overload 148

DECEMBER 2018 £4.50

Diseconomies of Scale

Bigger is not always better. We explore examples that demonstrate when smaller is actually more productive.

Algol 68 - A Retrospective

The many ways Algol 68 has influenced modern programming languages

Measuring Throughput and the Impact of Cache-Line Awareness

Appropriate metrics to measure throughput

Flip Model: A Design Pattern

A pattern to publish dynamic, complex data to many clients in a threadsafe manner

Compile-time Data Structures in C++17: Part 3, Map of Values

Testing code using compile-time maps of values

Memory Management Patterns in Business-Level Programs

We investigate memory management at the application level

A magazine of ACCU

ISSN: 1354-3172

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

OVERLOAD 148

December 2018

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 149 should be submitted by 1st January 2019 and those for Overload 150 by 1st March 2019.

Overload is a publication of the ACCU

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

4 Diseconomies of Scale
Allan Kelly considers why bigger isn't always better.

6 Flip Model: A Design Pattern
Daniele Pallastrelli presents the Flip Model to publish dynamic, complex data to many clients in a threadsafe manner.

10 Memory Management Patterns in Business-Level Programs
Sergey Ignatchenko considers memory management from an application level.

14 Compile-time Data Structures in C++17: Part 3, Map of Values
Bronek Kozicki shows a compile time map of values allows code to be tested more easily.

20 Algol 68 – A Retrospective
Daniel James reminds us just influential Algol 68 has been.

27 Measuring Throughput and the Impact of Cache-Line Awareness
Richard Reich and Wesley Maness investigate suitable metrics to measure throughput.

32 Afterwood
Chris Oldwood reminds us to make sympathetic changes.

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.

Revolution, Restoration and Revival

Trends cycle in seasons. Frances Buontempo wonders what programmers should on the lookout for.

I had a splendid week away in Yorkshire catching up with some friends recently but all the fresh air gave me a stinking cold, so I spent time trying to revive instead of writing an editorial. Sorry, again. Prior to my week away, I attended the Software Craftsmanship Conference in London [SC18a]. Most of the talks are on their YouTube channel [SC18b] so you can watch at your leisure. The closing talk from Michael Feathers encouraged us to be creative and not be unhappy at work. He talked through a few company models he'd seen, focusing on new-ish start-ups, aiming to get stuff done that was interesting and covering costs, rather than aiming to be the next big thing and make a fortune. Michael was emphasising that programmers have an amazing skill set and can do all kinds of creative and useful things, if they allow themselves time to imagine.

Recently, we've seen a rise of hipster bars, artisan coffee shops and start-ups or cottage industries, so the possibility of trying new small-scale enterprises extends beyond programming. In many cases, these new companies are reviving dying old towns and cities. With larger companies closing, and high streets tending to 'shut', the so-called gig economy has driven many people to find new ways of working. People are still willing to pay for food and drink, creating an obvious marketplace for home-made food or small batches of niche beer. Though jokes abound about food and drink served in various strange ways, such as chips in mini frying baskets, beer in a boot, or scrambled egg with a comb from a shoe [BlackBooks00], while countless quips are made about avocados, this direction seems here to stay for a while. While in Yorkshire, we dropped off for afternoon tea in Hebden Bridge, a small market town that was invaded by hippies trying to escape the rat-run in the 1960s. In many ways, the range of shops was similar to hipster areas of London. The similarity between the old hippy scene and the new hipster scene was striking. However, don't forget the hiring pro-tip: "Hippies are not the same as hipsters. I made this mistake once and now have a frontend framework written in Fortran" [@PHP_CEO14].

Trends often go in cycles; any current fashion is frequently a reworking of something that has gone before. Sometimes the economy drives changes, as alluded to by the move to the cottage industry and small start-ups in many places. We see trends and fashions in programming too. Kevlin Henney talked about Algol 86 at SCL. He's given variants of this talk before, including once at last year's ACCU conference. By coincidence, this issue has an Algol68 retrospective. Furthermore, Russel Winder recently commented on accu-general on old trends and techniques coming to the fore recently. For example, "Message passing over channels (or queues) between threads has to be The One True Way. Lots of people have been banging on about this for 40+ years, but it has taken 40 years for all the shared memory multi-threading people to admit defeat and come to their senses!" He also noted that despite different idioms between programming languages, "most programming languages are rapidly converging on a quasi-object, quasi-functional, with generics, coroutines, threadpools and fibres model. Which, of course, indicates we are due a new programming language revolution." Many revolutions, particularly in programming, are re-discoveries or re-vamps of older ideas.

Despite the apparent convergence, different languages do have different approaches. We all spot C-style code in places it doesn't belong. I still catch myself writing a `for` loop instead of a list comprehension in Python once in a while. Old habits die hard. Shoe-horning techniques into the wrong place is a recurring theme, with deep parallels to the craftsmanship movement. If you have a hammer, as the saying goes, everything looks like a nail. A true craftsperson will pick the right tools for the job. If they are restoring an old finely made cabinet, they need to know the proprieties of the wood, varnish, nails, and so on, as well as the right tools to use. I've noticed a few television programmes recently about antiques and furniture restoration, perhaps as a result of some afternoon telly while recuperating. An inappropriate restoration can devalue or even destroy an antique. It takes skill to be able to fix something. Just slapping some sticky-backed plastic or gaffer tape might hold parts in place, but spoils the look and feel, and possibly functionality of the piece.

Restoring an existing code base has similar problems. If someone tries to 'fix' code in an unfamiliar codebase, they might make things worse. A true craftsperson might be able to spot where something like an enterprise code base doesn't follow natural language idioms, and work with the 'grain' of that particular code base more easily than someone with less experience. I could wax lyrical about the parallels between blacksmiths, saddle makers, cabinet makers and metal workers, and writing and maintaining code, but I'll rein myself in. The analogies do work and in both cases we are talking about a skill, which takes time to learn, and is best learnt from a master. Books and online courses might get you started. A YouTube video may show you how to fix an oven element or replace a roof tile, but that's many miles away from being a qualified electrician or being able to re-thatch a cottage. There is also an expectation that masters will be able to train apprentices. Can you mentor or teach someone? Another point that emerged at SCL was you know you understand something properly when you can explain it to someone else. That's why some many of us give talks, or write articles. If you haven't tried this yet, please do. The ACCU local groups can help and support you and the editorial teams for both *Overload* and *CVu* can nurture and encourage you if you want a helping hand.

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.

A craftsperson uses their hands. Always. Programmers use their hands too. We type, sometimes we use a mouse. We draw diagrams, point (and sometimes laugh) at the code, we press buttons. Many other skills involve senses beyond just tactility. Metal and wood workers can hear the difference between something working well or being about to break. Attempts have been made to interpret and project code musically. We do talk about code smells as well. Sometimes code feels right, other times it seems more like the sticky-backed plastic hack. Can you explain what kicks off either feeling? Have a think for a moment. This kind of craft or skill is challenging to vocalise. I suspect it's similar to trying to program a grammar checker. I mention that because I can see green wiggly lines as I type, where my word processing software is 'trying to help' and failing to parse a few of my sentences correctly. Trying to be exact about language is difficult. Trying to be exact about anything can be difficult. And yet, you know when you've managed to communicate with someone. You know when your mentee has 'got it'; they then mentor someone else and you sit back and know your work here is done. Last time I mentioned trying to assess the progress of a mentee [Buontempo18]. I couldn't find any precise metrics, and now suspect that's the way it goes with crafts.

As a final note on the craftsperson metaphor, consider the process of making or restoring a thing, either code or something more physical. The BBC's recent 'Made in Britain' observed a skilled artisan or craftsperson has a hand in the product from start to end. Some companies talk about the software development lifecycle, expecting more senior devs to be able to step beyond the differences between a `for` or `while` loop and talk about how to build and release a product. A really skilled developer can also revive and restore a product once it has been realised. This might include bug fixing, as well as adding new features. I suspect there's a similarity between new features in code and up-cycling old furniture. I'll leave the reader to work out a quip about the trendy 'distressed' look, wherein an old cupboard or chair is sandpapered to look even more beaten up, then whitewashed and sold for a fortune, and a code based being distressed or even distraught, or bugged, if you will.

In addition to furniture/code, restoration, revival and repurposing, we briefly touched on seasonal fashions reoccurring, meaning many revolutions are just that. What goes around comes around. Coroutines, functional programming, Algol68 or similar, roll back into view regularly. Ideas revolve. I have observed some fashions tend to pop up every other generation. Parents hate tattoos, so the rebellious teenagers start to love them, only to discover Grandma was a painted lady displaying her fine covering of art work at fairs and fashion shows years ago. I wonder if programming paradigms and languages do likewise. The youth don't know the new trendy stuff happened before and yet Grandad can tell them all about it. Perhaps we could spot some patterns and make predictions about the future. For sure, the next revolution will not be televised, but stream live on some app that doesn't exist yet, no doubt. When *Overload* had proper editorials, Rik Parkin suggested 'The Computing Revolution Will Be Televised (Again)' [Parkin12], noting that 30 years ago we had to plug our first computers into the TV. Raspberry Pi anyone? He ends by saying, "Nostalgia has never been more cutting edge."

By looking into the history of a discipline, we can find trends and try to see where things are heading. I tried to find a definitive list of breakthroughs in computing, though this strays into the realms of opinions all too easily. I did find a time line [Aaronson14] Scott Aaronson, a quantum computing expert, compiled for MIT's 150th anniversary. He lists when the top 150 innovations happened (I know that's more than 150 – I checked).

Pre-1600s	1600s	1700s	1800s	1900s	1910s	1920s	1930s	1940s	1950s	1960s	1970s	1980s	1990s	2000s
2	3	2	7	1	0	2	4	16	20	29	23	15	19	9

Things appear to have tailed off, with a slight uptick when the internet hit the streets. So much of the history entwines with the history of mathematics, and has occasional outburst of skilled craftspeople or engineers managing to build a machine capable of increasingly precise and complicated movements. Games and viruses also get a mention. I don't think any patterns were obvious, so don't have any predictions of what to look out for next year. As I read the blog, I realised I am not up to date with cutting edge mathematical research, though I do keep my eyes open for trends in machine learning. An area I know little about is quantum computing. I did learn one thing from the blog headline: "Quantum computers would not solve hard search problems instantaneously by simply trying all the possible solutions at once." If anyone would care to write a slightly longer summary for *Overload*, you know what to do.

Now, Scott mentions viruses. In my attempt to track down old trends that are being revived, I did a random walk through the C2 wiki and fell across the 'Worse Is better' page [C2]. Part way down, C is described as a virus. This caught my attention. This comes from a paper by Richard Gabriel, where he describes early Unix and C as examples of the 'Worse is better' school of design [Gabriel94]. This principle prioritises simplicity over correctness. He says it means,

> that implementation simplicity has highest priority, which means Unix and C are easy to port on such machines. Therefore, one expects that if the 50% functionality Unix and C support is satisfactory, they will start to appear everywhere. And they have, haven't they? **Unix and C are the ultimate computer viruses.** [my emphasis]

He notes that the virus must be "basically good" in order to spread or gain traction. Once the programming language or approach is prevalent, people will then spend time ironing out some of the flaws. He finally concludes C is the wrong language for AI software. It appears that Python is gaining traction here, though I suspect Richard was suggesting Lisp is the right tool for the job.

If we use a compiled language, like C, we make things (or cmake them or use scons). Even if we use an interpreted language, we are still being creative. I have no idea where this will go next, but the journey is interesting. We are creative and have lots to offer. If you don't feel like part of the next revolution, take a restorative, and allow yourself time to revive. Cheers and Happy Christmas (assuming you are reading this just after it hits the decks). Or failing that, Happy New year. *Viva la revolution!*

References

[@PHP_CEO14] https://twitter.com/php_ceo/status/475056653285736448?lang=en

[Aaronson14] Scott Aaronson, 'Timeline of computer science' (updated 2014) at https://www.scottaaronson.com/blog/?p=524

[BlackBooks00] Episode 3 of series 1 of Grapes of Wrath, broadcast in the UK on Channel 4 from 2000–2004: https://en.wikiquote.org/wiki/Black_Books#The_Grapes_of_Wrath_(1.3)

[Buontempo18] Frances Buontempo, 2018, 'Are we nearly there yet?' in *Overload* 147, October 2018

[C2] 'Worse is better' at http://wiki.c2.com/?WorseIsBetter

[Gabriel94] 'Lisp: Good News, Bad News, How to Win Big' Richard P Gabriel, 1994: from https://www.dreamsongs.com/Files/LispGoodNewsBadNews.pdf

[Parkin12] Ric Parkin (2012) 'The Computing Revolution Will Be Televised (Again)' in *Overload* 108, April 2012 https://accu.org/index.php/journals/1933

[SC18a] Software Craftsmanship London http://sc-london.com/

[SC18b] Software Craftsmanship conference: https://www.youtube.com/playlist?list=PLGS1QE37I5lSWm0rmE7UkgEmZq6Spg9z7

Diseconomies of Scale

Bigger is not always better. Allan Kelly considers when smaller is more productive.

Without really thinking about it, you are not only familiar with the idea of economies of scale: you expect economies of scale. Much of our market economy operates on the assumption that when you buy or spend more you get more per unit of spending. The assumption of economies of scale is not confined to free-market economies: the same assumption underlies much Communist-era planning.

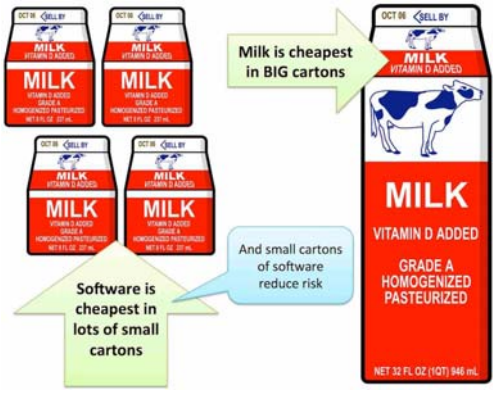
At some stage in our education – even if you never studied economics or operational research – you will have assimilated the idea that if Henry Ford builds a million identical black cars and sells a million cars, then each car will cost less than if Henry Ford manufactures one car, sells one car, builds another very similar car, sells that car, and continues in the same way another 999,998 times.

The net result is that Henry Ford produces cars more cheaply and sells more cars more cheaply so buyers benefit. This is *economies of scale*.

The idea and history of mass production and economies of scale are intertwined. I'm not discussing mass production here, I'm talking *economies of scale*, *diseconomies of scale* and software development.

Milk is cheaper in large cartons

That economies of scale exist is common sense: every day one experiences situations in which buying more of something is cheaper per unit than buying less. For example, you expect that in your local supermarket buying one large carton of milk – say four pints – will be cheaper than buying four one-pint cartons.



Allan Kelly helps companies large and small enhance their agile processes and boost their digital products. Past clients include: Virgin Atlantic, Qualcomm, The Bank of England, Reed Elsevier and many small innovative companies you've never heard of. He invented Value Poker, Time-Value Profiles and Retrospective Dialogue Sheets. A popular keynote speaker he is the author of *Dear Customer, the truth about IT* and books including *Project Myopia*, *Continuous Digital*, *Xanpan* and *Business Patterns for Software Developers*. His blog is at https://www.allankellyassociates.co.uk/blog/ and on twitter he is @allankellynet.

So ingrained is this idea that it is newsworthy when shops charge more per unit for larger packs complaints are made. In April 2015, *The Guardian* newspaper in London ran this story headlined 'UK supermarkets dupe shoppers out of hundreds of millions' about multi-buys which were more expensive per item than buying the items individually.

Economies of scale are often cited as the reason for corporate mergers. Buying more allows buyers to extract price concessions from suppliers. Manufacturing more allows the cost per unit to be reduced, and such savings can be passed on to buyers if they buy more. Purchasing departments expect economies of scale.

I am not for one minute arguing that economies of scale do not exist: in some industries economies of scale are very real. Milk production and retail are examples. It is reasonable to assume such economies exist in most mass-manufacturing domains, and they are clearly present in marketing and branding.

But – and this is a big 'but'...

Software development does not have economies of scale

In all sorts of ways, software development has diseconomies of scale. If software development was sold by the pint, then a four-pint carton of software would not just cost four times the price of a one-pint carton, it would cost *far more*.

Once software is built, there are massive economies of scale in reselling (and reusing) the same software and services built on it. Producing the first piece of software has massive marginal costs; producing the second identical copy has a cost so close to zero it is unmeasurable – Ctrl-C, Ctrl-V.

Diseconomies abound in the world of software development. Once development is complete, once the marginal costs of one copy are paid, then economies of scale dominate, because marginal cost is as close to zero as to make no difference.

