codes in the ASCII (and thereby the Unicode®) character set. The ones of interest here are the *separator* control codes. Checking a convenient ASCII table source [AsciiTable] we see these are:

- FS (value hex 1C) – File Separator
- GS (value hex 1D) – Group Separator
- RS (value hex 1E) – Record Separator
- US (value hex 1F) – Unit Separator

On the one hand, using these values to separate parts of a *dict* instance means keys and values cannot contain them. On the other hand, for the intended use cases this should not be much of a problem. Where it would be an issue would be keys or values being binary data but this is not a primary use case. If storing binary data in a *dict* is required then the data can be encoded, for example with *base64* encoding.

The current layout of a *dict* in Extended Backus Naur Form [EBNF] is as shown in Figure 1, overleaf.

There are several things to note about the layout. The first is that a *dict* instance has two main parts: the header followed by zero or more entries.

The header consists of an initial *dict* identifier 'chunk' consisting of the four characters 'DiCt' surrounded by an ASCII Group Separator character

the characters that could cause confusion are substituted for different sequences on entry with the reverse substitutions being applied on value extraction

```
            dict = header, { entry };
          header = dict-type, entry-separator,
                   version, field-separator,
                   size, entry-separator;
           entry = key, field-separator, value,
                   entry-separator;
       dict-type = GS, "D", "i" "C", "t", GS;
         version = digits, ".", digits, ".",
                   digits;
            size = digits;
             key = string;
           value = string | prepared-dict;
 entry-separator = field-separator,
                   record-separator;
 field-separator = US;
record-separator = RS;
   prepared-dict = ? a dict instance that has had
                     its special separator
                     characters substituted so it
                     can be safely used as a
                     nested dict value
                     ?;
          string = character, { character };
       character = ? any valid character except
                     ASCII FS, GS, RS, US ?;
          digits = digit, { digit };
           digit = "0" | "1" | "2" | "3" | "4" |
                   "5" | "6" | "7" | "8" | "9" ;
              GS = "\0x1D";
              RS = "\0x1E";
              US = "\0x1F";
```

Figure 1

on each side. This is followed by metadata fields for the version of the *dict* structure (currently '1.1.0') and the size of the *dict* being the number of entries, which starts at zero (an empty *dict*) and is updated as entries are added and removed. Note that each field is followed by an ASCII Unit Separator character, both in the header metadata record and within entries. The end of the metadata record is signified by an ASCII Record Separator character.

The following entries section consists of *key:value* pair entry records. As with the metadata record each key and each value is followed by an ASCII Unit Separator character, and each entry record is terminated with an ASCII Record Separator character.

Keys can be any string of characters except those used as separators – currently the ASCII GS, RS and US characters are used, but the FS – File Separator character might have a use in a later update, so I suggest it is best to not use any of the four ASCII separator characters.

The same restrictions apply to entry values, except that a *dict* can be nested as a value of another *dict*. In these cases the characters that could cause confusion are substituted for different sequences on entry with the reverse substitutions being applied on value extraction.

The dict API functions

The current set of *dict* API functions are:

- **`dict_declare`** to create a *dict* and optionally populate it with one or more initial entries.
- **`dict_set`** to add or update one or more entries to a *dict*.
- **`dict_get`** to return from a *dict* the value associated with the provided key, or an empty string if not found.
- **`dict_remove`** to remove an entry from a *dict* given its key.
- **`dict_size`** to return the number of entries as stored in a *dict*'s metadata record's *size* field.
- **`dict_is_dict`** – a predicate function – to check if a value is a *dict*.
- **`dict_for_each`** to iterate over the entries of a *dict* calling a function for each entry passing it the entry's key, value, one based computed record index and any extra parameters passsed to **`dict_for_each`**.
- **`dict_count`** returns the count of the number of records which should be the same as the value returned by **`dict_size`**. It uses **`dict_for_each`** and its computed record index to iteratively count the number of entries. The main use of **`dict_count`** is for testing.
- **`dict_pretty_print`** prints the entries of a *dict* according to a caller provided format – itself specified by a *dict*. Uses **`dict_for_each`** to recursively iterate over entries and nested *dict* entry values. This is a case where local variables are really required.
- **`dict_print_raw`** to print the raw *dict* string with the unprintable separator characters translated to others. Mainly useful for debugging.

In addition there are *simple* versions of functions that insert or extract values: **`dict_declare_simple`**, **`dict_set_simple`** and **`dict_get_simple`**. These function act as their non-simple versions except that passing *dict*s as values is not supported. They assume entry values inserted or extracted are *not dict* strings and thus elide the checks and special handling required for *dict* values.

Finally there is a function that is intended to be used with **`dict_for_each`**, **`dict_op_to_var_flat`**, that creates a variable for each entry in the *dict* based on the entry's key name. The 'flat' part of the name is intended to indicate that the function does not recurse down into nested *dict* values.

Slice and splice

There are some basic private internal constants and operations to aid in building and updating *dict* strings. For example the initial empty *dict* is a fixed string as the only variable part is the *size* metadata field value and initially this is always 0.

Some small functions help out with chores, such as ensuring keys are followed by the US field separator character and building an entry string by combining such a decorated key with a value and the US RS entry separator character sequence.

Locating entries is more interesting. The value associated with a key is required as the result. First the header is stripped from the *dict* (remember parameters are passed by value so in fact the header is stripped from a *copy* of the *dict*) by using the remove-smallest-prefix operation `${dict#hdr-pattern}`, where `hdr-pattern` is the fixed part of the header including the `entry-separator` with the `size` field value wildcarded.

Next the rest of the *dict* string up to and including the required key field is removed using the remove-smallest-prefix operation again, this time with a pattern specifying all characters up to the preceding `entry-separator` and the decorated key value. This leaves the start of the *dict* substring copy with the value associated with the key followed by all the subsequent entries. Finally, all these subsequent entries that follow the value are removed using the remove-largest-suffix operation `${value_plus%%pattern}` with a pattern specifying the `entry-separator` and all following characters, leaving just the value.

If removing the *dict* portion up to the required value did not remove anything then there was no pattern match meaning the key was not found in the *dict*. In this case an empty string is returned.

Add or update is similar except in this case if a key is found the prefix and suffix parts of the sliced up *dict* around the associated value are kept and the new value spliced in between them, the old value being discarded. In the case a key is not found a new entry is created and appended to the end of the *dict*.

Entry removal again consists of breaking the existing *dict* into parts and splicing parts back together. In this case the parts spliced together to form the result are a prefix part up to the key of the entry being removed and a suffix part from the start of the entry, if any, following the removed entry's entry separator sequence.

A wrinkle in the above is the updating of the *size* metadata field. This is performed if required after the *dict* (copy) has been spliced back together and is done by stripping the header and then appending the resulting entry records to a newly built header with the updated size value using string concatenation of the various substring parts.