Diseconomies

Software development diseconomies of scale have been observed for some years. Cost estimation models like COCOMO actually include an adjustment for diseconomies of scale. But the implications of diseconomies are rarely factored into management thinking – rather, economies-of-scale thinking prevails.

Small development teams frequently outperform large teams: five people working as a tight team will be far more productive per person than a team of 50, or even 15. Academic studies have come to similar findings.

The more lines of code a piece of software has, the more difficult it is to add an enhancement or fix a bug. Putting a fix into a system with a million lines of code can easily be more than ten times harder than fixing a system with 100,000 lines.

Experience of Kanban style *work in progress* limits shows that doing less at any one time gets more done overall.

Studies show that projects that set out to be *big* have far higher costs and lower productivity per deliverable unit than small systems.

Producing the first piece of software has massive marginal costs; producing the second identical copy has a cost so close to zero it is unmeasurable

Testing is another area where diseconomies of scale play out. Testing a piece of software with two changes requires more tests, time and money than the sum of testing each change in isolation.

When two changes are tested together the combination of both changes needs to be tested as well. As more changes are added and more tests are needed, there is a combinatorial explosion in the number of test cases required, and thus a greater than proportional change in the time and money needed to undertake the tests. But testing departments regularly lump multiple changes together for testing in an effort to exploit economies of scale. In attempting to exploit non-existent economies of scale, testing departments increase costs, risks and time needed.

If a test should find a bug that needs to be fixed, finding the offending code in a system that has fewer changes is far easier than finding and fixing a bug when there are more changes to be considered.

Working on larger endeavors means waiting longer – and probably writing more code – before you ask for feedback or user validation when compared to smaller endeavors. As a result there is more that could be 'wrong', more that users don't like, more spent, more that needs changing and more to complicate the task of applying fixes.

Think diseconomies, think small

First of all you need to rewire your brain: almost everyone in the advanced world has been brought up with economies of scale since school. You need to start thinking *diseconomies of scale*.

Second, whenever faced with a problem where you feel the urge to 'go bigger', run in the opposite direction: go smaller.

Third, take each and every opportunity to go small.

Fourth, get good at working 'in the small': optimize your processes, tools and approaches to do lots of small things rather than a few big things.

Fifth – and this is the killer: know that most people don't get this at all. In fact, it's worse, they assume bigger is better. ■

This is an excerpt from Allan Kelly's latest book, *Continuous Digital*, which is now available on Amazon and in good bookshops.

Flip Model: A Design Pattern

Publishing dynamic, complex data to many clients in a threadsafe manner is challenging. Daniele Pallastrelli presents the Flip model pattern to overcome the challenges.

In this article, I describe a design solution that I have adopted several times in the past while working on applications for the diagnosis of complex distributed systems. This solution worked well in several contexts, and it's still proving robust in many running systems.

Although I know for sure it's used by other developers, after some research I could not find any reference to it in the literature, and this finally convinced me to write about it.

After some thought, I decided to document it under the well-known form of a *Design Pattern* as I believe that it's still a convenient way to discuss *software design* and *architectural design* (that – from my point of view – remain fundamental topics in Software Engineering and should not be neglected in favor of more mundane topics).

Furthermore, some young developers might not know the book *Design Patterns* [Gof 1995] that made history, so I hope this article might fill a gap and makes them curious about patterns and software design in general.

In the remainder of the article, I present the pattern following the classic documentation format proposed in the original book (see the 'documentation' section in the Wikipedia article [Wikipedia] or – even better – read the original book).

Pattern name and classification

Flip Model (behavioral).

Intent

The pattern allows multiple clients to read a complex data model that is continuously updated by a unique producer, in a thread-safe fashion.

Also known as

Model publisher, Pressman, Newsagent.

Motivation (forces)

Sometimes it's necessary to decouple the usage of a complex data structure from its source, in such a way that every actor can run at their own pace without interfering with each other.

Consider, for example, an application that periodically retrieves information from a large sensor network to perform some kind of statistical elaboration on the collected data set and send alarms when some criteria are met. The data collected from the sensor network is structured in a complex lattice of objects resembling the ones you would find in the physical world so that the elaboration modules can navigate the data in a more natural way. The retrieval operation is a long, complex task, involving several network protocols, that is completely uncorrelated from the statistical analysis and alarms evaluation, and can possibly run in separated threads. Moreover, data retrieval and its usage have different timing (e.g., the sensor network is scanned every 5 minutes, while the statistical elaboration is performed on request by a human operator on the most recent collected dataset).

In this scenario, how can all the modules of the application work together on the same data structure? How can all the clients use the most updated data available in a consistent fashion? And how can the application get rid of the old data when it is no longer needed?

The main idea of this pattern is to pass the sensor data structure from the producer to the consumers by means of two shared pointers (in C++) or two variables (in languages with garbage collection): one (named `filling`) holding the object structure currently retrieving the sensor data, the other (named `current`) holding the most recent complete acquisition.

A class `SensorNetwork` decides when it's time to start a new acquisition and replaces `current` with `filling` when the acquisition is concluded. When a client needs to perform some tasks on the data acquired, it contacts `SensorNetwork`, which returns `current` (i.e., the most recent data acquired). An object of class `SensorAcquisition` is kept alive and unchanged during the whole time a client holds the smart pointer (and the same is still valid in garbage collected languages).

The data acquisition (performed by `SensorAcquisition`) and its reading (performed by the various clients: `Statistics`, `ThresholdMonitor` and `WebService`) are possibly executed in multiple threads. The safety of the code is ensured by the following observations:

- a `SensorAcquisition` object can be modified only by the thread of `SensorNetwork`, and never changed after it becomes public (i.e., the smart-pointer `current` is substituted by `filling`)
- the smart pointer exchange is protected by a mutex.

It is worth noting that here the mutex is required because `std::shared_ptr` provides a synchronization mechanism that protects its control-block but not the `shared_ptr` instance. Thus, when multiple threads access the same `shared_ptr` and any of those accesses uses a non-const member function, you need to provide explicit synchronization. Unfortunately, our code falls exactly under that case since the method `SensorNetwork::ScanCompleted` assigns the `shared_ptr` to a new value.

However, if the presence of a mutex makes you feel back in the eighties, please see the 'Implementation' section for some modern alternatives.

Figure 1 (overleaf) shows a typical Flip Model class structure.

Applicability

Use Flip Model when:

- You have a complex data structure slow to update.

Daniele Pallastrelli has been programming and designing software for the last 20+ years and he' s passionate about it. A professional software engineer, speaker, author, and runner, he is reluctant to discuss himself in the third person but can be persuaded to do so from time to time. In his spare time, Daniele writes papers and blog posts, which, considering where you're reading this, makes perfect sense. He can be contacted via twitter at @DPallastrelli

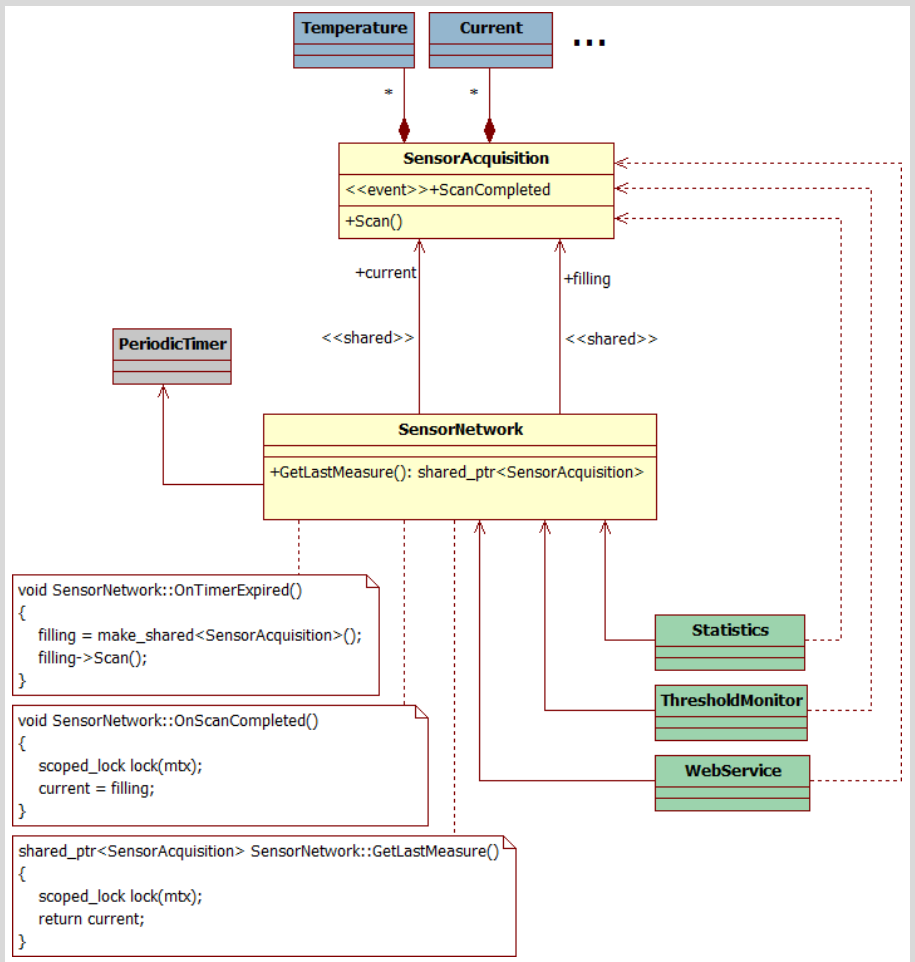


Figure 1

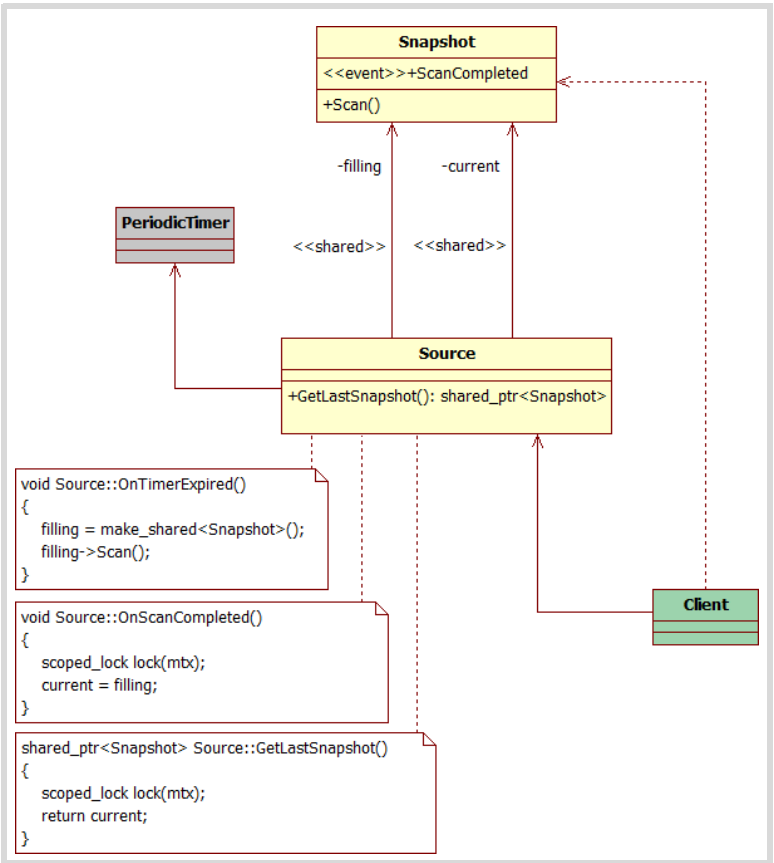


Figure 2

- Its clients must asynchronously read the most updated data available in a consistent fashion.
- Older information must be discarded when is no longer needed.

Structure

Figure 2 shows the structure.

Participants

- `Snapshot` (`SensorAcquisition`)
  - Holds the whole set of data acquired by the source.
  - Performs a complete scan.
  - Possibly provides const function members to query the acquisition.
  - Possibly is a set of (heterogeneous) linked objects (e.g., a list of `Measure` objects)
- `Source` (`SensorNetwork`)
  - Periodically asks the source to perform a new scan.
  - Provides the latest complete scan to its clients.
- `Client` (`WebService`, `ThresholdMonitor`, `Statistics`)
  - Asks the `Source` for the latest `Snapshot` available and uses it (in read-only mode).

Collaborations

- Periodically, `Source` creates a new `Snapshot` instance, assigns it to the `shared_ptr` filling, and commands it to start the acquisition.
- When the acquisition is terminated, `Source` performs the assignment `current=filling` protected by a mutex. If no clients were holding the previous `current`, the pointed `Snapshot` is automatically destroyed (by the shared pointer).
- When a client needs the most updated `Snapshot`, it calls `Source::GetLastSnapshot()` that returns current.

Figure 3 (overleaf) shows the collaborations between a client, a source and the snapshots it creates.

Consequences

- Flip Model decouples the producer from the readers: the producer can go on with the update of the data (slow) and each reader gets each time the most updated version.
- Synchronization: producer and readers can run in different threads.
- Flip Model grants the coherence of all the data structures that are read in a given instant from a reader, without locking them for a long time.
- Memory consumption to the bare minimum to ensure that every reader has a coherent access to the most recent snapshot.

Implementation

Here are 8 issues to consider when implementing the Flip Model pattern:

1. A new acquisition can be started periodically (as proposed in the example) or continuously (immediately after the previous one is completed). In the first case, the scan period must be longer than the scan duration. Should the scan take longer, it is automatically discarded as soon as the timer shoots again.

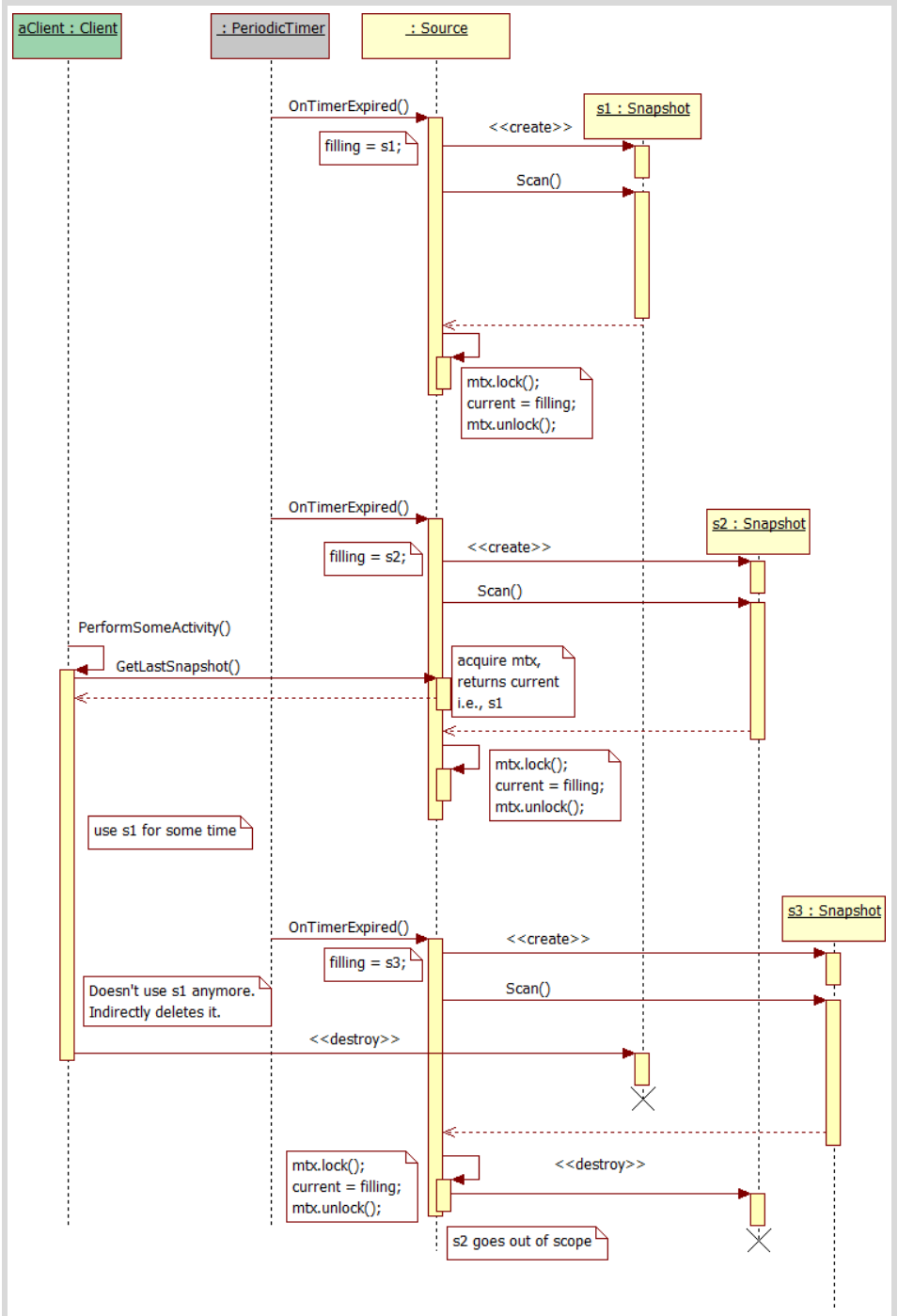


Figure 3

2. The pattern is described using C++, but it can be implemented as well in languages with garbage collection. In C++, `std::shared_ptr` is necessary to ensure that a `Snapshot` is deleted when no client is using it and `Source` has a more updated snapshot ready. In a language with garbage collection, the collector will take care of deleting old snapshots when they're no longer used (unfortunately this happens at some unspecified time, so there can be many unused snapshots in memory).
3. The `std::shared_ptr` (or garbage collection) mechanism will work correctly (i.e., old snapshots are deleted) only if clients use `Source::GetLastSnapshot()` every time they need a snapshot.
4. `Snapshot` (and the application in general) can be synchronous or asynchronous.