`dict_for_each` first strips off the header of its passed *dict* copy then repeatedly extracts the key and value of the initial entry record in the resulting entry records and then removes the whole of the initial entry record, until eventually there are no more entries to remove. On each iteration, a function whose name is passed as the second parameter to `dict_for_each` is called and passed the extracted key and value values and a one-based computed record number along with any parameters passed to `dict_for_each` following the second function name parameter.

Being prepared

If the value of an entry to add or update is itself a *dict* then it has to be modified before being inserted; otherwise, its entries can interfere with operations on the outer containing *dict*. Likewise, if the value of an entry located with `dict_get` is a modified nested *dict* then it has to be modified back to its original state before the value is returned.

`dict_declare` and `dict_set` check to see if a value to add or update is a *dict* and if it is it is prepared for nesting. Likewise, if a value extracted by `dict_get` appears to be a prepared nested *dict* then it is prepared for unnesting.

The prepare-for-nesting operation inserts an ASCII GS character before every `field-separator` and `record-separator` – which are the ASCII US and RS characters respectively. This prevents the matching that occurs with the slicing and splicing of *dict* operations from matching any fields or entries in the prepared for nesting *dict* value. For example, when stripping away entries before the value of an associated key the pattern used to match the smallest prefix to remove is everything (`*`) up to an `entry-separator` followed by the decorated key, being the key followed by a `field-separator`, which for a key value of "KEY" expands to:

```
match = [string], US, RS, "K", "E", "Y", US;
```

If a nested *dict* value happened to also have an entry with a key value of "KEY", and this occured before the entry being searched for, if the nested *dict* entry had not been prepared this entry would be found. However, after preparing for nesting, the entry fragments that would have matched would have the following sequence:

```
prepared-averted-match = GS, US, GS, RS, "K",
  "E", "Y", GS, US;
```

which fails to match both as the prepared modified `entry-separator` sequence is now `GS, US, GS, RS` and as if that were not enough the modified decorated key has become `"K", "E", "Y", GS, US`.

The prepare-for-unnesting reverses the effects of preparing for nesting. It replaces all occurrences of `GS, US` with `US` and every occurrence of `GS, RS` with `RS`.

These transformations work for multiple levels of nested *dict* values. Each additional level of nesting causes an additonal `GS` character to be inserted before each `US` and `RS` character. So if a *dict* has the following sequence of characters for an entry:

```
unnested-entry = "K", "E", "Y", US, "V", "A",
  "L", "U", "E", US, RS;
```

Then inserting the *dict* into another *dict* tranforms the entry character sequence like so:

```
nested-entry-level-1 = "K", "E", "Y", GS, US,
  "V", "A", "L", "U", "E", GS, US, GS, RS;
```

If the second *dict* is then inserted into a third, the already-prepared-for-nesting nested *dict* value is modified again when the second *dict* is prepared for nesting, and the entry sequence is further transformed thus:

```
nested-entry-level-2 = "K", "E", "Y", GS, GS, US,
  "V", "A", "L", "U", "E", GS, GS, US, GS, GS, RS;
```

If the second *dict* nested in the third *dict* is looked up in the third *dict* then its prepared separator sequences are transformed by the prepare-for-unnesting operation, effectively removing one `GS` character before each `US` and `RS` character, which for the initial *dict*, nested as a value of the second nested *dict*, removes one of the two accumulated `GS` characters before its `US` and `RS` characters, transforming the entry sequence from `nested-entry-level-2` back to `nested-entry-level-1`. If the *dict* nested in the second *dict* copy obtained from the third *dict* is then looked up in the second *dict* copy, its prepared separator sequences will again be transformed by the prepare-for-unnesting operation, removing the remaining `GS` characters before its `US` and `RS` characters, transforming the entry sequence from `nested-entry-level-1` back to `unnested-entry`.

Unforunately, the only way I could see to perform the required replacements for the prepare-for-nesting and prepare-for-unnesting operations was to farm the tasks out to `sed` using two global substitutions per preparation operation. This, of course, is quite a heavy weight afair.

Exemplary dict-ation

Enough of the waffle, let's dive in an see how to use *dict* in practice. The *dict.sh* shell script library and the examples shown are available in the GitHub *sh-utils* repository [ShUtils].

So let us start with the obligatory *Hello World* example (Listing 1, overleaf).

First the source of the *dict* shell script library is included using the dot built-in. It is assumed that *dict.sh* is either on the `PATH` or in the same directory as the *Hello World* script.

Next a *dict* is created using `dict_declare_simple`, as we are not nesting any *dict* values. It can be seen that the parameters to `dict_declare_simple` are values for initial entries to initialise the new *dict* with. They are grouped pairwise: *key1 value1 key2 value2 ... keyN valueN*. This is easy to handle in Shell Command Language thanks to the

```
#!/bin/sh

. dict.sh

record="$(dict_declare_simple 'greeting' 'Hello'\
  'who' 'World')"
echo "$(dict_get_simple "${record}" 'greeting'),\
 $(dict_get_simple  "${record}" 'who')!"

record="$(dict_set_simple "${record}"\
  'greeting' 'Hi' 'who' 'Earth')"
greeting="$(dict_get_simple "${record}"\
  'greeting')"
who="$(dict_get_simple  "${record}" 'who')"
echo "${greeting}, ${who}!"
```

Listing 1

`$@` and **`$#`** special parameters and the **`shift`** built-in. The returned *dict* string is assigned to a variable named **`record`**.

The whole greeting is then written to the console by calling **`dict_get_simple`** to obtain the values associated with the *greeting* and *who* keys. The returned values are expanded directly into the **`echo`**'d string.

The *dict* is then updated by calling **`dict_set_simple`**. Note the pattern of passing the expanded **`record`** variable as the first argument and receiving the returned *dict* string back into the **`record`** variable. If we wanted to keep the original state of **`record`** then the updated *dict* could be assigned to a different variable. Once again note the pairwise grouping of entry keys and values in the parameter list.

Finally, the *greeting* and *who* values are obtained from the updated *dict*, this time assigning each value to its own variable, which are then written to the console.

Assuming the *Hello World* script file is called **`dict_hello_world`**, we are in a console session and the working directory is the directory containing **`dict_hello_world`** then executing:

```
./dict_hello_world
```

should produce the following results (not forgetting to ensure the script is executable):

```
Hello, World!
Hi, Earth!
```

What is needed though is more colour! This can be achieved by using ANSI terminal control sequences [AnsiTermCodes]. In the next example the *Hello World* example is extended to add keys *foreground* and *background* which have nested *dict* values containing keys *r*, *g* and *b* to store 24-bit (8 bit + 8 bit + 8 bit) RGB colour value triples which are used to set the foreground and background colours of the output greeting text using 24-bit colour mode ANSI terminal control escape codes. (See Listing 2.)