   In the first case, the method `Snapshot::Scan` is a blocking function and the caller (i.e., `Source`) must wait until the data structure is completed before acquiring the mutex and assigning `current` to `filling`. Within a synchronous application, clients will run in other threads.

   In the second case, the method `Snapshot::Scan` starts the acquisition operation and returns right away. When the data structure is completed, an event notification mechanism (e.g., events, callbacks, signals) takes care to announce the end of the operation to `Source`, that can finally acquire the mutex before assigning `current` to `filling`. An asynchronous application can be single-thread or multi-thread.
5. The pattern supports every concurrency model: from the single thread (in a fully asynchronous application) to the maximum parallelization possible (when the acquisition has its own threads, as well as each client).

   When the acquisition and the usage of the snapshots run in different threads, a synchronization mechanism must be put in place to protect the `shared_ptr`. While the simplest solution is to add a mutex, starting from C++11 you can use instead the overload functions `std::atomic_...<std::shared_ptr>` (and maybe from C++20 `std::atomic_shared_ptr`). A point worth noting here is that the implementation of the atomic functions might not be lock-free (as a matter of fact, my tests with the latest gcc version show that they're not): in that case, the performances are likely worse than the version using the mutex.

   A better solution could be to use an atomic `int` as a key to select the right `shared_ptr` (see the 'Sample code' section for more details).
6. The objects composing `Snapshot` (usually a huge complex data structure) are (possibly) deleted and recreated at every scan cycle. It's possible to use a pool of objects instead (in this case `shared_ptr` must be replaced by a reference counted pool object handler).
7. Please note that `Snapshot` (and the classes it represents) is immutable. After its creation and the scan is completed, the clients can only read it. When a new snapshot is available, the old one is deleted, and the clients will read the new one. This is a big advantage from the concurrency point of view: multiple clients running in different threads can read the same snapshot without locks.
8. Be aware of stupid classes! `Snapshot` (and the classes it represents) should not be a passive container of data. Every class should at least contribute to retrieve its own data, and one could also consider whether to add methods and facilities to use the data.

Sample code

The C++ code shown in Listing 1 sketches the implementation of the Flip Model classes described in the 'Motivation' section.

The code in Listing 1 uses a mutex for clarity. A lock free alternative is shown in Listing 2.

```
class SensorAcquisition
{
public:
  // interface for clients
  const SomeComplexDataStructure& Data() const
  {
     // ...
  }
  // interface for SensorNetwork
  template <typename Handler>
  void Scan(Handler h) { /* ... */ }
};
class SensorNetwork
{
public:
  SensorNetwork() :
    timer( [this](){ OnTimerExpired(); } )
  {
    timer.Start(10s);
  }
  shared_ptr<SensorAcquisition>
  GetLastMeasure() const
  {
    lock_guard<mutex> lock(mtx);
    return current;
  }
private:
  void OnTimerExpired()
  {
    filling = make_shared<SensorAcquisition>();
    // start an async operation
   filling->Scan([this](){ OnScanCompleted(); });
  }
  void OnScanCompleted()
  {
    lock_guard<mutex> lock(mtx);
    current = filling;
  }
  PeriodicTimer timer;
  shared_ptr<SensorAcquisition> filling;
  shared_ptr<SensorAcquisition> current;
  mutable mutex mtx; // protect "current"
};
class Client
{
public:
  Client(const SensorNetwork& sn) : sensors(sn) {}
  // possibly runs in another thread
  void DoSomeWork()
  {
    auto measure = sensors.GetLastMeasure();
    // do something with measure
    // ...
  }
private:
  const SensorNetwork& sensors;
};
```

Listing 1

Just in case you were wondering, you do need an atomic integer type here, although only one thread is writing it (have a look at C++ memory model [cppreference] to go down the rabbit hole).

Known uses

Flip Model is used to retrieve the periodic diagnosis of network objects in several applications I worked on. Unfortunately, I cannot reveal the details given the usual confidentiality constraints that apply to these projects.

```
class SensorNetwork
{
public:
  SensorNetwork() :
    timer( [this](){ OnTimerExpired(); } )
  {
    // just to be sure :-)
    static_assert(current.is_always_lock_free,
                  "No lock free");
    timer.Start(10s);
  }
  shared_ptr<SensorAcquisition>
  GetLastMeasure() const
  {
    assert(current < 2);
    return measures[current];
  }
private:
  void OnTimerExpired()
  {
    auto sa = make_shared<SensorAcquisition>();
    // start an async operation
    sa->Scan([this](){ OnScanCompleted(); });
    // filling = 1-current
    assert(current < 2);
    measures[1-current] = sa;
  }
  void OnScanCompleted()
  {
    current.fetch_xor(1); // current = 1-current
  }
  PeriodicTimer timer;
  std::array< shared_ptr<SensorAcquisition>, 2>
measures;
  atomic_uint current = 0; // filling = 1-current
};
```

Listing 2

Related patterns

- The pattern is somewhat similar to 'Ping Pong Buffer' (also known as 'Double Buffer' [Nystrom14] in computer graphics), but Flip Model allows multiple clients to read the state, each at its convenient pace. Moreover, in Flip Model, there can be multiple data structures simultaneously, while in 'Ping Pong Buffer'/'Double Buffer' there are always two buffers (one for writing and the other for reading). Finally, in 'Ping Pong Buffer'/'Double Buffer', buffers are swapped, while in Flip Model the data structures are passed from the writer to the readers and eventually deleted.
- `Snapshot` can/should be a 'Façade' [Gof 95] for a complex data structure.
- `Source` can use a 'Strategy' [Gof 95] to change the policy of update (e.g., periodic VS continuous).

References

[cppreference] http://en.cppreference.com/w/cpp/language/memory_model

[Gof 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995 Addison-Wesley.

[Nystrom14] Robert Nystrom (2014) 'Double Buffer' in *Game Programming Patterns*, available from http://gameprogrammingpatterns.com/double-buffer.html

[Wikipedia] 'Software Design Pattern', Documentation section: http://en.wikipedia.org/wiki/Software_design_pattern#Documentation

Memory Management Patterns in Business-Level Programs

There are many memory management patterns. Sergey Ignatchenko considers these from an application level.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Discussions on 'how we should manage memory in our programs' are ages old; the first discussion which puts together 'memory management' and 'design patterns' (and which I was able to find) was over 15 years ago, in [Douglass02] – and BTW is still very relevant.

On business-level programming

Still, in spite of the topic being rather ancient, even in [Douglass02] the discussion is a bit cluttered with details which are not directly relevant for an app-level/business-level programmer (sure, pooling is nice – but 99% of the time it should be an implementation detail hidden from business-level programmer).

In this article, I will try to concentrate on the way memory management is seen by an app-level (more specifically, business-level) developer; this means that I am excluding not only OS kernel programmers, but also developers who are implementing stuff such as lower-level libraries (in other words, if your library is directly calling `poll()`, this whole thing is not really about you). FWIW, the developers I am trying to represent now are those who are writing business-level logic – more or less similar to the logic which is targeted by programming languages such as C# and Java (while certainly not all Java/C# code qualifies as 'business-level', most of it certainly does).

A few properties of a typical business-level code:

- It changes very often (more often than everything else in sight)
- More often than not, it is 'glue' code
- Development speed is of paramount importance (time to market rulezzz)
- Making changes to existing code quickly (~='on a whim of some guy in marketing department') is even more important than 'paramount'
- Raw performance of business-level code is usually not that important (it is the way that it calls the pieces of code it glues together which matters for overall performance)

'No Bugs' Hare Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including being a co-architect of a stock exchange, and the sole architect of a game with 400K simultaneous players. He currently holds the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

Pattern: Zero Memory Management

With this in mind, we can start discussing different memory management patterns which are applicable at the business level. As business-level programming has its own stuff to deal with and memory management is rarely a business requirement, more precisely, we'll be speaking about memory management which we cannot avoid using even at a business level.

It may be difficult to believe, but there are (ok, 'were') programming languages that didn't need memory management at all (or, more precisely, with memory management so rudimentary that we can ignore it for our purposes); one example of such a language is FORTRAN77 (though it was expanded with allocatable data in FORTRAN90). Let's call it Zero Memory Management.

The idea behind Zero Memory Management is simple: as long as we say that all the variables in our program behave 'as if' they're on-stack variables (and any references to them can *only* go downstream, i.e. from caller to callee), we can easily prove that we don't need any additional memory management at all, and the program is guaranteed to be correct memory-wise without any additional efforts. In other words:

> *the best way to avoid problems with pointers is to prohibit them outright.*

BTW, Zero Memory Management doesn't prevent us from using heap; the only thing I, as a business-level developer, care about is that all the variables behave 'as if' they're on the stack. It means that things such as `std::string` and `std::unique_ptr<>` in C++ (as well as any implementation which uses heap behind the scenes in a similar manner) still qualify as Zero Memory Management (sic!).

In a sense, if we could restrict business-level programming to Zero Memory Management, it would be a Holy Grail™ from the memory management point of view: there would be no need to think about anything memory-related – our programs would 'just work' memory-wise. Unfortunately, for real-world business programs with complicated data structures, it is not that easy.

Traversing problem

One thing which cannot be expressed easily under the Zero Memory Management model is the traversing of complicated data structures. For example, let's consider the following data structure:

```
//PSEUDO-CODE
class OrderItem {
  //some data here
};
class Order {
  vector<OrderItem> items;
};
```

So far so good, and we're still well within the Zero Memory Management model, until we need, given an `OrderItem`, to find the `Order` to which it belongs.

We can say that Manual Memory Management does allow us to support arbitrarily-complicated data structures – and very efficiently too

Traditionally, this task is solved by adding a pointer/reference to `Order` into `OrderItem` – but in Zero Memory Management there is no such thing as a pointer, so we're out of luck. Of course, we can always use a pair of `(OrderItem, index-in-vector-of-OrderItems)` to express the same thing, but with more complicated data structures with multiple levels of indirections it will quickly (a) become *really* ugly and (b) start causing significant performance hits.

Let's call this issue of traversing our complicated data structure a 'traversing problem'. In practice, it happens to be bad enough to prevent us from using the otherwise very simple and straightforward Zero Memory Management model ☹.

Pattern: Manual Memory Management

The second memory management pattern on our list is the dreaded manual memory management. We're allocating something on the heap – and obtaining a pointer to it, and then it is our obligation to delete this allocated memory.

Manual memory management has been known at least for 50 years (IIRC, C's `malloc()`/`free()` was not the first such thing); and the problems with it have been known for all those years too. With manual memory management, it is very easy to make one of the following mistakes:

- Forget to delete allocated memory (causing memory leak)
- Dereference a pointer to already deleted memory (causing all kinds of trouble, with the worst cases being crashes or data corruption ☹).
- On the positive side, we can say that Manual Memory Management *does* allow us to support arbitrarily-complicated data structures – and very efficiently too. But its lack of protection from silly mistakes is a serious drawback for business-level programming.

Pattern: Reference Counting

NB: I won't engage in a discussion whether reference counting should be considered a flavor of Garbage Collection or not. As with any debate about terminology, it is outright silly; what really matters is how the things work, not how we name them.

One very simple idea to avoid Manual Memory Management is to use Reference Counting: each object has a reference counter in it, and whenever the object is no longer referenced (because the last reference to an object is dropped) – the object is deleted. Unfortunately, this approach can easily cause loops of objects which are referenced only by each other, causing syntactic memory leaks (also, the semantic memory leaks discussed in the section 'Pattern: (full-scale) Garbage Collection' below, still apply).

In C++, Reference Counting is represented by `std::shared_ptr<>`. To help with addressing the loop problem, there is a `std::weak_ptr<>` which allows us to avoid loops, but it is still the *developer*'s responsibility to make sure loops don't occur. In addition, there are certain problems with the implementation of `std::shared_ptr<>` such as the need for the *developer* to choose the lifetime of the tombstone (very briefly: if we're using `make_shared()`, we're improving locality, but this is at the cost of the memory f*or the whole object* being kept until the last `weak_ptr<>` to it is removed, which can take a while ☹).

Overall, Reference Counting is not too popular in the industry (and IMNSHO for a good reason too): on the one hand, it doesn't provide firm safety guarantees, and on the other one – for a vast majority of use cases – it is not really necessary, with `std::unique_ptr<>` (and its 'definite, predictable lifespan') generally preferred over reference counting most of the time ([Stroustrup][Parent13][Weller14], and most importantly ☺ – my personal experience; IIRC, in my career I have needed an equivalent to `shared_ptr<>` only thrice – all the cases *not* at business-level – with number of examples when `unique_ptr<>` was sufficient, going into goodness knows how many thousands).

Pattern: (full-scale) Garbage Collection

After people have tried reference counting (which still failed to achieve the Holy Grail™ of memory safety), the next thing which arose to deal with memory management was (full-scale) Garbage Collection. The premise of Garbage Collection is simple:

- Objects are allocated on the heap (but references can live on the stack).
- An object is not deleted as long as it is reachable from the stack (or from globals) via a chain of references. In other words, *as long as there is at least one way to reach the object, it stays alive.*

Garbage Collection (a) does solve the Traversing Problem, and (b) does avoid the mistakes which are typical with Manual Memory Management. But does this mean that we've found a new Holy Grail of Memory Management™ with Garbage Collection? Not really.

The Big Fat Problem™ with Garbage Collection is that if we forget to clean a no-longer-needed reference, it leads to a so-called semantic memory leak, which is a well-known plague of garbage-collected programming languages. Moreover, as has been argued in [NoBugs18], for code to be both safe and leak-free, the cleaning of such no-longer-needed references should happen *exactly at the same places where we'd call manual* `free()`/`delete` *in Manually Managed languages.*

More generally, keeping an object alive as long as there is 'at least one way to reach' it (a concept central to garbage collection) means, effectively, loss of control over object lifetimes (they become next-to-impossible to track and control in a sizeable project where anybody can silently store a reference causing all kinds of trouble); this, in turn, creates the potential for Big Fat Memory Leaks™ – and these have to be avoided in quite a few Big Fat Business-Level Programs™.

Note that these semantic leaks could be avoided (and control over object life times can be claimed back without introducing the risk of a crash) by using 'weak references' for all the references *except* those references which do express ownership, but such approaches are uncommon for garbage-collected languages (where traditionally 'weak references' are seen as a way to implement caches rather than something to prevent unwanted expansion of an object's life time, and what's more important

I have to learn not only 'how to write correct programs', but also 'how to write programs which the current Rust compiler considers correct'

for practical purposes, 'weak references' have *much* bulkier syntax than default references).

Pattern: Rust Memory Management (~='proof engine for dumb pointers')

A new kid on the memory management block is the Rust programming language, which has its own approach to ensuring memory safety. Very, very briefly: [within code which is not marked as 'unsafe'] Rust tries to *prove* that your program is memory-safe, and if it cannot prove safety – your program fails to compile.

The problem with such an approach is that the reasoning becomes convoluted, and what's *much* worse is that this proof engine becomes exposed to the developer. Moreover, this restriction is fundamental: Rust cannot (and won't be able to, *ever*) ensure that its proof engine accepts *all* correct programs (while still rejecting *all* incorrect ones); in turn, it means that as a Rust programmer, I have to learn not only 'how to write correct programs' (which is inherent for *any* programming language), but also 'how to write programs which the current Rust compiler considers correct'. In practice, this last thing happens to be a Big Pain In The ~~Ahem~~ Neck™. Personally, I'd compare the complexity and the amount of jumping through hoops when programming in Rust with the complexity of programming templates in C++ (~='manageable but really ugly especially for business-level programming') – but at least in C++, not all the code is within templates (and *using* templates in C++ doesn't suffer from additional complexity).