After including the *dict.sh* source as before, there is support for setting the ANSI terminal foreground and background 24-bit colours consisting of a bunch of constants (**`readonly`** variables) that build up the ANSI terminal control escape codes and the **`set_ansi_24bit_colours_string`** function that takes *dict*s having *r*, *g* and *b* keys whose values specify the individual 8-bit components of the 24-bit colours for the foreground and background. The **`set_ansi_24bit_colours_string`** result is returned in the **`return_value`** global variable and is a string that will set the requested 24-bit colours when written to a supporting ANSI terminal such Xterm, GNOME terminal or KDE's Konsole (as listed in the **24-bit** section of [AnsiTermCodes]).

As previously, a *dict* is created. However, this time **`dict_declare`** must be used as some of the initial entry values are *dict*s. The nested *dict* values for the *foreground* and *background* entries are created by calling **`dict_declare_simple`** as none of the nested *dict*s' entries' values are themselves *dict*s.

`set_ansi_24bit_colours_string` is called before outputting the complete greeting, and is passed the foreground and background RGB

```
#!/bin/sh
. dict.sh

readonly ASCII_ESC=$(echo '\033')
readonly ANSI_CMD_PREFIX="${ASCII_ESC}["
readonly ANSI_CMD_END="2m"
readonly ANSI_CMD_RESET="${ANSI_CMD_PREFIX}0m"
readonly \
  ANSI_CMD_FG_MULTIBIT="${ANSI_CMD_PREFIX}38"
readonly \
  ANSI_CMD_BG_MULTIBIT="${ANSI_CMD_PREFIX}48"
readonly \
  ANSI_CMD_FG_24BIT="${ANSI_CMD_FG_MULTIBIT};2"
readonly \
  ANSI_CMD_BG_24BIT="${ANSI_CMD_BG_MULTIBIT};2"

set_ansi_24bit_colours_string() {
  local fg="${1}"
  local bg="${2}"
  local r="$(dict_get_simple "${fg}" 'r')"
  local g="$(dict_get_simple "${fg}" 'g')"
  local b="$(dict_get_simple "${fg}" 'b')"
  local fg_cmd="${ANSI_CMD_FG_24BIT};${r};${g};\
${b}${ANSI_CMD_END}"

  r="$(dict_get_simple "${bg}" 'r')"
  g="$(dict_get_simple "${bg}" 'g')"
  b="$(dict_get_simple "${bg}" 'b')"
  return_value="${fg_cmd}${ANSI_CMD_BG_24BIT};\
${r}; ${g};${b}${ANSI_CMD_END}"
}

record="$(dict_declare \
  'greeting' 'Hello' 'who' 'World' \
  'foreground' "$(dict_declare_simple 'r' '127' \
                  'g' '255' 'b' '80' )" \
  'background' "$(dict_declare_simple 'r' '80' \
                  'g' '0'   'b' '0'  )" \
  )"

set_ansi_24bit_colours_string \
  "$(dict_get "${record}" 'foreground')" \
  "$(dict_get "${record}" 'background')"

echo -n "${return_value}"
echo -n "$(dict_get_simple "${record}"\
       'greeting'),\
  $(dict_get_simple  "${record}" 'who')!"
echo "${ANSI_CMD_RESET}"

record="$(dict_set_simple "${record}" \
  'greeting' 'Hi' 'who' 'Earth')"

record="$(dict_set "${record}" \
  'foreground' "$(dict_declare_simple 'r' '255' \
                  'g' '127' 'b' '80' )" \
  'background' "$(dict_declare_simple 'r' '0' \
                  'g' '80' 'b' '0' )" \
  )"

fore="$(dict_get "${record}" 'foreground')"
back="$(dict_get "${record}" 'background')"
set_ansi_24bit_colours_string "${fore}" "${back}"

greeting="$(dict_get_simple "${record}" \
         'greeting')"
who="$(dict_get_simple  "${record}" 'who')"
echo "${return_value}${greeting},\
  ${who}!${ANSI_CMD_RESET}"
```

Listing 2

triple *dict*s which were obtained using **`dict_get`** (as they are nested *dict* values) from the **`record`** *dict* and stored in variables **`fore`** and **`back`**.

The string returned by **`set_ansi_24bit_colours_string`** in **`return_value`** is **`echo`**'d to the console without a terminating newline to set the terminal's foreground and background colours and then the greeting is output as before except no terminating newline is output as following the greeting the **`ANSI_CMD_RESET`** constant string is output to reset the terminal, removing the previously set foreground and background colours.

The *greeting* and *who* entries of **`record`** are updated as before with **`dict_set_simple`**. Then the *foreground* and *background* RGB triple *dict* values are updated using **`dict_set`**. Note that all four entries could have been updated in a single call to **`dict_get`**. However, doing it in two parts demonstrates that a *dict* containing nested *dict* values can still be operated on by the *simple* forms of the API functions so long as no *dict* entry values are involved in that specific function call.

Once more, the (updated) foreground and background RGB triple *dict*s are passed to **`set_ansi_24bit_colours_string`** to obtain the new set-colours ANSI control escape code string. The updated values for the *greeting* and *who* entries are, as before, obtained with calls to **`dict_get_simple`** and stored in variables **`greeting`** and **`who`**.

Finally, the whole output sequence: set-colour-control-escape-codes, **`greeting`**, **`who`**, ANSI reset control-escape-code is output in a single **`echo`** command.

Executing the extended *Hello World* example should produce the same textual output, only with more colour involved.

Other tricks

There are a few things that you can do with *dict*s that might be of interest. First, *dict*s are strings. Yes, I know that has been stated a time or two already. However, as strings they can be used like any other string, although of course just printing them out loses something in the process – the unprintable separator characters for a start.

More usefully, they can be passed to commands and other scripts as arguments. This is useful if, for example, you have a suite of mutli-stage scripts that run one after the other and it is the intial script's job to obtain the arguments and options for the job – they are collected in a *dict* which is passed from one script to the next, each accessing the arguments and options relevant to itself.

Of course as strings they can also be saved to disk and read back later, or sent over a network. Basically, *dict*s are already serialised. The only issue in the future would be with regard to possible differing versions of the *dict* data format, at which time the *version* metadata field in the *dict* header would become a lot more relevant.

As an associative *key:value* container it is possible to use *dict*s to emulate other container types. Two that come to mind are *vectors* (that is single dimension arrays) and *sets*.

The characteristics of a *vector* are that values are identified by an integer index, usually starting at 0 or 1 and increasing by one for each entry. Another characteristic of some implementations is that they can only have elements efficiently added to the end of the vector.

A *dict* can emulate a vector which for example has elements starting at index 0 and only allows appending elements to the end. The idea would be to wrap the underlying *dict* operation function calls in vector specific operation functions.