And as an app-level developer I certainly do NOT want to spend time thinking about Rust proof engine messages that are cryptic for anyone who doesn't know them. For example, you 'cannot borrow `foo` as immutable because it is also borrowed as mutable'. Furthermore, how I should explain my intentions to the proof engine so it can prove that the program is correct. In Rust, this corresponds to specifying explicit lifetimes.

> *Sure, it is possible to learn how the Rust proof engine works – but it is yet another Big Fat Thing™ to remember, and with a cognitive limitation of 7±2 entities we can deal with at any given time, it becomes a Big Fat Burden™ on us (='poor defenseless app/business-level programmers')*

This complexity IMNSHO stems from an original intention of Rust which I'd describe as 'let's make an equivalent of the C language and try to prove the correctness of such programs'; and as the C language is all about 'dumb pointers', this means trying to prove the correctness of an arbitrary program with dumb pointers (things don't change if we rename 'pointers' to 'references'). And this, as Rust experience IMNSHO demonstrates, is an unsurmountable task ☹ without placing *too much* burden on app-level developers (though IMHO Rust still ***may*** have its use at system level).

NB: currently, Rust is in the process of improving their proof engine with 'Non-Lexical Lifetimes' (NLL). This is supposed to improve Rust's ability to prove correctness. IMNSHO, while NLL will indeed allow Rust to prove more cases, it won't make the life of the developer significantly simpler. First, there will *still* be correct programs for which Rust cannot prove correctness, and second, when the Rust compiler fails to prove correctness, understanding *why* it has failed will become even more cryptic. As one of the NLL features is to analyze conditional control flow *across functions*, this means that **`errors become essentially non-local`**; by changing my function **`z()`** in a million-LoC project I can cause a compile-time failure in a function **`a()`** which calls **`b()`** which calls **`c()`** … which calls **`y()`** which calls my function **`z()`**. Under these circumstances, figuring out which of the functions **`a()`**…**`z()`** has to be fixed becomes a hugely non-trivial task (and with a huge potential for fingerpointing between different people/teams about implied contracts between different functions).

Pattern: Zero+ Memory Management

And last, but certainly not least, I want to describe a memory management approach which is successfully used in quite a few projects (without this fancy name, of course), ensuring very straightforward programming (and without semantic memory leaks too).

In essence, it is a Zero Memory Management pattern with some kind of {raw|safe|weak} pointers/references added to get around the Traversing Problem. This allows to keep a Zero Memory Management pattern that is very undemanding for the developer, adding only the **`very minimal`** fix necessary to address the Traversing Problem.

In essence, this approach relies on expressing data structures via a combination of (a) 'owning' pointers, and (b) 'non-owning' ones. It means that our resulting data structure is a forest, with trees of the forest built from 'owning' pointers, and 'non-owning' pointers going pretty much anywhere within this tree (in particular, in our example above adding a non-owning pointer from **`OrderItem`** to **`Order`** won't be a problem).

Formally, such a forest can express an arbitrary complex data structure. An arbitrary data structure say, in Java, corresponds to a directed graph all parts of which arbitrary data structure are reachable from certain entry points of this directed graph. But then, we can remove those edges which are not necessary for reachability (actually, replacing them with 'non-owning' pointers) and we'll get a forest which is built out of 'owning' pointers.

Less formally, in all my 20+ years of development, I can remember only three times when I have seen a data structure which doesn't *naturally* lend itself to the Zero+ Memory Management model above. Just to illustrate this observation, let's take a look at the data structures we're routinely using to express things: structures, nested structures, lists, vectors, hash tables, trees – *all* of them are *naturally* expressed via some kind of a tree built from 'owning pointers' (with an occasional 'non-owning pointer' going in the direction opposite to the direction of 'owning' ones). Those few cases when this model wasn't enough in practice were the same cases when **`shared_ptr<>`** was indeed necessary, but *all* such cases weren't at the business-level to start with.

the so-called 'generational hypothesis' says that young objects are more likely to die than old ones

Pattern: Zero+Something Memory Management

One more thing about Zero+ Memory Management is that we don't necessarily need to follow this model for *all* the objects which can arise in our program. Rather, most of the time we can separate our objects into:

- long-term objects: for example, those representing the state of our state machine – and
- temporary objects, which are going to be destroyed after we leave our current function.

This separation is also consistent with the so-called 'generational hypothesis' which says that young objects are more likely to die than old ones. With this in mind, we can say that Zero+ Memory Management is *really* important only for long-term objects; for temporary ones we can live with pretty much any other model which works for us. In one example, we may want to use Zero+ for long-term objects, and classical GC-based Memory Management for temporary ones (as long as long-term objects are properly structured, GC will indeed take care of all the temporary ones as soon as temporary objects go out of scope).

Examples of how Zero+Something Memory Management can be used with different programming languages:

- C++ with **`std::unique_ptr<>`** as 'owning' pointers, and naked C-style pointers for 'non-owning' ones. This is not safe (there is a risk of dangling pointers), but at least will protect against memory leaks. This is actually an approach which was used for quite a few serious C++ projects (including some of my own, including a G20 stock exchange and an MOG with 500K simultaneous players) – and very successfully too.
- C++ which uses **`shared_ptr<>`** as an implementation of **`owning_ptr<>`** (with copying prohibited for **`owning_ptr<>`**); this also allows for safe pointers (implemented on top of **`weak_ptr<>`**).
- 'Safe C++' along the lines described in [NoBugs18]. Essentially replaces 'non-owning' pointers with their safe counterparts – being able to provide safety guarantees (!). (NB: in [NoBugs18] there is also a big chunk of logic aimed to make naked pointers safe for those temporary objects.)
- Java with usual Java references as 'owning' pointers, and **`WeakReference<>`** as 'non-owning' ones. This will allow us to avoid memory leaks, including most existing semantic memory leaks too (at the very least, it will be much easier to track semantic memory leaks). Zero+Something approach allows us to use the usual Java references for temporary objects (i.e. *everywhere* except for long-term ones).
- C# with usual C# references as 'owning' pointers, and WeakReference (a 'short' kind(!)) as 'non-owning' ones. Similar to Java, this will allow us to avoid most existing semantic memory leaks. And also same as with Java, Zero+Something approach allows us to use usual C# references for temporary objects (i.e. everywhere except for long-term ones).

Conclusion

We considered quite a wide range of existing memory management patterns, ranging from Zero Memory Management to Rust Memory Safety. And IMNSHO, at least for business-level programming (which covers *most* of app-level programming), Zero+Something Memory Management tends to work the best. This approach allows to (a) represent pretty much everything we need, and (b) to avoid pitfalls which are typical for other models. Moreover, it has quite a significant history at least in serious C++ projects (my own ones included), with very good results. ■

References

[Douglass02] Bruce Powel Douglass (2002) *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley, https://www.amazon.com/Real-Time-Design-Patterns-Scalable-Architecture/dp/0201699567

[Loganberry04] David 'Loganberry', *Frithaes! – An Introduction to Colloquial Lapine!*, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs18] 'No Bugs' Hare, 'Java vs C++: Trading UB for Semantic Memory Leaks (Same Problem, Different Punishment for Failure)', http://ithare.com/java-vs-c-trading-ub-for-semantic-memory-leaks-same-problem-different-punishment-for-failure

[Parent13] Sean Parent (2013) C++ Seasoning, GoingNative, https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning

[Stroustrup] Bjarne Stroustrup, C++11 FAQ, http://www.stroustrup.com/C++11FAQ.html#std-shared_ptr

[Weller14] Jens Weller, shared_ptr addiction, https://www.meetingcpp.com/blog/items/shared_ptr-addiction.html

Compile-time Data Structures in C++17: Part 3, Map of Values

A compile time map of values allows code to be tested more easily. Bronek Kozicki demonstrates how to avoid a central repository of values.

In the previous two parts of the series [Kozicki18a] [Kozicki18b] we were introduced to a compile-time set of types and a map of types. In both of these data structures, all operations were guaranteed to have precisely zero runtime cost. This part is different, for two reasons:

- we are going to store actual values in the data structure. Since only literal types [cppreference] support construction during the compilation time, and we want to support the broadest possible range of types, it follows that non-literal types will have to endure run-time penalty (of construction and, possibly, copy)
- as far as the implementation is concerned, there is very little difference between the map of types (presented previously) and the map of values, shown in Listing 1. Hence, having introduced the basic programming techniques in the preceding parts, the focus of this part will be on the uses of such data structures in the modern C++, rather than its implementation details.

There is also the elephant in the room which needs to be addressed first: STL-style containers supporting compile-time operations. One such container library is **Frozen** [Frozen], but there is also an ongoing effort to make standard C++ containers compliant with the literal types requirements [Dionne18] [P0980R0] [P1004R1]. Since the topic of the series is 'compile-time data structures', there is an obvious overlap here. For a start, we are dealing with collections which support compile-time operations such as construction or lookup. The implementation of such containers will necessarily rely on a similar set of meta-programming techniques. Finally, we are talking about modern C++, and that's about all the similarities. The differences are less visible, but not less significant. The STL-style containers are designed for imperative programming style, and they are homogeneous (that is, only support single prescribed data type). This homogeneity is inherent to imperative programming style because, without it, it is somewhat tricky to populate a container, pass it around as a function parameter or its return type, or write a simple `for` loop, iterating over the container elements. The compile-time collections presented so far in the series do not appear to support iteration at all, but as the code excerpt in the Listing 2 demonstrates, this is quite achievable (although not nice) with the help of generic lambdas.

Similarly, passing parameters and returning heterogeneous collections imply the use of templates, which means a code similar to one in the code excerpt in the Listing 3 (note unconstrained `template <typename Val>` passed to `foo`). Templates such as this make it difficult to reason about the code because of the very few constraints attached. Speaking of which, we have seen 'constraints' used in our compile-time data structures in the preceding parts of the series, and they present great means of differentiation of the data structures as demonstrated in Listing 4. Since overload resolution complements template matching, it is also possible to overload based on 'constraints' set on the compile collection, like the code sample also demonstrates. Our heterogeneous containers offer one more

Bronek Kozicki developed his lifelong programming habit at the age of 13, teaching himself Z80 assembler to create special effects on his dad's ZX Spectrum. BASIC, Pascal, Lisp and a few other languages followed, before he settled on C++ as his career choice. In 2005, he moved from Poland to London, and promptly joined the BSI C++ panel with a secret agenda: to make C++ more like Lisp, but with more angle brackets. Contact him at brok@incorrekt.com

Overview of the previous parts

There is no such thing as 'O(0)' time complexity of a function (hence quotes) because time complexity implies that some action will be actually performed during the program execution. Here we are asking the compiler to perform all the actions required by the function (or more accurately, a meta-function) during the compilation itself, which allows us to use the result as an input for further compilation.

A meta-function might be a function with a `constexpr` specifier, but typically we will use either a template type (wrapped in a `using` type alias if nested inside a template type) or a `constexpr static` variable template (also nested inside a template type). In the former case, a result of a meta-function is a type, and in the latter, it is a value.

A tag type is a type which is meant to be used as a name – it has no data members and no useful member functions. The only purpose of objects of such types is the deduction of their type. Examples in the C++ standard library include `std::nothrow_t` or types defined for various overloads of `std::unique_lock` constructor.

A pure function is a function which has no side effects and, for any valid and identical set of inputs, always produces the same output. For example, any deterministic mathematical function is also a pure function. A function which takes no input, but always produces the same result and has no side-effect is also a pure function. Mathematical functions in C++ standard library are not pure functions, but this is about to change [P0533R3]. We can view many meta-functions as pure functions.

A limitation of meta-functions is that they do not have a stack in any traditional sense (they have template instantiation depth instead), and cannot manipulate variables. They can produce (immutable) variables or types, which means that they can be used to implement recursive algorithms. Such implementation will be typically a template type, where at least one specialisation implements the general algorithm, while another specialisation implements the recursion terminating condition. The compiler selects the required 'execution path' of the recursive algorithm utilising template specialisation matching.

A higher order function is a function which consumes (or produces, or both) a function. Since in our case a (meta)function is a template, we can implement a higher order (meta)function consuming a (meta)function, as a template consuming template parameter (or in other words, a 'template template parameter'). Since template types can naturally output a template type, any meta-function which is a type can trivially produce a meta-function.

A selector is some entity mapped to another – in a C++ standard library, a selector in `std::map<int, std::string>` is some value of type `int`. One way to perform such mapping during compilation is to employ overloading, which makes tag types an obvious choice of the selector. Mapping result could be either a type (deduced with the help of `decltype` keyword) or actual value, returned from an overloaded function. To avoid instantiating arbitrary types, we are going to use a small wrapper template when only a type is needed.

There is also the elephant in the room which needs to be addressed first: STL-style containers supporting compile-time operations

```
// copy the definition of set in Listing 6 from
// https://accu.org/index.php/journals/2531
// to here
namespace val_impl {
template <typename T> struct wrap {
  constexpr explicit wrap() = default;
  using type = T;
};

template <template <typename> typename CheckT,
  template <typename> typename CheckV,
  typename... L> struct impl;
template <template <typename> typename CheckT,
  template <typename> typename CheckV>
struct impl<CheckT, CheckV> {
  using selectors = set<CheckT>;
  constexpr explicit impl() = default;
  constexpr static void pair() noexcept;
  constexpr static void type() noexcept;
};
template <template <typename> typename CheckT,
  template <typename> typename CheckV, typename T,
  typename V, typename... L>
struct impl<CheckT, CheckV, T, V, L...>
  : impl<CheckT, CheckV, L...> {
  using check = typename CheckV<V>::type;
  using base = impl<CheckT, CheckV, L...>;
  using selectors =
    typename base::selectors::template insert<T>;
  static_assert
    (not base::selectors ::template test<T>);
  constexpr impl(const impl<CheckT, CheckV, L...>&
    b, const check& v) : base(b), val_(v) {}
  using base::pair;
  constexpr const auto& pair(wrap<T>) const
    noexcept { return val_; }
  using base::type;
  constexpr auto type(wrap<T>) const noexcept {
    return wrap<V>{}; }
```

Listing 1

```
private:
  check val_;
};
}
template <template <typename> typename CheckT,
  template <typename> typename CheckV,
  typename... L>
class val {
  template <template <typename> typename,
    template <typename> typename, typename...>
  friend class val;
  using impl = val_impl::impl<CheckT, CheckV,
    L...>;
  impl val_;
  constexpr explicit val(const impl& v) : val_(v)
  {}
public:
  constexpr val() = default;
  template <typename T, typename V> constexpr auto
    insert(const V& v) const noexcept
  {
    using result = val<CheckT, CheckV, T, V,
      L...>;
    using rimpl = typename result::impl;
    return result(rimpl(val_, v));
  }
  using set = typename impl::selectors;
  template <typename U> using type =
    typename decltype(impl::type
      (val_impl::wrap<U>()))::type;
  template <typename U> constexpr const
    auto& get() const noexcept
  {
    return val_.pair(val_impl::wrap<U>());
  }
  template <typename U, typename... A> constexpr
  auto run(A&&... a) const -> decltype(auto)
  {
    return get<U>()(std::forward<A>(a)...);
  }
};
```

Listing 1 (cont'd)

feature which is unattainable for traditional containers – overloading based on content (as seen with `Ver1` and `Ver2`, which imply one potential application – admittedly, such code is likely to be very brittle).

Hopefully, the code demonstrated so far is enough to trigger the 'this is interesting' response of the reader. But is it actually useful? Here is a 10 thousand feet view at such a container:

- a hardcoded 'selector' must be used to refer to an element (a value or a lambda or a type), stored within the container;
- type of each element is derived from the 'selector';
- there is no central repository of 'selectors' and each can be defined independently of others

The first two points are also available to any user-defined type in a C++ program. We use hardcoded names to refer to a field, member function, or nested type, and there is nothing new here. The last one is what makes the data structures presented here unusual. Let's have a look at the two popular design patterns where data members, or functions, are typically used.

Our heterogeneous containers offer one more feature which is unattainable for traditional containers – overloading based on content

```
namespace set_impl {

template <template <typename> typename Check,
  typename... L> struct unique;
template <template <typename> typename Check>
  struct unique<Check> {
// add this to set_impl::unique<Check> code :
  template <typename F> constexpr static bool
    apply(const F&) noexcept { return true; }
};

template <template <typename> typename Check,
  typename T, typename... L>
struct unique<Check, T, L...> {
// add this to set_impl::unique<Check, T, L...> :
template <typename F> constexpr static bool
    apply(const F& fn) noexcept
  {
    if constexpr (not std::is_void_v<T> && not
      contains_detail<Check, T, L...>::value) {
      if (not fn((T*)nullptr)) return false;
    }
    return unique<Check, L...>::apply(fn);
  }
};
} // namespace set_impl

template <template <typename> typename Check,
  typename... L> class set {
  using impl = set_impl::unique<Check, L...>;

public:
// add this to set<Check, L...> :
  template <typename F>
  constexpr static bool for_each(const F& fn)
  { return impl::apply(fn); }
};

int main()
{
  constexpr static auto s1 = set<PlainTypes>
    ::insert<int>::insert<Baz>::insert<Fuz>();
  decltype(s1)::for_each([](auto* d) -> bool {
    using type = decltype(*d);
    std::cout << typeid(type).name() << std::endl;
    return true;
  });
}
```

Listing 2

```
template <typename Val>
void foo(const Val& v)
{
  auto fuz = v.template run<Fuz>(
    v.template get<Baz>());
  // ...
}
int main()
{
  constexpr static auto m1 =
    val<PlainTypes, PlainNotVoid>{}
    .insert<Fuz>([](int v){ return Fuzzer(v); })
    .insert<Baz>(13);
  foo(m1);
};
```

Listing 3

Abstract factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes

~ Design Patterns [GoF95].

The 'interface', as referred to above is in the context of object-oriented programming, which in C++ programs translates to a base class with (pure) virtual functions. The use of this design pattern imposes runtime polymorphism on our project since (by definition) there is no way for the factory implementation to convey via the interface the dynamic type of the objects being created. We can, however, take the factory pattern and

```
template <typename T> void foo(T);

template <typename... L>
void foo(const val<PlainTypes, Plain, L...>& v)
{ /* ... */ }

template <typename... L>
void foo(const val<DataSel, Plain, L...>& v)
{ /* ... */ }

template <typename... L>
auto foo(const val<PlatformSel, Plain, L...>& v)
  -> std::enable_if_t<std::decay_t<decltype(v)>
    ::set::template test<Ver1>>
{ /* ... */ }

template <typename... L>
auto foo(const val<PlatformSel, Plain, L...>& v)
  -> std::enable_if_t<std::decay_t<decltype(v)>
    ::set::template test<Ver2>>
{ /* ... */ }
```

Listing 4

```
template <typename Factory>
void baz(const Factory& f)
{
  auto a = f.logger(); // . . .
}
struct MyFactory {
  Logger logger() const;
};
int main()
{
  MyFactory factory;
  baz(factory);
}
```

Listing 5

```
template <typename Context>
void baz(const Context& c)
{
  auto& a = c.logger; // ...
}
struct MyContext {
  Logger& logger;
};
int main()
{
  Logger logger;
  MyContext context {logger};
  baz(context);
}
```

Listing 6

```
struct CommandLine {
  CommandLine(int argv, char* const* argc);
};
enum ConfigurationSelector {};
struct Configuration {
  explicit Configuration(const CommandLine& cmd);
};
enum PlatformServicesSelector {};
struct PlatformServices {
  template <typename... L>
  explicit PlatformServices(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  {
    const auto& conf =
      *v.template get<ConfigurationSelector>();
    // ...
  }
};
enum MarketPriceInputsSelector {};
struct MarketPriceInputs {
  template <typename... L>
  explicit MarketPriceInputs(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  {
    const auto& conf =
      *v.template get<ConfigurationSelector>();
    const auto& plat =
      *v.template get<PlatformServicesSelector>();
    // ...
  }
};
```

Listing 7

```
enum ClientPriceCalcSelector {};
struct ClientPriceCalc {
  template <typename... L>
  explicit ClientPriceCalc(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  {
    const auto& conf =
      *v.template get<ConfigurationSelector>();
    const auto& mark =
     *v.template get<MarketPriceInputsSelector>();
    // ...
  }
};
enum NetworkOutputSelector {};
struct NetworkOutput {
  template <typename... L>
  explicit NetworkOutput(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  {
    const auto& conf =
      *v.template get<ConfigurationSelector>();
    // ...
  }
};

struct ClientPriceOutput {
  template <typename... L>
  explicit ClientPriceOutput(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  {
    const auto& conf =
      *v.template get<ConfigurationSelector>();
    const auto& price =
      *v.template get<ClientPriceCalcSelector>();
    auto& out =
      *v.template get<NetworkOutputSelector>();
    // ...
  }
};

int main(int argv, char** argc)
{
  const auto v0 = val<PlainTypes, PlainNotVoid>{};
  try {
    const auto cmdln = CommandLine(argv, argc);
    const auto config = Configuration(cmdln);
    const auto v1 =
      v0.insert<ConfigurationSelector>(&config);
    const auto plsrv = PlatformServices(v1);
    const auto v2 =
      v1.insert<PlatformServicesSelector>(&plsrv);
    const auto input = MarketPriceInputs(v2);
    const auto v3 =
     v2.insert<MarketPriceInputsSelector>(&input);
    const auto price = ClientPriceCalc(v3);
    auto       output = NetworkOutput(v3);
    const auto v4 =
      v3.insert<ClientPriceCalcSelector>(&price)
        .insert<NetworkOutputSelector>(&output);
    const auto clout = ClientPriceOutput(v4);
    // ...
  }
  catch (...) {
    // ...
  }
  return 0;
}
```

Listing 7 (cont'd)

transform it to the implied interface in the context of generic programming, as demonstrated in Listing 5. This approach enables us to dispose of the dynamic polymorphism since the generic programming style means that the dynamic type is available at the point of use.

Encapsulated context

Solution: Provide a Context container that collects data together and encapsulates common data used throughout the system.

~Allan Kelly [Kelly]

Interestingly, the definition of this pattern also uses the word 'container'. The selector in the case of this pattern is accessor member function which may imply the use of runtime polymorphism. However, this pattern works also with data fields and generic programming, as demonstrated in Listing 6.

Both design patterns impose a single, shared dependency on all the uses of the context or factory class. A programmer has to choose between dependency on the implementation indirectly via the generic interface (as demonstrated in Listings 5 and 6), or directly on the interface class in the OOP sense, hence imposing runtime polymorphism. The heterogeneous containers presented in this part of the series offer an alternative solution: since there is no concrete implementation class as such, the dependency is on the containers (a utility class), selectors and optionally also constraints. The constraints are considered to be optional because their definition can be as loose or as tight as desired by the user. One end of the spectrum was demonstrated in the preceding articles as generic `PlainTypes` while on the opposite end we might enforce inheritance of selectors from certain types or presence of specific markers (e.g. nested type), or even expressly list them in the form of a 'set of types'. The selectors are not optional; however, they may be defined in the context of their use, rather than some arbitrary central location (as opposed to fields or member functions of a class). This distributed nature of selectors removes the single shared dependency we have seen earlier. Additionally, with the use of lambdas, we can define both a context and a factory, within the same instance of the map of types (demonstrated in Listing 3).

The lack of central location comes with one more benefit. Imagine we have hypothetical 'real world' financial software, with the following internal dependencies:

- command line parser has no dependencies;
- configuration depends on the command line parser;
- platform services (logging, network RPC, monitoring etc.) depends on the configuration;
- market price inputs depend on network RPC and configuration;
- client price calculation depends on market price inputs and configuration
- network output depends on the configuration
- client price output depends on configuration, client price calculation and network output (to send the price to)

Such dependencies imply that objects have to be instantiated in a specific order. If we were to use a context pattern to carry the configuration, platform services, market price inputs, network outputs etc. then our choices are to either:

- create multiple Context classes with increasing number of (reference) fields, or
- create a single Context class with optional fields and implement runtime checks that the dependencies are set when required

Neither is of those is very appealing. The latter (as simpler) is likely to be preferred, but it will cause brittleness of code and enable a whole category of bugs. The value map, as presented here, allows a solution which enforces the correct order of instantiation in the compile time – see Listing 7. Admittedly, multiple copies of the map and the use of pointers may seem like a 'code smell', but such code is also very readable. The more refined version presented in Listing 8 makes use of lambdas instead One point of note is that in this version, the bodies of constructors use `v.template run<...Selector>(v)` instead of `template get<...Selector>()`. We have also changed `ClientPriceOutput` to a template to own the `NetworkOutput` subobject and made the class `NetworkOutput` non-copyable, even though it is returned by value from the lambda defined for `NetworkOutputSelector`. Such construction by-value of non-copyable objects is guaranteed to work thanks to obligatory copy elision in C++17 (users of Visual Studio 2017 version 15.8 will have to find a workaround or downgrade to 15.7 because of a regression bug [VS]). The attentive reader will notice how this last code excerpt mixes both context and factory pattern.

There is one more question remaining to answer 'why bother?' The answer is the same as for any other kind of polymorphism 'to make the code testable'. The use of templates enables us to avoid the cost of dynamic polymorphism (not only in the runtime but also boilerplate OOP interface code), while the value map removes the need for a single central repository. With its help, our unit tests will be able to easily pass mock objects to the code being tested as demonstrated in Listing 9 (for objects created on the stack) and Listing 10 (for objects returned by lambdas). Note that the static object lifetime of `MockConfiguration` in Listing 10 will delay the destruction of `config` object until unit tests termination. A good workaround is to store our mock objects on the stack and then pass them to lambda as a reference, as demonstrated with `MockPlatformServices`. Readers will hopefully find other useful applications of the compile-time polymorphism offered by the map of values.

We started the series with the demonstration of the most useful meta-programming techniques, in the context of functional programming where the program is run by the compiler. The 'set of types' presented in the first part, even though rather simple, enables some interesting uses – among them a constraint to validate that certain template parameters are within a predefined set of types (which is left as an exercise to the reader). Then we moved to the, perhaps not very exciting, 'map of types' and finally, in this part discussed what I like to call a 'heterogenous compile-time container', demonstrating how such data structure can be used to solve real problems. The work is not finished, but the code discussed here is usable and available for readers to download from the **Juno** library [Juno]. The library is in very early stages of development and open to significant changes, but the similar solutions to presented here have found their use in the 'real world' code already. The very liberal MIT license allows anyone to 'lift' the parts of the source and use in their projects, which is preferred over taking the dependency (due to library immaturity). Readers are invited to the discussion on the future development, design and interface, in the 'issues' section of the library or sending me email directly. All inputs are welcome! ■

References

[cppreference] 'C++ named requirements: LiteralType' at https://en.cppreference.com/w/cpp/named_req/LiteralType

[Dionne18] Louis Dionne 'Compile-time programming and reflection in C++20 and beyond' from CppCon 2018, available at https://youtu.be/CRDNPwXDVp0

[Frozen] 'Frozen – zero cost initialization for immutable containers and various algorithms', https://blog.quarkslab.com/frozen-zero-cost-initialization-for-immutable-containers-and-various-algorithms.html

[GoF95] *Design Patterns, Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley 1995

[Juno] The Juno library: https://github.com/Bronek/juno/releases

[Kelly] Allan Kelly (updated 2009) 'Encapsulated Context' (design pattern), available from https://www.allankellyassociates.co.uk/patterns/encapsulated-context/

[Kozicki18a] Bronek Kozicki (2018) 'Compile-time Data Structures in C++17: Part 1, Set of Types' in *Overload* 146, available at https://accu.org/index.php/journals/2531

Static variables inside lambdas

Users coming from other languages may have some difficulty with static variables defined inside a lambda. Here is how Anthony Williams explained this on the accu-general mailing list:

> A lambda is an object of an unnamed type. The lambda body is the `operator()` member function of this type. For any given lambda expression there is only one such type and one such member function, so there is only one copy of each static object declared within the lambda body.

In practice, this means that the lambdas which make use of static variable will 'cache' the once-created object, making them equivalent to accessors of the context pattern with lazy evaluation. The afflictions of static objects do apply (e.g. static object lifetime, the accidental coupling of points of use if the object passed as a mutable reference) but are moderated by the fact that each lambda is its own type.

[Kozicki18b] Bronek Kozicki (2018) 'Compile-time Data Structures in C++17: Part 2, Map of Types' in *Overload* 147, available at https://accu.org/index.php/journals/2562

[P0533R3] `constexpr` for `<cmath>` and `<cstdlib>`, 5 August 2018, available at: https://wg21.link/p0533

[P0980R0] 'Making `std::string` constexpr' at https://wg21.link/P0980

[P1004R1] 'Making `std::vector` constexpr' at https://wg21.link/P1004

[VS] 'MSVC 15.8 C++17 RVO regression for non-static data members: https://developercommunity.visualstudio.com/content/problem/318693/msvc-158-c17-rvo-regression.html

```
enum ConfigurationSelector {};
struct Configuration {
  template <typename... L>
  explicit Configuration(const val<PlainTypes,
    PlainNotVoid, L...>& v,
    const CommandLine& cmdln)
    { /* ... */ }
  };
// use template run<...> as appropriate, e.g. :
enum NetworkOutputSelector {};
struct NetworkOutput {
  NetworkOutput(const NetworkOutput&) = delete;
  template <typename... L>
  explicit NetworkOutput(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  {
    const auto& conf =
      v.template run<ConfigurationSelector>(v);
    const auto& plat =
      v.template run<PlatformServicesSelector>(v);
    // ...
  }
};
template <typename Out> struct ClientPriceOutput {
  Out out_;
  template <typename... L>
  explicit ClientPriceOutput(const val<PlainTypes,
    PlainNotVoid, L...>& v)
  : out_(v.template run<NetworkOutputSelector>(v))
  {
    const auto& conf =
      v.template run<ConfigurationSelector>(v);
    const auto& price =
      v.template run<ClientPriceCalcSelector>(v);
    // ...
  }
}
```

Listing 8

```
int main(int argv, char** argc)
{
  try {
    const auto cmdln = CommandLine(argv, argc);
    const auto v = val<PlainTypes, PlainNotVoid>{}
    .insert<ConfigurationSelector>([cmdln]
      (const auto& v) -> const Configuration& {
        static auto config = Configuration(
          v, cmdln);
        return config;
    }).insert<PlatformServicesSelector>([]
      (const auto& v) -> const PlatformServices& {
        static auto plsrv = PlatformServices(v);
        return plsrv;
    }).insert<MarketPriceInputsSelector>([]
      (const auto& v) -> const MarketPriceIputs& {
        static auto input = MarketPriceInputs(v);
        return input;
    }).insert<ClientPriceCalcSelector>([]
      (const auto& v) -> const ClientPriceCalc& {
        static auto price = ClientPriceCalc(v);
        return price;
    }).insert<NetworkOutputSelector>([]
      (const auto& v) -> NetworkOutput {
        return NetworkOutput(v);
    });
    const auto clout =
      ClientPriceOutput<NetworkOutput>(v);
    // ...
  }
  catch (...) {
    // ...
  }
  return 0;
}
```

Listing 8 (cont'd)

```
TEST("Test MarketPriceInputs")
{
  const auto config = MockConfiguration();
  auto       plsrv = MockPlatformServices();
  const auto v1 = val<PlainTypes, PlainNotVoid>{}
    .insert<ConfigurationSelector>(&config)
    .insert<PlatformServicesSelector,
      const MockPlaformServices*>(&plsrv);
  auto input = MarketPriceInputs(v1);
  plsrv.push("Some test inputs");
  // ...
}
```

Listing 9

```
TEST("Test MarketPriceInputs")
{
  auto       plsrv = MockPlatformServices();
  const auto v = val<PlainTypes, PlainNotVoid>{}
    .insert<ConfigurationSelector>([](auto&){
      static auto config = MockConfiguration();
      return config;
    }).insert<PlatformServicesSelector>([&plsrv]
      (auto&) -> const MockPlatformServices&
      { return plsrv; });
  auto input = MarketPriceInputs(v);
  plsrv.push("Some test inputs");
  // ...
}
```

Listing 10

Algol 68 – A Retrospective

Algol 68 has influenced programming languages in many ways. Daniel James reminds us just how many.

This month marks the 50th anniversary of the inception of the Algorithmic Language Algol 68 – the first programming language I ever learned, back in 1974 when I first encountered these things called computers, as an undergraduate.

I wrote 'inception' … perhaps a little background overview is required.

Nowadays we have more programming languages than you can shake the proverbial stick at, and computers that can shake the stick for you, but in the early 1960s there were really only three major languages available: COBOL for business programming, and FORTRAN and Algol for scientific work (LISP was also around, and deserves an honourable mention, but I'd hesitate to call it 'major'). The latter two were designed with slightly different goals in mind: FORTRAN (**FOR**mula **TRAN**slation) was developed at IBM specifically for programming computers, while Algol (**ALGO**rithmic **L**anguage), developed by a European and American team at ETH Zurich, was designed also to enable the expression of algorithms in a concise but human-readable form.