Operations that create new elements do not require the index key values as parameters. Rather they synthesise the next index as being the value returned by **`dict_size`**. Hence an empty vector-dict would have size zero and so the index key for the first element would be '0', the size is then one thus the index of the next added element would be '1' and so on. Operations that add new elements would be created with initial element values and append new element values, which would be wrappers around **`dict_declare`** (or **`dict_declare_simple`**) and **`dict_set`** (or **`dict_set_simple`**). Listing 3 shows what a **`vector_append`** function might look like.

```
vector_append() {
  local vector="${1}"
  shift
  local count=$#
  local index=$(dict_size "${vector}")
  while [ ${count} -gt 0 ]; do
    local value="${1}"
    shift
    set -- "$@" "${index}" "${value}"
    count=$(( ${count}-1 ))
    index=$(( ${index}+1 ))
  done

  # pass the vector and converted entry values
  # to dict_set:
  vector_return_value="$(dict_set \
                         "${vector}" "$@")"
}
```

Listing 3

Note that the values to be added have to be pairwise interleaved with their index key values which requires modifying the **`$@`** special parameter while iterating over it which is why a snapshot of the value of **`$#`** is taken before the iteration starts which is then manually decremented. Similarly a snapshot of the intitial size of the vector is taken before iteration start which is incremented manually – this is to reduce the overhead of calling **`dict_size`**.

Other operations that could be useful would be accessing a specific element by index key, which would simply devolve to a call to **`dict_get`** (or **`dict_get_simple`**), and updating an existing element, which would basically wrap a call to **`dict_set`** (or **`dict_set_simple`**) with checks on the index value to ensure it is a valid and in range index value.

Iteration can of course be performed via **`dict_for_each`**, possibly via a wrapper that adapts the function called if not all of the usual three arguments **`dict_for_each`** passes to the supplied function for each entry (key, value and computed one based record number) are not required.

Emulating a *set* is similar in that both keys and values do not need to be specified when adding set members. In this case, it is only the *key* values that need to be given, the associated values are of no interest other than them not being empty so that set membership can be determined by passing the cadidate value to **`dict_get_simple`** (or **`dict_get`**) and testing the result to see if it is not empty. The entry values can all be the same – preferably a short value such as **`'_'`**.

As with the vector emulation case, functions can be written that wrap the underlying *dict* operations. For example, Listing 4 shows what a predicate function to check if a set-dict contains a value might look like.

Conclusions and possible future directions

The *dict.sh* Shell Command Language library script implements a dictionary container using *mostly* just standard features of the POSIX Shell

```
set_contains() {
  local set="${1}"    # the haystack to search
  local value="${2}"  # the needle to find
  local maybe_in="$(dict_get_simple \
                 "${set}" "${value}")"
  if [ -n "${maybe_in}" ]; then
    true; return
  else
    false; return
  fi
}
```

Listing 4

Command Language plus simple use of a few core utilities, with the exception of **`local`** and **`echo -n`**.

However, performance is a concern. The call-out to **`sed`** to perform the required substitutions when inserting or retrieving nested *dict*s is particularly heavy on performance.

The use of *Command Substitution* is also an area that drags down performance. In fact internally the private, uglyfied support functions are now all called directly and return their results by setting a **`__dict_return_value__`** global variable.

I have a couple of ideas to potentially address these two issues.

The first is to re-work how *dict*s are nested that removes the need for changing the data format and thus removes the dependence on **`sed`**. The idea – very nebulous at the moment – would be to give each *dict* an identifier, which might be random and may need to be globally unique, and bolt nested *dict*s onto the end of the *dict* string with some sort of new separator sequence – maybe using the currently unused FS character. Entries that have nested *dict* values would have values that reference that *dict* by identifier.

The second idea is to add a parallel set of API functions that are called directly rather than by *Command Substitution* and return their value in a variable specified by name by the caller as an additional parameter by making used of the **`eval`** built-in utility. These would be more cumbersome to use but eleminate having to spin up a sub-shell process for every call.

However, at the end of the day there is only going to be so much that can be done to reduce performance overheads. The underlying sequential nature of strings and the complexity required by the slice and splice operations will end up limiting performance. *dict*s are never going to have any great performance, especially at scale. However they were not intended to.

The reliance on **`echo -n`** for the existing API functions can be fixed by using standard facilties of the **`printf`** utility.

Some additional operations would be useful. One that springs to mind is a **`dict_merge`** or **`dict_extend`** operation that combines the elements of two (or more?) *dict*s.

Rather than emulating other container types, it would be cleaner to implement each as their own library script, tailoring the data structure and operations to suit each.

I do not know if or when I will make these changes as I had already gone down a few rabbit holes while getting on with a larger task when I started implementing *dict*. It would be good to bottom out and pop back up out of some of the rabbit holes. ■

References

[AsciiTable] ASCII Table https://www.asciitable.com/

[AnsiTermCodes] ANSI terminal control codes https://en.wikipedia.org/wiki/ANSI_escape_code

[EBNF] Extended Backus Naur Form https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

[OpenGroup] The Open Group https://www.opengroup.org/

[ShellLang] Shell Command Language https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html

[ShellLangRationale] Rationale for Shell and Utilities, Shell Command Language, Shell Commands, Function Definition Command, p.2 https://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xcu_chap02.html

[ShUtils] sh-utils repository https://github.com/ralph-mcardell/sh-utils

And the winners are...

In *Overload* 166 and *CVu* 33.6, we invited you to vote for your favourite articles of 2021 both in *Overload* and in *CVu*, which is our sister magazine for members. The results are in.

For CVu:

1st place: James Handley for 'The Culture of Code' in *CVu* 33.2 (May 2021)

Roger Orr for 'How Many Braces Must A Programmer Write Down' in *CVu* 33.3 (July 2021)

Simon Sebright for 'Let's Reproduce' in *CVu* 33.5 (November 2021)

Closely followed (with 3 votes each) by:

Roger Orr for 'Buffer Overflows on Windows and How to Find Them' in *CVu* 32.6 (January 2021)

Frances Glassborow for 'Russel Winder 1955/12/30-2021/1/23' in *CVu 33.1* (March 2021)

For Overload:

1st place: Bjarne Stroupstrup for 'C++ – an Invisible Foundation of Everything' in *Overload* 161 (February 2021)

Followed (with 3 votes each) by:

Eugene Hutomy for 'Chepurni Multimethods for Contemporary C++' in *Overload* 162 (April 2021)

Steve Love for 'Amongst Our Weaponry' in *Overload* 162 (April 2021)

Frances Buontempo for 'Teach Your Computer to Program Itself' in *Overload* 164 (August 2021)

Lucian Radu Teodoresco for 'C++ Executors: the Good, the Bad, and Some Examples' in *Overload* 164 (August 2021)

Anders Knatten for 'No Move vs Deleted Move Constructors' in *Overload* 166 (December 2021)

Thank you to everyone who took time to vote, and for those who wrote the articles. We can't offer a prize to these winners, just the mention here. A number of other writers got a vote – so be assured if you wrote for us someone probably thoroughly enjoyed what you had to say.

Keep up the good work.

The article titles above link to the articles if you are reading this online. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones. If you're not a member yet, why not join?

Why Should Automation Be Done By The Dev Team?

Test automation and BDD are related but different. Seb Rose explains why developers need to be involved in the automating of test scenarios for BDD.