FORTRAN was the more widely used – in part because it had IBM behind it, but also because it was seen as the more portable language. The designers of Algol decided not to specify standard input and output operations as they felt that language implementers were better placed to define these facilities for the platform on which they were working. This meant that each new implementation of Algol had its own non-portable i/o functions, and this lack of portability had an impact on the acceptance of Algol.

As the 1960s progressed, a number of new languages appeared and the existing languages were further developed. The IFIP Algol Working Group 2.1 [Wikipedia-1] was formed in 1962 to design a replacement for Algol 60 that was then code-named Algol X. Many ideas were discussed, some in the light of experience of similar features in other languages, but no complete implementation of any proposed candidate for Algol X was produced before the language was formally accepted. This acceptance came when the WG 2.1 members met in December 1968 and voted to accept the language specification [Original] as it then stood, without ever seeing a working prototype. This is the language that became known as Algol 68.

Their decision was by no means unanimous, with some of the committee saying that the language had become over-complex and had lost the simplicity of Algol 60. Some of these famously went on to publish their dissent as the 'Minority Report' [Minority], in which they claimed that the language was a failure.

The style of the language reports deserves a mention. The reports are written using a two-level grammar in which the first level defines a meta-language, and the second defines Algol 68 in that meta-language. This style is known as a Van Wijngaarden grammar, after Adriaan van Wijngaarden who invented it and was also instrumental in bringing about the final design of Algol 68. The style of the reports also accounts for their legendary impenetrability to the casual reader.

In July 1970 a conference to discuss Algol 68 implementation [Peck70] took place. While many of the delegates presented papers that discussed the difficulty of implementation of the new language, the small team from the Royal Radar Establishment presented their working compiler for a substantially complete subset of the language that they called Algol 68-R [Wikipedia-2]. Algol68-R was already in daily use for production code on the ICL 1907F computer at RRE, where it had replaced Algol 60 for new development. The Algol 68-R compiler was later distributed without charge by ICL to other users of 1900-series hardware, and it was this compiler that I first used when I went to University.

Development of Algol 68 continued into the 1970s and a revised edition of the Algol 68 Report [Revised] was published in 1975. The team at RRE (by then known as the Royal Signals and Radar Establishment) produced a new version of their compiler called Algol 68-RS, which ran on, among other things, the newer ICL 2900-series mainframes and Dec VAX minicomputers.

A brief overview

So, what sort of language is Algol 68? I'm going to talk mostly about the language of the Revised Report because it is a little more complete than the original language and has fewer idiosyncrasies, but the differences are slight and mainly cosmetic.

An expression-oriented, block-structured, procedural language

Algol 68 is an 'expression-oriented' language in that almost every statement – what the Revised Report calls a ***unit*** (the original report used the term 'unitary clause') – delivers a value, and so can be treated as an expression. Every expression can also legally be used as a statement. If a ***unit*** has no value to deliver, it returns the special type `VOID`. If an expression that yields a value but doesn't do anything with it, such as

```
41+1
```

appears as a statement on its own, the compiler is likely to offer a warning that the result is to be discarded, but the code is legal.

Declarations aren't classed as ***unit***s, and aren't expressions. A sequence consisting of ***unit***s or declarations and ***unit***s separated by semicolons is called a ***series*** (originally 'serial clause') and has the value of its last ***unit***. Note that the semicolon is a separator, not a terminator: there shouldn't be one after the last ***unit*** in a ***series***.

Algol 68 is a block-structured language in that a program is composed from nested lexical blocks. Each new block introduces a lexical scope that limits the lifetime and visibility of constructs declared within that scope. Each new scope inherits from any containing scope. A block – and everything

Daniel James Daniel has carried on programming in a variety of languages ever since that first brush with Algol 68 in 1974, and the sparkle has never quite worn off. He sometimes wonders how his life would have turned out if the friend who introduced him to computing had been a FORTRAN user, as all good scientists were supposed to be. He can be contacted at djames@sonadata.co.uk

Each new block introduces a lexical scope that limits the lifetime and visibility of constructs declared within that scope

it contains – is syntactically a ***unit*** whose value is the value of the last statement in the block.

A ***block*** may simply be one or more statements enclosed between **BEGIN** and **END**, or may be a conditional such as **IF** or **CASE**, or a loop.

Algol 68 is a procedural language in that its programs are built-up from procedures. A ***procedure*** in Algol 68 may be a function that returns a result, or may be a pure procedure that returns no result, in which case its return type is specified as **VOID**. The body of a procedure is a ***unit***.

As is the case in Algol 60 and some other languages (such as Pascal), but not C or C++ (at least until lambda functions were introduced in C++11) procedures in Algol 68 do not have to be declared at the outermost lexical scope but can be declared within a program block, even a block of another procedure.

Algol 68 also supports user-defined operators, which are defined identically to procedures except that operators can have only one or two arguments. The user can set the priority (from 1 to 9) of an operator symbol, and can define new operator symbols. Operators can be overloaded, however, while procedures cannot.

Basic types

Data types in Algol 68 are called **MODE**s. The **MODE**s provided as standard are mostly unsurprising (see Table 1).

Additional **MODE**s are defined in terms of these simple types. A **MODE** declaration associates an identifier with a **MODE** as an alias for the full definition.

Derived types

References

Algol 68 supports reference types using the **REF** keyword, so a reference to an **INT** type would be written **REF INT**. Syntactically Algol 68 references are more like C pointers than Java references, and they can be set to a null value which is written **NIL**.

Algol 68 uses the term 'name' when talking about anything that has a reference type. This term is equivalent to 'lvalue' in C, and means that a value can be assigned to it. The term 'identifier' is used to refer to symbolic names used in source code.

References can be compared for equality using the built-in **IS** and **ISNT** operations.

INT	Signed Integer. Single machine word.
REAL	Single-precision floating point number.
COMPL	Complex number.
BITS	Binary value, packed Booleans.
BYTES	Packed characters (provided for efficiency on architectures that are not byte-addressable, such as the ICL1900)
All of the above modes may be prefixed with **LONG** or **LONG LONG** to get double or higher precision, depending on the implementation.	
CHAR	Character. Implementation defined representation.
BOOL	Boolean value

Table 1

Arrays

Algol 68 supports single- and multi-dimensional arrays. Subscripts are written between square brackets.

```
MODE MATRIX = [1:5,1:5] REAL
```

The lower bound of an array defaults to 1, but may be any value less than the upper bound – even negative. An array 'knows' its bounds, and these can be queried at runtime using **LWB** and **UPB**. Like Algol 60 and C (but unlike FORTRAN) Algol 68 stores arrays in row major order.

Arrays can be flexible. The built-in **STRING** datatype is defined as a flexible array of character values:

```
MODE STRING = FLEX [1:0] CHAR
```

Arrays can be sliced. If **r** is a **[1:10]INT** then **r[3:7]** is a **[1:5]INT**.

The use of flexible arrays implies the use of dynamic (heap) allocation. No explicit deallocation is necessary as this is managed using garbage collection.

Structures

Data structures are defined with the keyword **STRUCT**. For example the standard type **COMPL** is defined as:

```
MODE COMPL = STRUCT( REAL re, im )
```

Fields of a structure are selected using the **OF** keyword (not an operator):

```
re OF complexvalue
```

A structure type may not contain members of that type, but may contain references to that type, so it is possible to write linked lists, binary trees, etc. Listing 1 shows how this may be used to implement linked list functionality.

The identifiers given to the fields of a structure are part of its type, so two structures containing the same types of fields but with different field names would not be the same type.

Unions

Unions are defined with the keyword **UNION**.

```
MODE STRINT = UNION( STRING, INT )
```

An Algol 68 union 'knows' the type of its current value, and this may be queried at runtime.

The identifiers given to the fields of a structure are part of its type, so two structures containing the same types of fields but with different field names would not be the same type.

```
# (Unordered) Linked List example #
# Demonstrates using HEAP storage #
MODE ELEMENT = STRING;
MODE NODE =
  STRUCT ( ELEMENT value, REF NODE next );
MODE LIST = REF NODE;
LIST empty = NIL;

# Add an element to the end of the list #
PROC append = ( REF LIST list, ELEMENT val ) VOID:
BEGIN
  IF list IS empty
  THEN
    list := HEAP NODE := ( val, empty )
  ELSE
    REF LIST tail := list;
    WHILE next OF tail ISNT empty
    DO
      tail := next OF tail
    OD;
    next OF tail := HEAP NODE := ( val, empty )
  FI
END;

# Add an element to the beginning of the list #
PROC prepend =
  ( REF LIST list, ELEMENT val ) VOID:
  list := HEAP NODE := ( val, list )
```

Listing 1

```
# Print the list out - using a loop #
PROC print list = ( LIST start ) VOID:
BEGIN
  IF start ISNT empty
  THEN
    REF NODE ptr := start;
    WHILE
      print(( "-- ", value OF ptr, newline ));
      next OF ptr ISNT empty
    DO
      ptr := next OF ptr
    OD
  FI;
  print(( "End of list", newline ))
END;

# Print the list out - using recursion #
PROC print list recursively = ( LIST start ) VOID:
BEGIN
  IF start IS empty
  THEN
    print(( "End of list", newline ))
  ELSE
    print(( "-- ", value OF start, newline ));
    print list recursively( next OF start )
  FI
END;
```

Listing 1 (cont'd)

Declaration and assignment

A declaration associates an identifier with some value, and uses the 'equals' symbol **=**. The following introduces a new identifier called **ultimate answer** that represents the integer value 42:

```
INT ultimate answer = 42
```

Note that white space is not significant within an identifier, so this could equally well be written **ultimateanswer** or **ulti mate answ er**.

A variable can be created simply by naming it and its type:

```
REAL half pi
```

The example above introduces a new identifier **half pi** and associates it with the storage for a variable in the current scope. This is shorthand for:

```
REF REAL half pi = LOC REAL
```

which says that **half pi** is a constant identifier that represents a reference to a **REAL** and is initialized to some location allocated on the stack. Although this long form is used by a lot of the early Algol 68 literature, it isn't really necessary. It is a bit like saying that the declaration of a float variable in C++ is equivalent to:

```
float & half_pi =
  *(float*)alloca( sizeof(float) );
```

It does, however introduce the concept of a ***generator***. In the example above, **LOC REAL** is a ***generator*** that allocates local (stack-based) storage for a **REAL** variable. Whilst it is never necessary to use this long form with the explicit **LOC** ***generator*** to create storage for a local variable, the use of a **HEAP** ***generator*** is the only way to allocate variables on the heap.

A value can be assigned to a variable using the 'becomes' symbol **:=** in what Algol 68 calls an 'assignation statement'.

```
half pi := pi/2
```

Declaration and initialization can be combined into a single statement:

```
REAL half pi := pi/2
```

Note that **pi** is a built-in constant of type **REAL** (higher precision versions – **long pi**, etc. – also exist).

The difference between the constant declaration (using the 'equals' symbol, **=**) and the declaration and initialization of a variable (using the 'becomes' symbol, **:=**) is slight, and they are easy to confuse.

Spelling IF and CASE backwards to indicate the end of the block that they introduce is a device that Algol 68 uses in a number of places

Algol 68 also provides 'updating' operators like **PLUSAB** ('**PLUS A**nd **B**ecomes') which can be abbreviated **+:=**. This syntax has been adopted by many other languages since.

Control flow

Conditions are supported by the choice statements, **IF** and **CASE**. These are identical in structure but the **IF** statement bases its selection on a **BOOL** value, while the **CASE** bases its on an **INT** value:

```
IF boolvalue THEN some expression ELSE other
expression FI

CASE intvalue IN expr1, expr2, …, exprN OUT
another expr ESAC
```

IN and **OUT** are **CASE**'s equivalents of **THEN** and **ELSE**. **ELIF** and **OUSE** are provided as shorthand forms of **ELSE IF** and **OUT CASE**.

That these really are equivalent constructions becomes more apparent when we learn that a **BOOL** value can be converted to an **INT** using the **ABS** operation, and that **ABS TRUE** has the value 1, while **ABS FALSE** is 0.

As with all statements in Algol 68, a choice statement may return a value, the value returned is the value of whatever expression is chosen (so long as all the expressions return the same **MODE**). While C and some other languages have a 'ternary operator' that allows one of two values to be used depending on some boolean condition. Algol 68 has no need of such a construction because the structure of the language allows an ordinary **IF** statement to be used.

This allows for constructions like:

```
days in month :=
  CASE month OF date IN
    31,
    IF is leap year( year OF date ) THEN 29 ELSE 28
    FI,
    31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  ESAC
```

IF and **CASE** introduce lexical scopes, just as **BEGIN** does. The scope ends with the closing **FI** or **ESAC** statement.

Spelling **IF** and **CASE** backwards to indicate the end of the block that they introduce is a device that Algol 68 uses in a number of places. The use of **FI** and **ESAC** has been adopted in some other languages, notably in the Bourne shell, while still more have taken up the idea but opted for keywords such as **ENDIF** – perhaps we should be grateful that **END** is not spelt **NIGEB** in Algol 68!

It's interesting that C++17 introduces new '**if** with initializer' [P0305R0] and '**switch** with initializer' features that let one declare variables in the condition part of an **if** or **switch** statement that have the scope of the whole statement. This enables one to write things like:

```
if( int val = get_value(); is_valid( val ) )
{
  do_stuff_with( val );
}
```

which is directly equivalent to the Algol 68 code

```
IF INT val := get value(); is valid ( val )
THEN
  do stuff with( val )
FI
```

except that the Algol 68 form is more general.

Note that with the Algol 68 **CASE** statement there is no possibility that one case can run on into the next (even should you want it to) so there is need for any break-type statement, as in C (nor does the language support one).

Loops are supported by the very versatile Algol 68 **DO** loop, which has the general form:

```
FOR counter FROM start BY step TO stop WHILE
condition
 DO something OD
```

Everything here is optional, apart from the **DO something OD** part, which is the loop itself, the other parts just provide conditions that control the termination of the loop.

The **counter**, if present, is an integer whose scope is limited to the loop; by default it counts from 1 in increments of 1 but that can be changed using the **FROM** and **BY** parts. The loop terminates when the variable reaches the **stop** value, if specified, or when the condition becomes true … otherwise it will run for ever. The counter cannot be assigned to within the loop – it is not a variable but an **INT** constant that is re-created with a new value each time around the loop.

The condition of a loop including a **WHILE** part is syntactically a series of declarations and statements. This syntax caters for cases in which all or part of the body of the loop is to be executed before the **WHILE** condition is evaluated.

```
WHILE INT n; read( n ); is valid input( n )
  DO process( n ) OD
```

If the whole of the loop is to be executed before the test (as with a **do ... while** loop in C, for example) the do-nothing operation **SKIP** can be used to satisfy the syntactic requirement for one or more statements between **DO** and **OD**. The Algol 68 **DO** loop offers all the varieties of loop control that are provided by the **for**, **while**, and **do ... while** constructions in C but with greater flexibility.

Unlike most other constructions in Algol 68, a loop does not return a result.

Here are some examples. These also show three forms of **COMMENT**, and a call to the built-in **print** function. Note that the call to print uses two sets of parentheses because multiple values are being printed; the inner pair belong to the representation of the single argument, which is in effect an array of **UNION**s of various types.

```
# Zero all elements of ar from lower to upper
  bound #
FOR index FROM LWB ar TO UPB ar DO
  ar[index] := 0
OD
```

No special syntax is needed for lambda, the functionality is a natural consequence of the design of the language

```
CO enhance a until it is greater than some
threshhold CO
FOR i WHILE a < threshhold DO
  print( ( "After ", i, " iterations a is ", a,
    newline ) );
  a := enhance( a )
OD

COMMENT sound an alarm 10 times COMMENT
TO 10 DO sound alarm OD
```

For completeness, and to annoy purists, Algol 68 also supports **GOTO**.

Procedures

Algol 68 procedures take a number of parameters, possibly zero, and return a result or VOID. A procedure is defined like so:

```
PROC twice as much = (INT i) INT: i+i
```

This defines a procedure named **twice as much** which takes a single integer parameter by value, and returns an integer result. The body of this procedure is a single statement and so doesn't need to be enclosed in a block, but most procedures do.

In this definition, the procedure denotation **(INT i) INT: i+1** is assigned to the identifier **twice as much**, which therefore has the mode **PROC (INT) INT**. As with data quantities, though, the procedure body could be assigned to a procedure variable identifier

```
PROC operation := (INT i) INT: i+i
```

We could then, elsewhere in the program, change the value of that procedure:

```
operation := (INT foo) INT: 3*foo+7
```

Listing 2 shows another example of a function denotation being used as a lambda.

The procedure body that we assign in this way is effectively a lambda function. No special syntax is needed for lambda, the functionality is a natural consequence of the design of the language.

Transput

Transput is the word used in the Algol 68 reports to refer to both input and output.

One of the big shortcomings that was recognized in Algol 60 was its lack of any standardized i/o system. The designers of Algol 68 addressed this lack by introducing a standard system for both formatted and formatless output. We have already seen the procedure **print**, which accepts any simple argument and outputs it in a standard way. This tends not to be very pretty as, for example, numbers are output in their maximum width and precision. This output can be tamed somewhat using built-in functions **whole**, **fixed**, and **float** that convert numbers to strings with specified width and precision.

```
# Print a table of values #

# print a table of values of f(angle) #
# for angle from start to stop (degrees) #

PROC print trig =
  ( REAL start, REAL stop,
    PROC (REAL) REAL f ) VOID:
BEGIN
  REAL val := start;

  printf( $" Deg |"$ );
  FOR i FROM 0 TO 9 DO printf(( $z-d.dx,b(l,x)$,
    i/10, i=9 )) OD;
  printf( $5"-" "+" 71"-"l$ );

  FOR col FROM 0 WHILE val < stop
  DO
    IF col MOD 10 = 0
    THEN
      printf(( $zd.d" | "$, val ))
    FI;

    printf(( $+d.3d,b(l,x)$, f((val*pi)/180),
      col MOD  10 = 9 ));
    val +:= 1
  OD
END;

# Print a table of sin(x) using the built-in
  sin function #
print(( newline, "Printing sin x", newline ));
print trig( 0, 90, sin );

# Print a table of sin^2(x)+cos^2(x) using a lambda
#
print(( newline, "Printing sin^2 x + cos x",
  newline ));
print trig( 0, 90, (REAL n) REAL: sin(n)^2+cos(n))
```

Listing 2

There is also a system for formatted output that gives the programmer complete control over the format of the output by means of a format designator. This is similar to the format string of C's **printf**, but is a special Algol68 **FORMAT** type, delimited by **$** characters, and can be evaluated at compile-time for efficiency. A **FORMAT** consists of one or more 'pictures', separated by commas, each of which describes the layout of a single item.

Formatless input and output are achieved using the functions **put** and **get**. We have already seen the **print** function, which is equivalent to **put** but always uses the default standard output channel called **stand out**

the language I was always told would never catch on because it was too big for any but the largest of computers runs nicely on a Raspberry Pi Zero, on battery power, sitting on the palm of my hand

(similar to **cout** in C), there is also a **read** function that is equivalent to **get** using **stand in**. Formatted i/o uses **putf** and **getf** (and **printf** and **readf**).

These three produce identical output on **stand out**:

```
put( stand out, (i, blank, r, newline ));
print(( i, blank, r, newline ));
printf(( $gx,gl$, i, r ))
```

If **i** is an integer with the value 42 and **r** is a **REAL** with the value 22/7 then this would produce three identical lines of:

```
+42 +3.14285714285714e  +0
```

In the **put** and **print** cases, the list of items to be printed are enclosed in brackets because they are conceptually an array of values to be output. These include blank and newline, which are **PROC(REF FILE) VOID** functions that are called to manipulate the output stream.

In the **printf** case, the argument list begins with a **FORMAT** containing two pictures using **g** tags meaning that the values should be printed in the same default format as is used for formatless output, and **x** and **l** tags that supply the blank and newline.

Printing values without formatting is not the point of formatted output, of course, and one might more typically see something like

```
printf(( $4z-dx,+3d.5dl$, i, r ))
```

The format string contains two pictures. The first depicts an integer with 4 digits that will be printed as blanks if they are all leading zeros (four **z** tags) followed by one digit that will always be visible (a **d** tag) and a space (**x** tag), with a sign character that will be a blank if the value is positive immediately to the left of the first non-zero digit (**-** tag) and then a blank (**x** tag). The second picture depicts a real number with a sign always shown (**+** tag), and with three digits (three **d** tags) before and five digits after (five **d** tags) a decimal point (**.** tag), and then a newline (**l** tag).

With the values of **i** and **r** above, this would produce:

```
42 +003.14286
```

Listing 2 shows some examples of formatted output.

Notation

In the Revised Report, Algol 68 is written in lower-case letters, with keywords emphasized in bold. Text files holding program source to be read by compilers don't provide a mechanism for the representation of bold text, so keywords are typically written in upper-case, as I have done here. The original Algol 68-R compiler ran on ICL 1900 computers, which had a basic character set of only 64 characters. Lower-case characters were available only through a complicated system of shift prefixes, and weren't used in programming. In Algol 68-R, keywords were distinguished from identifiers by enclosing them in single quotes, which was called 'quote stropping'.

```
'BEGIN'
'INT' I := 42;
PRINT(( I, NEWLINE ))
'END'
```

Other upper-case systems used 'dot stropping', in which a keyword was prefixed with a dot character.

I've deliberately chosen a rather 'wordy' notation for the examples in most places in this article, but it is worth noting that Algol68 provides symbolic short forms for many operations. For example the pseudo-operators **IS** and **ISNT** can be written **:=:** and **:/=:** and the **MOD** operator can be written **%***.

Algol 68 also allows round brackets to be used for **BEGIN** and **END** … or to look at it another way, Algol 68 treats any expression in round brackets as a **BEGIN**/**END** block.

Round brackets can also be used for **IF**, **FI**, **CASE**, and **ESAC**; vertical bar characters for **THEN**, **ELSE**, **IN**, and **OUT**, and vertical bar followed by a colon for **ELIF** and **OUSE**. It is usually clearer to spell things out, but the short forms can be useful when writing a one-liner.

Implementations

Algol 68 has been implemented on quite a variety of hardware platforms. There's a good summary in the Wikipedia article [Wikipedia-3] and a lot of detail on the Software Preservation Society's Algol 68 page [SPS]. In its heyday there were several implementations written in different countries – including the UK, the US, the Netherlands, and Russia – and supporting a wide range of hardware architectures. It was never as widely used as FORTRAN or COBOL, but it had a significant following.

Algol 68 was also the basis for a number of other languages, including the S3 job-control language of the VME/B operating system of ICL 2900 series mainframes, and influenced a good many more.

Most implementations of Algol 68 are no longer in use, as they were developed for computer architectures that are now obsolete. Some are still accessible in one form or another, for example the Algol-68R compiler can still be run under a GEORGE 3 emulator [GEORGE], part of the Algol 68RS compiler survives as the front end of the Algol 68 to C translator [A68TOC] from the ELLA project developed at RSRE and has been Open-Sourced, one of the two implementations named Algol 68S has been ported to a number of more modern platforms and Open-Sourced. A more recent implementation from 2001, Algol 68 Genie [van der Meer16], runs on Windows, Linux, and Mac systems and is Open Source; if you run a Debian-derived Linux distro such as Ubuntu or Raspbian you will find it in the standard repository.

Algol 68 has always had the undeserved reputation of being a large, complex, language for mainframe computers only. The language is actually smaller than C [Henney18], and certainly much smaller than many modern languages. I find it pleasingly ironic that the language I was always told would never catch on because it was too big for any but the largest of computers runs nicely on a Raspberry Pi Zero, on battery power, sitting on the palm of my hand – but perhaps that says more about developments in hardware in the last 50 years than it does about software.

Algol 68 has influenced many other languages, many of which are still in widespread use today

Legacy

Algol 68 has influenced many other languages, many of which are still in widespread use today.

The influence of Algol 68 on C is clear, and Algol 68 is referred to many times by Bjarne Stroustrup when describing the development of C++ [Stroustrup94]. Indeed, Stroustrup notes [Stroustrup13] that "… my ideal when I started on C++ was 'Algol 68 with Classes' rather than 'C with Classes'."

Finally, no discussion of programming languages would be complete without an implementation of FizzBuzz, so I present a modest example in Listing 3. Note here that in Algol 68 the **AND** operator between two **BOOL** values does not short-circuit – that is, both **BOOL** terms are fully evaluated even if the first is found to yield **FALSE**. Many implementations provide an alternative short-circuiting form usually called **ANDF** or **ANDTH** (for 'and false' or 'and then'). ■

```
# Fizzbuzz in Algol68 #

print(( "Fizzbuzz from 1 to 100", newline ));

PROC fizz = (INT n) BOOL:
    IF n MOD 3 = 0 THEN print( "fizz" ); TRUE ELSE
FALSE FI;

PROC buzz = (INT n) BOOL:
    IF n MOD 5 = 0 THEN print( "buzz" ); TRUE ELSE
FALSE FI;

FOR i TO 100
DO

    IF NOT fizz( i ) AND NOT buzz( i )
    THEN
        print( whole( i, 0 ) )
    FI;
    newline( standout )

OD
```

Listing 3

References

[A68TOC] The ELLA and A68toC source is available from: https://cs.nyu.edu/courses/spring02/G22.3130-001/ella/

[GEORGE] Algol 68-R included with the GEORGE 3 Emulator on the BCS Software Preservation and Machine Emulation pages. http://sw.ccs.bcs.org/CCs/g3/index.html

[Henney18] Kevlin Henney. Procedural Programming: It's Back? It Never Went Away. From the 2018 ACCU Conference. https://www.youtube.com/watch?v=mrY6xrWp3Gs This talk has a good 10-minute account of Algol 68 from about 12 minutes in.

[Minority] Published in the *Algol Bulletin*, archived online at: http://archive.computerhistory.org/resources/text/algol/algol_bulletin/A31/P111.HTM

[Original] The original report can be found here, scanned to PDF (without OCR): http://web.eah-jena.de/~kleine/history/languages/Algol68-Report.pdf

[P0305R0] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0305r0.html

[Peck70] Peck, J.E.L., ed. (1970), *Proceedings of the IFIP working conference on ALGOL 68 Implementation*, Munich: North-Holland, ISBN 0-7204-2045-8. Available online at: http://www.softwarepreservation.org/projects/ALGOL/paper/ALGOL_68-Implementation.pdf

[Revised] *Revised Report on the Algorithmic Language Algol 68*. Edited by van Wijngaarden, et al. Springer-Verlag (1976). An HTML version is available at: https://jmvdveer.home.xs4all.nl/en.post.algol-68-revised-report.html

[SPS] The Software Preservation Society pages on Algol 68 http://www.softwarepreservation.org/projects/ALGOL/algol68impl/

[Stroustrup94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley (1994).

[Stroustrup13] Bjarne Stroustrup. *The C++ Programming Language*, 4th Edition. Addison Wesley (2013). Sec. 1.2.2.

[van der Meer16] Algol 68 Genie. Marcel van der Veer. https://jmvdveer.home.xs4all.nl/en.algol-68-genie.html

[Wikipedia-1] IFIP Working Group 2.1: https://en.wikipedia.org/wiki/IFIP_Working_Group_2.1

[Wikipedia-2] ALGOL 68-R: https://en.wikipedia.org/wiki/ALGOL_68-R

[Wikipedia-3]AlLGOL 68: https://en.wikipedia.org/wiki/ALGOL_68

Measuring Throughput and the Impact of Cache-line Awareness

How do you measure throughput? Richard Reich and Wesley Maness investigate suitable metrics.

In our previous article [Maness18] we collected measurements of latency which was later used in our analysis of the impact of cache-line aware data structures. In this paper, which can be thought of as a follow-up to our previous paper, we instead focus on throughput as a quantifiable metric. We will also be using newer hardware and the Centos 7.2 tool chain with a custom gcc 7.1. We will refer to some code examples, not shown here, that may require the reader to reference the code in the previous article, but we will attempt to minimize the need to reference, focusing as we investigate many ways to measure throughput and their observed behavioral changes. These changes will be measured in multiple ways to demonstrate the different aspects of throughput detailed below. We will also follow the same approach, which is to say we will first introduce our definitions, then take measurements, introduce aligned data structures (the same queue as before), and measure those observations. Keeping things simple for now so we can pin down the observations and the impact of cache-line aware data structures.

Before we get into the analysis of throughput, why does it matter? What does throughput tell us and what are the various scenarios in which we require a firm understanding of throughput and its implications? Throughput simply tells us how much work/data we can send or distribute given some constraints such as a network cable, pipe, or wireless signal. If we have an idea of how much work or data we can send to one location to another, we can have a better understanding of the upper limits of our network.

If we can postulate some upper bound limits of our throughput, we can better model how much work can be performed, given the flow of input into our received end points. Once we have a theoretical ceiling of the received data, we can then start to see where these questions become quite interesting in many domains. For example, within a defined time window: how many frames per second can we render on screen, how many updates can we compute, how much market data can we distribute, how many orders can we route to an algorithm or broker, how many messages can we send out to subscribers. These examples all require some basic understanding of throughput, how to measure it, and implications on data load.

Taking it one step further, once a firm understanding of the expected throughput is established the new insights will lend to more effective design, or even deployment of various components in a technical domain. If this is abstracted even further – we can start to explore networks and control flow problems using throughput as a weight on each node in a graph, thinking of routing traffic in real time, and other interesting areas to explore, but not here.

Definitions

For us to have some way of comparing results we need to provide clear definitions of throughput and discuss some issues surrounding measurement taking.

We offer two types of throughput for our analysis. The first one can be thought of as the classical definition of throughput. This is simply a measure of data of some period. The data can be structured in any form and we can slice and dice the time to any unit we deem necessary. Going forward in the paper we refer to this point of observation and or operations over time as spatial throughput(ST). Often the analogy that comes to mind for ST is network bandwidth. Our formal definition for ST will be defined as the number of work units (WU) over time *t*. We will measure a single WU as the number of cycles observed in the TSC register. All the code found for measuring ST is shown in Listing 1.

```
// one billion is once per second
// works_ is the number of work units
//   executed this observation period.
uint32_t bandwidth(uint32_t per = 1'000'000'000)
{
    // CPU is CPU speed in GHz
    // per is observation time scale units
    // end_ - start_ is the observation window.
    return (static_cast<float>((works_)*CPU*per) /
  (end_ - start_));
}
```

Listing 1

Note: to scale to a time quantum, divide ST by the desired time quantum and multiply by the CPU clock speed.

A second type of throughput we intend to observe in the paper is how much work can we do given new work to be done. This can be thought of as temporal throughput (TT). Often this is another way of looking at how frequently I am polling versus how frequently I am pulling data and processing that data. This can be thought of as saturation and often quoted as a percentage.

We will measure TT as two distinct types of value. TT ratio (TTR) will be defined as the number of WU divided by poll count and quoted as a percentage. A value of 1 would be full saturation. All the code found for measuring TTR is shown in Listing 2.

The second type of TT we will measure will be TT cycles (TTC). TTC will be defined as the total time of work divided by the total time of work plus the overhead. All the code found for measuring TTC is shown in Listing 3.

Richard Reich has 25 years of experience in software engineering ranging from digital image processing/image recognition in the 90s to low latency protocol development over CAN bus in early 2000s. Beginning in 2006, he entered the financial industry and since has developed seven low latency trading platforms and related systems. He can be reached at richard@rdrtech.com

Wesley Maness has been programming C++ for over 15 years, beginning with missile defense in Washington, D.C. and most recently for various hedge funds in New York City. He has been a member of the C++ Standards Committee and SG14 since 2015. He enjoys golf, table tennis, and writing in his spare time and can be reached at wesley.maness@aya.yale.edu.

We intentionally introduced false sharing into the simulated work as there are many cases, in a production system, that this is unavoidable.

```
// polls_ is the amount of time taken to poll for
work
uint16_t saturationRatio()
{
    if (polls_)
        return static_cast<uint16_t>(
(static_cast<float>(works_) / polls_));
    else
        return 0;
}
```

Listing 2

There are some observational complexities that surround TT data gathering especially compared to ST. In the TT case, we should really focus on using RDTSC as an instruction and not RDTSCP, as RDTSCP basically creates a code fence and can in many ways hamper or prevent hardware optimizations such as out of order execution and execution pipelining that would affect our cycles count. Due to the aggregate nature of measuring many WU in a particular time window the inaccuracies of RDTSC do not affect our measurements in a significant way. In our previous paper each WU was measured and recorded, necessary for the purpose of statistical analysis. When many WU are recorded as an aggregate quantification the inaccuracies are tiny compared to the impact of the serialized RDTSCP instruction.