I've been writing and talking about test automation and BDD for quite a while now. In February 2021, I gave a short version of a talk called 'Are BDD and test automation the same thing?' at the Automation Guild conference [Rose21a] to explore their relationship and address the confusion that exists in the industry.

The conference organiser, Joe Colantonio, hosted a Q&A session after the talk, but there wasn't enough time to answer all of the questions. Handily, he provided me with a list of all the questions asked, along with his estimation of their 'sentiment' – either neutral or negative. He marked five unanswered questions as having a negative sentiment.

- Why should automation be done by the dev team?
- Isn't the business-readable documentation just extra overhead?
- What's wrong with changing the scenarios to enable automation?
- Can all testing be automated?
- How can Cucumber help us understand the root causes of failure?

This article addresses the first of them.

The question

> I do not get, why u say the automation part #5 on the graph, should be done by the DEV team and never the QA, because it's part of the design process. For example, if a team uses Serenity-BDD with screenplay, which makes it very easy to implement SBEs, shouldn't DEV team focus on prod code instead?

The question relates to the diagram in Figure 1 (opposite), which was published in *Discovery* [Nagy18] and as a LinkedIn article [Rose18].

In my talk, I made it clear that I believed that developers should be involved in step **#5 - Automate**. The questioner asks "*... shouldn't DEV team focus on prod code ...?*" This is a very common question, rooted in a confusion between BDD and Test Automation.

Test automation

Test automation is a generic term that can be applied to any activity that results in the automation of tests. Programmer (or unit) tests are one aspect of test automation. So are integration tests and end-to-end (E2E) tests. Load, performance, and penetration tests can also be automated.

Typically, teams that are focused on test automation do that automation *after* the code has been written. Development and testing are separate activities, often undertaken by different teams with different goals.

- Development goal: implement the feature
- Testing goal: check the implementation achieves expected quality

This approach is widespread and valuable but has some drawbacks. If you'd like to dig into those drawbacks there are some links in the 'Going deeper' section below.

BDD

Behaviour-driven development (BDD) is an approach that grew out of test-driven development (TDD) and agile software development. The goals of BDD [Cucumber] are:

- Encouraging collaboration across roles to build shared understanding of the problem to be solved
- Working in rapid, small iterations to increase feedback and the flow of value
- Producing system documentation that is automatically checked against the system's behaviour

The diagram at the beginning of this article lays out an idealised behaviour-driven process flow. As you can see, step **#5 – Automate** comes before **#6 – Implement**, which can seem back-to-front from some perspectives. How can we automate the testing of software that doesn't exist yet?

Take a minute to change perspective and things don't seem quite so crazy. The Automate step isn't about testing at all. Each scenario describes one behaviour of the system which will need to be implemented in code. When we automate that scenario, we begin to *imagine the code we wish we had*. This is a detailed design activity – and as such needs the involvement of someone with development skills.

This approach is, in this respect, identical to TDD – except that TDD is generally a developer-only activity. BDD is collaborative, bringing together the 3 Amigos (business, developer, tester) to collaborate throughout. The additional benefits that BDD brings are a shared understanding of the business domain and business-readable documentation.

#5 – Automate transforms a scenario in the business-readable documentation into a failing automated test. This guides the development team as they design and implement the code that will deliver the specified functionality. Once the behaviour has been implemented, the automation ensures the continued correctness of the system's behaviour, and the documentation is considered *living* documentation. In contrast, documentation that lives in textual documents that are external to the system itself, can be thought of as *dead* documentation, because it usually reflects how the system *used to behave*.

BDD does not replace traditional testing and test automation, but it does reduce a team's reliance on them to some degree. Nor does BDD replace a team's need for people with testing skills. They are needed more than ever – to help the team reach a shared understanding, to share with developers their specialised domain knowledge, and to ensure customer satisfaction using specific skills such as exploratory, load, or usability testing.

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. He's now Continuous Improvement Lead with SmartBear, helping apply the lessons he has learned to internal development practices and product roadmaps. Co-author of the BDD Books series *Discovery* and *Formulation* (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

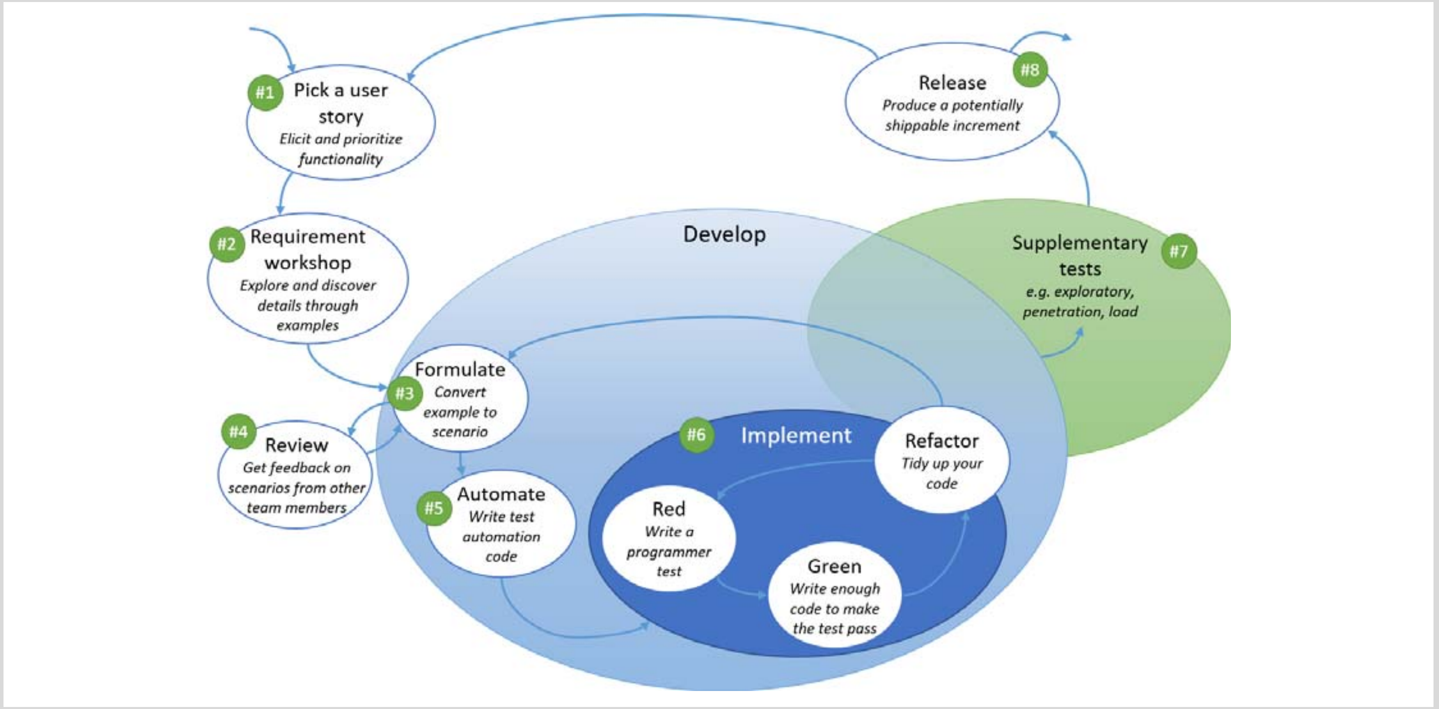

#1 Pick a user story
Elicit and prioritize functionality
#2 Requirement workshop
Explore and discover details through examples
#3 Formulate
Convert example to scenario
#4 Review
Get feedback on scenarios from other team members
#5 Automate
Write test automation code
Develop
#6 Implement
Red
Write a programmer test
Green
Write enough code to make the test pass
Refactor
Tidy up your code
#7 Supplementary tests
e.g. exploratory, penetration, load
#8 Release
Produce a potentially shippable increment

Figure 1

Confusion

The confusion between test automation and BDD may have its roots in its predecessor, TDD. It has always been problematic that a design and development technique should have the word "test" so prominent in its name. Especially in an industry where development and test have been so siloed for so long.

The confusion has been exacerbated by the understandable desire to utilise testers that don't have development skills to write automated tests. This desire has been encouraged by the existence of natural language automation formats using the language of Given/When/Then:

- Given/When/Then are the core keywords of Gherkin
- Gherkin is the structured syntax understood by automation tools such as Cucumber
- Cucumber is a widely downloaded, open-source tool, available on numerous platforms
- Cucumber was created to support BDD

These facts allow people to deduce an 'obvious' but incorrect conclusion:

- I conform to Gherkin when I write my tests
- I use Cucumber to automate my Gherkin
- Cucumber was created to support BDD
- **Therefore, I am 'doing BDD'**

The correct conclusion should be:

- **Therefore, I am automating tests using Given/When/Then**

The correct logic flows in the opposite direction:

- To benefit from a shared understanding, the team needs to collaborate on the detailed specifications
- To assure the value of that understanding, it must be captured using business-readable terms
- Gherkin's use of natural language and Given/When/Then makes it an ideal choice
- Cucumber's ability to understand Gherkin makes it the ideal automation tool
- **Therefore, Cucumber and Gherkin are supporting the team to work in a behaviour-driven way**

Conclusion

If the goal is to automate a test, then you may not need developer skills (I think you still need developer skills, but that is another article).

If the goal is to reduce misunderstandings (and hence defects, rework, and costs), then you should look beyond test-after automation to BDD. In which case functional automation is an integral part of the design and implementation process and requires the development team to be leading participants. ■

Going deeper

I've presented a session called'Contrasting test automation and BDD' on this topic at a number of conferences and webinars over the past year. For more extensive coverage, please watch the video and take a look at the slides [Rose21b].

References

[AG21] The Automation Guild Conference 2021website: https://guildconferences.com/ag-2021/

[Cucumber] 'Behaviour-Driven Development' available at https://cucumber.io/docs/bdd/

[Nagy18] Gaspar Nagy and Seb Rose (2018) *Discovery: Explore behaviour using examples: Volume 1 (BDD Books)*, ISBN 978-1983591259

[Rose18] Seb Rose 'BDD Tasks and Activities', posted on Linkedin 21 December 2018, available at https://www.linkedin.com/pulse/bdd-tasks-activities-seb-rose/

[Rose21a] 'Are BDD and test automation the same thing?', presented at the Automation Guild conference [AG21], available at https://www.slideshare.net/sebrose/are-bdd-and-test-automation-the-same-thing-automation-guild-2021

[Rose21b] Seb Rose 'Contrasting test automation and BDD' (2020-2021) Slides: https://www.slideshare.net/sebrose/test-automation-and-bdd-vivit-unicom-2020 Video: https://tinyurl.com/2p8s7tmc

This article was first published on the Cucumber blog, which is supported by SmartBear (https://cucumber.io/blog/bdd/why-should-automation-be-done-by-the-dev-team/) on 17 March 2021.

C++20 Benefits: Consistency With Ranges

Where do you begin when walking over a container in C++? Andreas Fertig shows how C++20 Ranges simplify this.

This article is a short version of Chapter 3, 'Ranges', from my latest book *Programming with C++20*. The book contains a more detailed explanation and more information about this topic.

You have probably all already heard of C++20's ranges. With ranges-v3, Eric Niebler has already provided us with a solution, independent of C++20 [Niebler]. In this article, I would like to shed some light on how C++20's ranges work and the benefits you get from them. There are multiple benefits from ranges. Today, I want to talk about *consistency*. I assume that you already know about ranges or that you can catch up quickly, so I'm not focussing on the various algorithms ranges bring us, nor the pipe syntax. I want to teach you how ranges help achieve consistency, what this means, and how you can apply it to your own codebase, independently of C++20. Let's get started.

What's consistency in this context?

The first question is, what is consistency? Let's have a look at the example in Listing 1.

Essentially, we can see two types there: **`Container`** ❶ and **`OtherContainer`** ❸. The internals do not matter for this article. What matters is the function begin. We see it in ❷ as a free-function for Container and as a member-function in **`OtherContainer`**.

In **`Use`**, we look at an abbreviated function template from C++20. For those who haven't seen this before, think of it as a function template. The key here is that we don't know the type of parameter **`c`** – a situation we have regularly in generic code. The question now is, what is the correct way to call **`begin`**? I'm showing you two approaches here. ❹ does call a free function begin, relying on overload-resolution. ❺, on the other hand, does explicitly call the **`std`** version of **`begin`**.

The issue is, we don't know which type **`c`** is, and both attempts are good for only one of the containers. This is a usual burden in generic code. The workaround is a so-called two-step **`using`**. We use **`using`** to bring **`std::begin`** into the overload-set. Now, we use an unqualified call to **`begin`**. This picks up the version in **`std`** and the free-function we provided for **`Container`**. In code, it looks like Listing 2.

Arthur O'Dwyer wrote a post about two-step with **`std::swap`**, which explains it from a different angle. [O'Dwyer]

The one issue pre-C++20 is that **`std::begin`** deals only with member-functions, which brings an inconsistency. While we can get the example above working in generic code, we end up with at least three different functions being called:

```
struct Container {}; // ❶ Container without begin
int* begin(Container); // ❷ Free-function begin
                       // for Container

struct OtherContainer { // ❸ Container with begin
  int* begin();
};

void Use(auto& c)
{
  begin(c);       // ❹ Call ::begin(Container)
  std::begin(c);  // ❺ Call STL std::begin
}
```

Listing 1

```
void Use(auto& c)
{
  using std::begin;  // Bring std::begin in the
                     // namespace

  //  Now both functions are in scope
  begin(c);
}
```

Listing 2

- **`begin(Container)`** for **`Container`**
- **`std::begin`** for **`OtherContainer`**
- **`OtherContainer::begin`** also for **`OtherContainer`**

In the case of the member function, when **`std::begin`** can be used, it calls the member function for us. The inconsistency is that not *all* calls are routed via **`std::begin`**. What if **`std::begin`** does a couple of checks on the type and puts some safety measures on if these checks fail? Then we do get these benefits for **`OtherContainer`** but not **`Container`**. This is not only sad. It can be a nightmare to debug.

Ranges for consistency

Of course, we wouldn't talk about ranges if they could not solve this situation. Here is what you do when ranges are available:

```
void Use(auto& c)
{
  // Use ranges
  std::ranges::begin(c);
}
```

`ranges::begin` looks for free- and member-functions. This makes it so much better. But why doesn't **`std::begin`** do the same? Well, because of ADL (argument dependent lookup). Once we've provided our own free function, **`begin`**, for a type, it beats **`std::begin`**. Why? Because this is how ADL works (I'm not going into the details here, it could fill at least another article.)

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

Just use ranges in this case, and you don't need to learn the two-step **`using`** and about ADL. At this point, you can stop reading. You have already learned how you could improve your code with ranges. But you would like to learn more? Good. Why should only ranges do this magic?

Consistency for your code-base

Okay, we do want to get the same result as with ranges. We want to have a function, let's say **`begin`**, which users can customize, but all calls should first go to our **`begin`** function.

We use the data types from before. The goal is to provide our own **`begin`** function in the namespace **`custom`**, giving us the same consistent behaviour as ranges do.

```
void Use(auto& c)
{
  custom::begin(c);
}
```

The code above is what we need to use. Now let's see how we build **`custom::begin`**.

A function object to avoid ADL

The first step is to avoid ADL. It is great, but in our case effectively prevents us from having a **`custom::begin`** call regardless of existing free functions. How can we do this? Well, we avoid the function call. Paraphrased from a famous space movie, 'These are not the functions you're looking for.' Instead of the function **`begin`**, we provide a callable with the name **`begin`** (see Listing 3).

In ❶, we see our callable **`begin`**. It is a plain **`struct`** with a templated call operator. Inside this call-operator, in ❷, we use **`constexpr if`** from C++17 together with C++20's Concepts ("I love it when a plan comes together" comes to mind) first to check whether the type **`Rng`** provides a free-function **`begin`**. If so, we call it by moving the data to it. Otherwise, the **`else if`** checks with the same utilities whether **`Rng`** has a member-function **`begin`**. The procedure is the same. If found, the member function is called, and the parameter is moved into it.

```
namespace custom {
  namespace details {
    struct begin_fn {  // ❶ Callable
      template<class R>
      constexpr auto operator()(R&& rng) const
      {
        // ❷ Free-function
        if constexpr(requires(R rng) {
          begin(std::forward<R>(rng)); }) {
            return begin(std::forward<R>(rng));

          // Same as above for containers
        } else if constexpr(requires(R rng) {
          std::forward<R>(rng).begin();
          }) {
          return std::forward<R>(rng).begin();
        }
      }
    };
  }  // namespace details

  // Callable variable named begin
  inline constexpr details::begin_fn begin{};
}  // namespace custom
```

Listing 3

```
namespace custom {
  namespace details {
    constexpr auto begin_fn = []<class R>(R&& rng)
{   // Callable
      // Free-function
      if constexpr(requires(R rng) {
        begin(std::forward<R>(rng)); }) {
          return begin(std::forward<R>(rng));

        // Same as above for containers
      } else if constexpr(requires(R rng) {
        std::forward<R>(rng).begin();
        }) {
        return std::forward<R>(rng).begin();
      }
    };
  }  // namespace details

  // Callable variable named begin
  inline constexpr auto begin
    = details::begin_fn;

}  // namespace custom
```

Listing 4

Congrats! With this simple change, I hope you agree that it is simple or at least manageable, your code is now more consistent. As long as we call **`custom::begin`**, this function is called first and routes the call to the free or member-function. But there is more.

Chipping in a bit more C++20?

Since we have already used abbreviated function templates and Concepts from C++20, why not see what other features from the future are here now that we can apply?

The callable seems a bit much to write. Plus, you all probably know by now that a lambda is a callable as well. In fact, what I presented above could as well be a lambda. The only thing pre-C++20 was that there was no nice way to have a template type-parameter. Yes, C++14's generic lambdas together with **`decltype`** allowed us to do this already, but isn't the version below cleaner? (See Listing 4.)

This code here does the same as before. Just that here we use C++20's lambdas with a template-head, allowing us to specify the template type parameter **`R`**. The body of the lambda is a copy of the callable's body. ■

References

[O'Dwyer] Arthur O'Dwyer, 'What is the **`std::swap`** two-step?', available at https://quuxplusone.github.io/blog/2020/07/11/the-std-swap-two-step/

[Niebler] range-v3, available on GitHub: https://github.com/ericniebler/range-v3

This article was first published on Andreas Fertig's blog (https://andreasfertig.blog/2021/05/cpp20-benefits-consistency-with-ranges/) on 4 May 2021.

Afterwood

Humans are fallible and frequently confused by seeming paradoxes. Chris Oldwood reminds us to question our assumptions and try to think straight.

When I was a boy, I got an illustrated encyclopaedia as a present one Christmas or birthday. I wasn't into reading fiction and loved flicking through the book looking at the various cut-away drawings of machines – both old and new – as I was fascinated by how things worked. The one section of the book I remember most clearly though was a picture of a man in a running race with a tortoise. The text described how it was impossible for the man to ever beat the tortoise, if the tortoise had a head start, because when the man reaches the point where the tortoise stood, the tortoise will have moved on. While the book probably referred to the puzzle by its more well-known name of *Achilles and the Tortoise*, I never managed to remember that. It probably even mentioned this is just one of Zeno's many paradoxes, as I discovered some decades later via Wikipedia.

Being a child, I wasn't capable of unravelling the paradox and pointing out where Zeno 'had cheated' but I knew he must have because empirically I saw the outcome disproved every day on the school field at playtime. And yet the logic in the simple statement made perfect sense. Despite reading the various rebuttals numerous times over the years, I'm still not entirely sure I fully understand where the falsehood lies in theory, but I do know that it does in practice.

Over the last couple of years, the pandemic has made me aware of some other interesting statistical anomalies as a variety of people have tried to make sense of the ever growing body of data around the effects of the virus so that models can be built, hypotheses formulated, and (ideally) policy implemented to minimise its effects. One anomaly that seemed to crop up repeatedly in the early days of the pandemic was Simpson's Paradox. In an explanation of its effects, the particular example that most struck a chord was a plot showing that 'in general' more exercise caused an increase in cholesterol, which clearly goes against medical advice. For those of you like me not well versed in statistics, the illusion here is that what you're seeing is just the natural rise in cholesterol as we get older. If you break the plot down into clusters based on age, you see that *within* an age group cholesterol does indeed go down with exercise. The devil, as they say, is in the details.

Another popular mathematical puzzle which crops up regularly is the Birthday Paradox, which has serious applications in cryptography, not just as a parlour trick. The problem is often stated as: "How many people need to be in a room for there to be a better than 50% chance that any two share a birthday?" The answer is 23, which is surprising to a lot of people; hence it's commonly described as a paradox because the answer is counterintuitive, even though with the right level of maths ability you can 'easily' prove it. Somewhat ironically, according to the Wikipedia page, the reason it's discoverer Harold Davenport didn't publish his findings was because he couldn't believe it hadn't already been published.

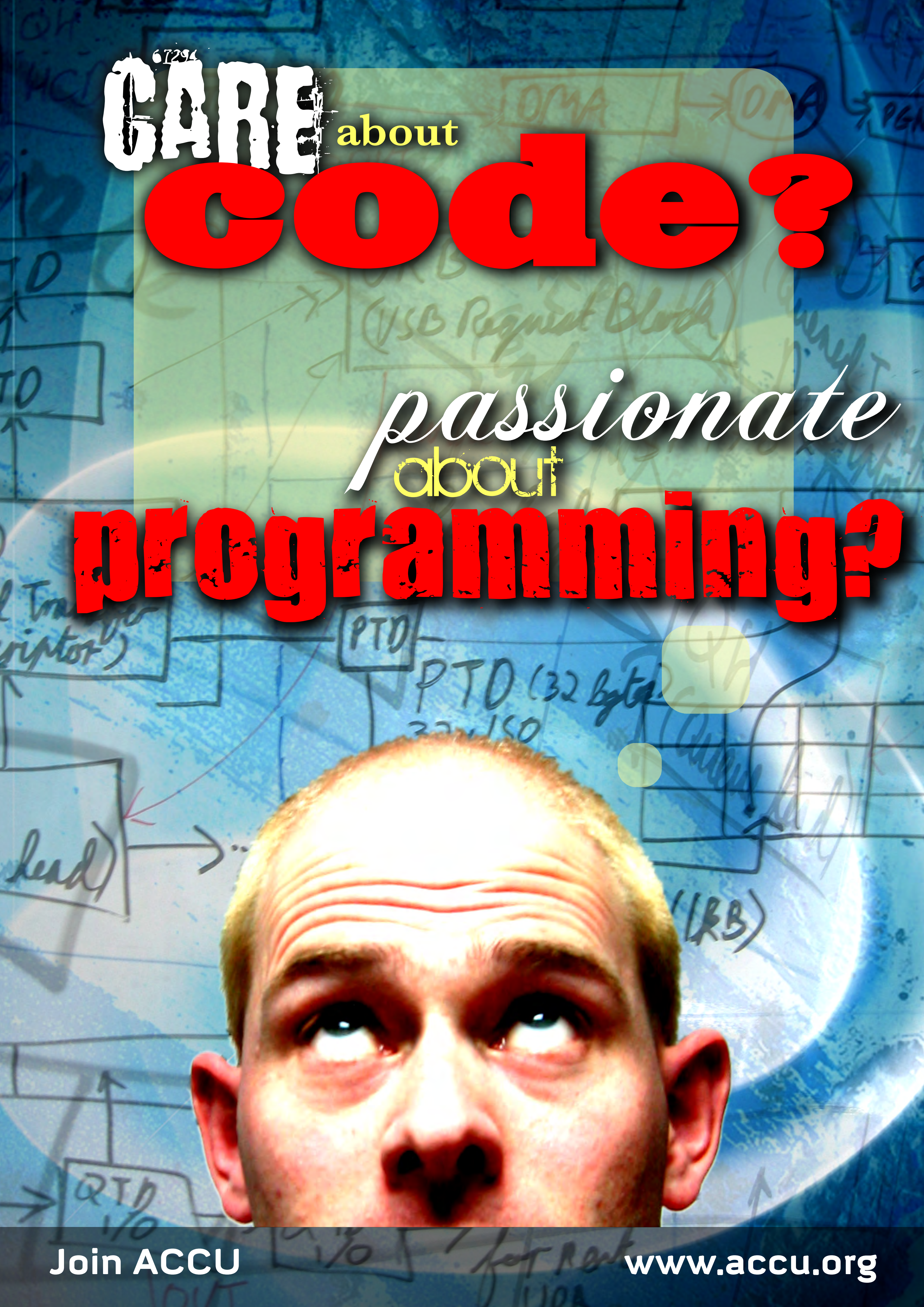
I think we programmers tend to think of ourselves as a pretty logical bunch. The act of programming requires us to be incredibly precise in our instructions to the computer, a device which makes most pedants appear quite liberal. Our job involves reasoning about problems and, where necessary, turning this into code which will ultimately be consumed by a gazillion logic gates. It's logic all the way down. And yet my take-away from Achilles vs Tortoise is that what I think might be logical may in fact be flawed because I can't obviously see a counter-argument. In our eagerness to get something working or a bug fixed we can fall into the trap of formulating a hypothesis that can be proved correct when we should be finding ways to *disprove* it, or in the very least putting the scaffolding in place to make that process easier. In some respects, the practice of TDD borrows from the ideas of falsifiability as it puts a focus on the testability angle of our code (hypothesis). By making the code testable, we make it easier for ourselves to try and conjure up reasons why the code might not work correctly, and ultimately express those scenarios too, in the form of executable code. Much as I'd like to believe I can follow Sir Tony Hoare's advice and "make the program so simple, there are obviously no errors" I'm also aware of the assumptions that can eventually lead to our undoing.

Larry Wall famously described the three great virtues of a programmer as laziness, impatience, and hubris. I've always found the last a curious choice as hubris is commonly used as a pejorative, although I understand it can be seen as a positive force in some circumstances. I wonder how many others misinterpret that quote leading to a lack of humility? In contrast I find fallibility is a more useful position to take.

I once got involved in a support query involving corruption of a cache file. The programmer who wrote the file handling code reached a conclusion that there must be a bug in the Windows `CopyFile` API rather than doubt his own code. While I would never rule that out entirely, there are many reasons why I think it's improbable and I'd be looking far closer to home before even contemplating such an idea. A quick trawl through the parent process's logs and I pointed out that the child process was now being terminated for taking too long to start-up, ultimately caused by an increase in the cache file's size and I/O load on the remote server. This felt more plausible and, more importantly, easier to validate.

Of all the paradoxes in Wikipedia's long list, the one I'm finding easiest to relate to as I get older and more experienced is Socrates' apocryphal saying "I know that I know nothing". Clearly I do know something, lots in fact, but the pace with which our industry moves it can feel like technology is the tortoise and I'm Achilles desperately trying to keep up. ■

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


CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org

accu 2022

Spring ACCU 2022.4.6-9

Pre-Conference Tutorials 2022.4.4-5

REGISTRATION NOW OPEN! **Visit https://conference.accu.org**

HYBRID EVENT

Follow @ACCUConf

Tweet #ACCUConf