Setup

ST setup

In our first experiment, we will focus only on measuring ST with and without cache-line awareness scaling from 1, 4, 8, and 16 threads, each thread will be pinned to a different core and producing work which will be in the simplified form of a 4-byte atomic increment. Shown on the right is the section of code (Listing 4) that performs this first setup.

```
// saturation_ is number of cycles spent working
// overhead_ is number of cycles spent polling
// for work
uint16_t saturationCycles()
{
    if (saturation_)
        return static_cast<uint16_t>(
          (static_cast<float>(saturation_) /
(saturation_
           + overhead_)));
    else
        return 0;
}
```

Listing 3

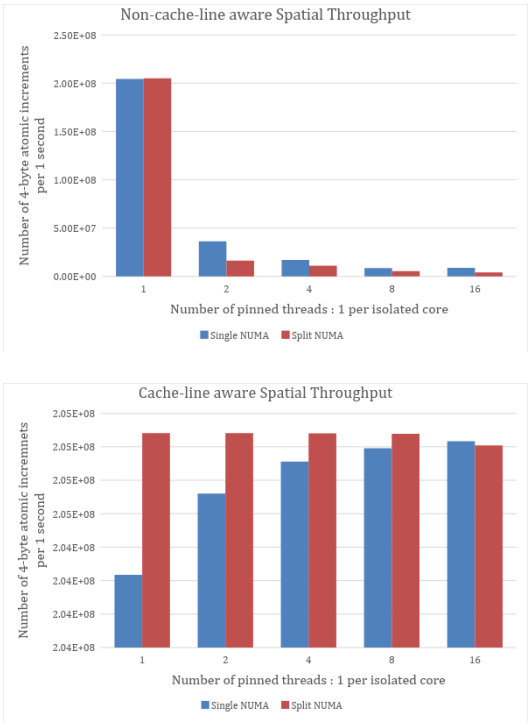


Figure 1

In Figure 1, we have captured average ST numbers for incrementing a single 4-byte atomic `int`. In those tests, we use thread numberings from 1 to 16 in powers of 2. In the split NUMA the threads are evenly distributed across two NUMA nodes. Each thread is pinned to a single core and no more than one thread per core.

TTR, TTC, and ST Setup

In our second experiment, we will measure TTR, TTC and ST. We will focus on non-cache-line awareness and cache-line awareness for various combinations of producers and consumers. The code we present below is in the same order in which they would be executed at the functional level. The first is the `simpleTest` method driven from main shown in Listing 5. The second unit of execution is the run method shown in Listing 6. Heavily utilized during the run method is our consumer method shown in Listing 7. Other code was deliberately left out but for anyone interested please contact the authors and we can provide access to a complete package. After everything has run and executed we have collected our results into two tables. Table 1 is a simple bandwidth analysis of the Boost MPMC queue for cache-line awareness and non-cache-line awareness. Table 2 is where we provide a simulated work load of 3000 cycles and 10 iterations per cycle.

```
template <int X>
struct DataTest
{
  alignas (X) std::atomic<uint32_t> d{0};
};

void CLTest ( std::atomic<uint32_t>& d )
{
  for (;;)
  {
    ++d;
  }
}

template <int Align>
int simpleTest (const std::string& pc)
{
  using DataType_t = DataTest<Align>;
  DataType_t data[128];
  std::vector<std::unique_ptr<std::thread>>
    threads;
  threads.reserve(pc.length());

  int32_t core{0};
  int32_t idx{0};
  for(auto i : pc)
  {
      if (i == 'p')
      {

threads.push_back(std::make_unique<std::thread>
            (CLTest, std::ref(data[idx].d)));
          setAffinity(*threads.rbegin(), core);
          ++idx;
      }
      ++core;
  }

  auto counters =
std::make_unique<uint32_t[]>(idx);

  for (;;)
  {
    sleep(1);
    int32_t i{0};
    for (i = 0; i < idx; ++i)
    {
       counters[i] = data[i].d.load();
       data[i].d.store(0);
    }


      uint64_t total{0};
      for (int i = 0; i < idx; ++i)
      {
        std::cout << "d" << i << " = " <<
          counters[i] << ", ";
        total+= counters[i];
      }
      std::cout << ", total = " << total
        << ", avg = " << static_cast<float>
        (total) / idx << std::endl;
      total = 0;
  }
  return 0;
}
```

Listing 4

```
template <int Align>
int simpleTest (const std::string& pc)
{
  using DataType_t = DataTest<Align>;
  DataType_t data[128];

  std::cout << "Sizeof data = " << sizeof(data)
            << std::endl;

  std::vector<std::unique_ptr<std::thread>>
    threads;

  threads.reserve(pc.length());

  int32_t core{0};
  int32_t idx{0};
  for(auto i : pc)
  {
    if (i == 'p')
    {
      threads.push_back(std::make_unique
          <std::thread>(CLTest,
            std::ref(data[idx].d)));

      setAffinity(*threads.rbegin(), core);
      ++idx;
    }
    ++core;
  }

  auto counters =
    std::make_unique<uint32_t[]>(idx);

  for (;;)
  {
    sleep(1);
    int32_t i{0};
    for (i = 0; i < idx; ++i)
    {
      counters[i] = data[i].d.load();
      data[i].d.store(0);
    }

    uint64_t total{0};
    for (int i = 0; i < idx; ++i)
    {
      std::cout << "d" << i << " = "
                << counters[i] << ", ";
      total+= counters[i];
    }
    std::cout << ", total = " << total
              << ", avg = "
              << static_cast<float>(total) /
                    idx
              << std::endl;
    total = 0;
  }
  return 0;
}
```

Listing 5

We intentionally introduced false sharing into the simulated work as there are many cases, in a production system, that this is unavoidable. Without false sharing, the results would have shown much more sensational numbers but, in our opinion, would have misrepresented a significant amount of real world application. In the event a system is fortunate enough to eliminate all false sharing, a greater boost in performance will be realized due to cache-line awareness. If the reader is curious, they can

```
template<typename T,template<class...>typename Q>
void run ( const std::string& pc, uint64_t
workCycles, uint32_t workIterations )
{
  using WD_t = WorkData<alignof(T)>;
  // shared data amongst producers
  WD_t wd;

  std::cout<< "Alignment of T "
    << alignof(T)
    << std::endl;

  std::cout   << "Size of CycleTracker "
    << sizeof(CycleTracker)
    << std::endl;

  std::cout   << "Size of ResultsSync "
    << sizeof(ResultsSync)
    << std::endl;

  std::vector<std::unique_ptr<std::thread>>
    threads;

  threads.reserve(pc.length());
  // reserve enough of each for total number
  // possible threads
  // They will be packed together causing false
  // sharing unless aligned to the cache-line.
  auto rs =
    std::make_unique<Alignment<ResultsSync,
      alignof(T)>[]>(pc.length());
  auto ct =
      std::make_unique<Alignment<CycleTracker,
      alignof(T)>[]>(pc.length());

  Q<T> q(128);

  // need to make this a command line option
  // and do proper balancing between
  // consumers and producers
  uint32_t iterations = 1000000000;

  uint32_t core{0};
  uint32_t index{0};
  for (auto i : pc)
  {
    if (i == 'p')
    {
      threads.push_back(
          std::make_unique<std::thread>
          (producer<T,Q<T>>
           , &q
           , iterations
           , workCycles
           , workIterations));
      setAffinity(*threads.rbegin(), core);
    }
    else if (i == 'c')
    {
      threads.push_back(
          std::make_unique<std::thread>
          (consumer<T,Q<T>,WD_t>
           , &q
           , iterations
           , std::ref(rs[index].get())
           , std::ref(ct[index].get())
           , std::ref(wd)));
      ++index;
```

Listing 6

```
      // adjust for physical cpu/core layout
      setAffinity(*threads.rbegin(), core);
    }
    else if (i == 'w')
    {
      threads.push_back(
          std::make_unique<std::thread>
          (worker<WD_t>,
           std::ref(wd)));

      // adjust for physical cpu/core layout
      setAffinity(*threads.rbegin(), core);
    }

    ++core;
  }

  Thread::g_cstart.store(true);
  usleep(500000);
  Thread::g_pstart.store(true);

  auto results =
    std::make_unique<Results[]>(index);

  for (;;)
  {
    sleep(1);
    for ( uint32_t i = 0; i < index; ++i)
      results[i] =
        ct[i].get().getResults(rs[i].get(), true);

    uint64_t totalBandwidth{0};
    std::cout << "----" << std::endl;
    std::cout << "workCycles = " << workCycles
              << std::endl;
    std::cout << "workIterations = "
              << workIterations
              << std::endl;

    for ( uint32_t i = 0; i < index; ++i)
    {
      // T1 Begin
      std::cout << "Temporal: saturation [Cycles]"
        "= " << results[i].saturationCycles()
        << std::endl;
      std::cout << "Temporal: saturation [Ratio]"
        " =  " << results[i].saturationRatio()
        << std::endl;
      std::cout << "Spatial: Bandwidth [work/sec]"
        " = " << results[i].bandwidth()
        << std::endl;
      totalBandwidth += results[i].bandwidth();
      // T1 End
    }
    std::cout << "Total Bandwidth = "
              << totalBandwidth << std::endl;
    std::cout << "----\n" << std::endl;
  }

  for (auto& i : threads)
  {
    i->join();
  }
}
```

Listing 6 (cont'd)

```
template <typename T, typename Q, typename WD>
void consumer(Q* q, int32_t iterations,
    ResultsSync& rs, CycleTracker& ct, WD& wd)
{
  while (Thread::g_cstart.load() == false) {}

  T d;
  uint64_t start;
  bool work = false;

  ct.start();
  for (;;)
  {

    CycleTracker::CheckPoint cp(ct, rs);
    // roll into CheckPoint constructor?
    cp.markOne();

    start = getcc_ns();
    work = q->pop(d);
    if (!work)
    {
      cp.markTwo();
      __builtin_ia32_pause();
      continue;
    }
    cp.markTwo();

    // simulate work:
    // When cache aligned WD occupies 2
    // cache lines
    // removing the false sharing from the read
    for (uint32_t k = 0;
        k < d.get().workIterations; k++)
    {
      // get a local copy of data
      WD local_wd(wd);
      // simulate work on data
      while (getcc_ns() - start <
          d.get().workCycles){}

      for (uint32_t it = 0;
          it < WriteWorkData::Elem; ++it)
      {
        // simulate writing results
        // This is false sharing, which
        // cannot be avoided at times
        // The intent is to show the separation of
        // the read and write data
        wd.wwd.data[it]++;
      }
    }
    cp.markThree();
  }
}
```

Listing 7

	ST	TTC	TTR
Cache-line aware	1350832.5	0.0098	0.63525
Non cache-line aware	1301837.5	0.01005	0.7081

Table 1

	ST	TTC	TTR
Cache-line aware	615981	0.80315	1
Non cache-line aware	275164	0.8685	1

Table 2

obtain a copy of the code and experiment by removing the false sharing in the consumer thread.

Analysis

The results of the bandwidth test are shown in Table 1. Here we can see the approximate savings/gains moving from a non-cache-line aware to cache-line aware queue are around 2.48% for TTC, 10.29% TTR, and 3.76% ST respectively. The results of the simulated workload are shown in Table 2. The approximate savings/gains from non-cache-line aware to cache-line aware are around 7.52% for TTC, 0% for TTR, and 123.86% ST respectively. The largest overall improvement we noticed was when workload decreased the savings for ST improved the greatest when adjusting for cache-line awareness. [the more work load there is per WU, i.e. more cycles, the less of an impact cache line alignment has on throughput.] The other measurements of TTC and TTR generally had overall improvements but the savings or gains in this case were not as great as they were for ST.

Conclusion and future direction

Based on our limited observations, we can clearly see that the gains achieved for cache-line awareness in our queues were greatest for ST data points under simulated workload. This suggests that there could be some interesting relationships between workload scaling and ST for cache-line aware MPMC queues. These relationships could be explored further to attempt to understand the savings gained given the workload and attempt to better identify potential bottlenecks in the utilized MPMC queue. ■

Notes

GCC 7.1 was used with the flags -std+c++17 -Wall -O3. Boost 1.64 was used on a dual socket 18 core (36 total) Intel (R) Gold 6154 CPU @ 3GHZ. Hyper-threading was not enabled. CPU isolation is in place. Special thanks to Frances Butontempo for her early feedback and suggestions for improvement. Complete source code packages utilized in this article can be made available by contacting the authors directly.

References

[Mannes18] Wesley Maness and Richard Reich (2018) 'Cache-Line Aware Data Structures' in *Overload* 146, published August 2018, available online at: https://accu.org/index.php/journals/2535

Afterwood

Renovation or redecorating throws up decisions. Chris Oldwood reminds us to make sympathetic changes.

Starting work on a mature codebase is always an interesting prospect. In some ways it's like moving into a fully furnished home. As you wander around your new abode you'll wince at the choice of wallpaper, soft furnishings, carpets and cabinets. As time passes those more superficial distractions will fade away and be replaced by the more niggly issues which actually affect your day-to-day life like the temperamental door lock, the broken light over the sink, and the low ceiling in the cellar. While it's somewhat disingenuous to liken all mature codebases to the derelict house renovated by Mary Bailey (Donna Reed) in *It's a Wonderful Life*, the scenes of frustration where George Bailey (James Stewart) accidentally pulls the finial off the top of the newel post every time he goes upstairs has a familiar ring to it.

It is all too easy when we come across some code we don't like to be highly dismissive and even vocalise our displeasure. I don't know about builders, plumbers and electricians in other countries but here in the UK they are renowned for looking at somebody else's handiwork, shaking their head and then telling you they wouldn't have done it like that. I guess that somehow that's meant to make you feel more confident in their abilities because *they* can spot bad workmanship in others and by logical deduction theirs must be better?

The late, great Jerry Weinberg has some advice that's worth bearing in mind when approaching a codebase that's been around the block: "everything got the way it is one logical step at a time". The evolution of the system happened under many constraints that we will not be aware of and it behoves us to be respectful of the choices that were made even if we don't personally agree with them. Similarly 'The Prime Directive' which is commonly read out before a retrospective reminds us that we consider that people make choices which are a product of the environment they work in rather than through any personal weaknesses – the environment should not set you up to fail.

It's easy for us when attempting to reshape a system towards new ends to become frustrated with the code as it stands and to lash out at the misgivings of our predecessors, even if that person is us. This negativity, which perhaps feels justified, does little to help the situation and if left unchecked begins to affect those around us. That famous poster from the Second World War tells us "careless talk costs lives" and badmouthing other people's work when those involved may easily be in earshot or in the social network of those around us does little to make them feel any better about the choices they may have had to take. It always starts very innocently, often just a bit of 'banter', but slowly that negativity can become the norm. Sarcasm is a dangerous tool to wield in an open plan office where information disseminates by osmosis pieced together from fragments of other conversations within earshot.

But it's our house, right? Surely that gives us permission to shape it as we see fit? Yes, but only insofar as we shape what's necessary to allow us to overcome any hurdles placed in our way as a consequence of earlier choices. You do not *automatically* gain permission to simply remove the woodchip wallpaper because your personal preference is for painted walls. Aside from the waste of time and money it is disrespectful and does little to engender respect for yourself and your choices as your successors reflect on your actions in the future. Collaboration is not simply about communication with those present in the here-and-now but also with those who went before us and will be there to take over once we have moved on. Entropy always wins and therefore we need to be aware that those who come after us will likely be dealing with more complexity than we ever had to so let's not make it any harder by throwing unrelated noise into the mix when it's not an impediment to our current progress.

When a spot of refactoring is beneficial it can be a struggle to remain focused when we see paint starting to peel off the walls, the tap in the bathroom dripping and know the kitchen door needs a few drops of oil to stop it squeaking. None of these imperfections significantly hinder our day-to-day living but are just symptoms of age, and codebases accrete similar blemishes over time as the people, tools, and practices change – unsightly does not imply a liability.

Lucius Cary (2nd Viscount Falkland, according to Wikipedia) once said "Where it is not necessary to change, it is necessary not to change." Taken too literally you might not even bother to consider the practice of refactoring or cleaning up code at all, let alone use it as a guide to reign in any overzealous changes that appear to involve tidying up every piece of unrelated code you passed through when debugging. The problem is that change is occurring all around us and therefore it may become necessary to change, despite not changing ourselves. While software doesn't rot in the physical sense it could be said to deteriorate when inconsistencies in the idioms, layout, domain language, design, etc. begin to affect comprehension, at which point that may well become an impediment. Sometimes change is so slow that it is imperceptible at first, like an unused house decaying. Then, as the Broken Windows theory posits, a tipping point is reached where upon that dwelling rapidly deteriorates into the dilapidated form where Mary and George Bailey spend their honeymoon and eventually set up home.

Both refactoring and the (Boy) Scout Rule can, and often are, held up as a cause under which it's okay to make sweeping changes in a codebase because they are 'best practice' and it makes the code 'better'. Like many programmers I have my own personal crusades such as the banishment of 'get' in favour of more expressive verbs such as make, format or derive, and wider acceptance of domain types instead of an obsession with primitives. But they are just that – personal preferences – it is entirely possible to write a correctly functioning system without them.

Making changes in sympathy with how a system has already evolved, and will continue to evolve is hard. The temptation to 'fix' everything is very real but first we need to get straight in our heads what is actually broken or untenable and what is simply unappealing. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel code.

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner

To find out more about Intel products please contact us:
020 8733 7101 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE