overload 143

FEBRUARY 2018 £3

Functional Error-Handling with Optional and Expected

Exceptions should be exceptional. We see how to deal with disappointments using C++17's optional.

An MWSR Queue with Minimalist Locking

An implementation of a multiple-writer single-reader queue

Testing: Choose The Right Level

Keep your focus right to help make testing easy

CTAD – What Is This New Acronym All About?

C++17's class template argument deduction

C++ with Meta-classes?

Comparing meta-classes to C++ developments of the 1990s

A Wider Vision of Software Development

The final part of the Organising Principles series

Practical Scale Testing

Scrutinising methods for testing and scalability

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 143

January 2018

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Balog Pal
pasa@lib.h

Ben Curry
b.d.curry@gmail.com

Paul Johnson
paulf.johnson@gmail.com

Klitos Kyriacou
klitos.kyriacou@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Philipp Schwaha
<philipp@schwaha.net>

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 144 should be submitted by 1st March 2018 and those for Overload 145 by 1st May 2018.

Overload is a publication of the ACCU

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

4 A Wider Vision of Software Development
Charles Tolman brings his Organising Principles series to a close.

6 An MWSR Queue with Minimalist Locking
Sergey Ignatchenko describes his implementation of a multiple writer single reader queue.

10 Testing: Choose the Right Level
Andy Balaam considers levels to keep your focus test just right.

13 CTAD – What Is This New Acronym All About?
Roger Orr elucidates the class template argument deduction C++17 feature.

16 C++ With Meta-classes
Francis Glassborow compares meta-classes to developments of C++ in the 1990s.

18 Practical Scale Testing
Arun Saha scrutinises methods for testing and scalability.

21 Functional Error-handling with Optional and Expected
Simon Brand shows us how to deal with disappointments using C++17's optional

20 Afterwood
Chris Oldwood recounts his attempts to write a calendar in an interview.

The ACCU

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

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

Copyrights and Trade Marks

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

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

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

of future performance. You need to try and see if experiments match your predictions.

You can drop the attempt to fit data statistically, and try some machine learning techniques. Many of these don't require rigorous mathematics to draw conclusions. This leads to derision from tormentors, disparaging machine learning as curve fitting. Quora [Quora] has a short discussion on this subject. Fitting a curve to data can be interesting. In fact, if you chuck out outliers you can get a better curve. Or at least a smoother one, with fewer kinks and a far simpler equation. How do you decide which points to ignore? Sometimes a decision is made up front. LIBOR, the London intrabank offered rate, averages the rate banks are offered when they want to borrow money. The data submitted covers a few currencies over several time periods. This is used to calculate a benchmark of the interest rate on loans. The top and bottom quartiles are discarded, so the figures are based on the middle numbers. Why? Possibly an assumption that the extremes were anomalies or even lies. It is very hard to find out now, because most online resources concerning LIBOR point to the various fixing scandals [for example, Wikipedia-c]. The London Review of Books wrote an interesting overview, in the midst of the financial crash in 2008 [LRB]. The process might involve an email of the numbers, mid-morning, but could be followed up with a phone call if the email didn't arrive or the numbers looked wrong. Very hi-tech! There are two significant points here. First, the data was based on what bankers claimed they could borrow money at. Since the various scandals, there have been moves towards using actual transactions to find a benchmark. Using actual data rather than hearsay can give you different outcomes. Second, the extremes: the top and bottom quarter of the numbers are ditched, on the grounds they are or could be outliers.

If you have data with a Normal distribution, you can work out how likely a value is. If your data has a mean of zero and variance of one, a reading of 0.1 is reasonable. A reading of 100.1 is less likely. Much less likely. That, of course, doesn't mean it's wrong. You can't reject something on the grounds it's a bit of a surprise. Sometimes you can establish a reason for the unexpected observations. Perhaps you have a measuring instrument that needs recalibrating. Sometimes you might find your assumption of a normal distribution was incorrect. Sometimes you might need to introduce a hierarchical model, essentially introducing if-then-else to deal with some specific circumstances. Nassim Taleb has written about surprising events that have major impacts. He calls these Black Swan events. He states three criteria for such events:

1. They are a surprise
2. They have a major impact
3. Once it's happened, various theories are concocted to show it was bound to happen and you can predict when it will happen again

Taleb described these as "stemming from the use of degenerate metaprobability" [Wikipedia-d]. We are driven to explain something, assuming the world follows predictable patterns that make sense. Taleb emphasises that some events are unprecedented, and therefore cannot be reasoned about.

Some one-off events have catastrophic effects. In contrast, some outcomes may be surprising, but good. If you just happen to notice a config change which explains why your production system is broken, you might be able to fix things quickly. How likely were you to spot the problem? It depends. It might be unlikely, especially if the config was for a different system entirely, but for some spaghetti entanglement reasons it had a knock-on effect. I'm sure you have your own examples. A miracle? The philosopher, David Hume attempted to define a miracle in *An Enquiry Concerning Human Understanding* (originally published 1748, and republished many times since). He defined a miracle as a violation of a law of nature, by the intervention of a Deity. He regarded this definition as a way to stop "all kinds of superstitious delusion" [Wikipedia-e]. The thinking goes that a miracle must be a singular event, and so weight of evidence shows any witness of such an event is unreliable. Critics suggest he has got his maths wrong. I'll leave you to decide whether miracles are possible. Whatever conclusions you draw, some events are ordinary while some are extraordinary.

As a programmer, you are extraordinary. I struggled to find reliable data to back this up. Computerworld [Computerworld13] claimed in 2013 there were about 18.2 million developers worldwide. Of seven billion or so people, that's not many. It's not clear how you count programmers. By profession? Incidentally making spreadsheets from time to time? CAD designers? Teenagers teaching themselves to code, alone at home? Stackoverflow runs a survey annually to investigate developer demographics [Stackoverflow17]. However many there are, it seems most have at least 20 years' experience, are white and are male. If you are black, female or new to programming you are extraordinary. Nonbinary, genderfluid, genderqueer, trans, agender, etc. programmers are extraordinary too. If you are a white male programming with twenty or more years' experience, you are also extraordinary. Being described as an anomaly or outlier sounds negative. Who wants to be normal though? Everyone is unique. That's a good thing. If you are derided for being a geek, fear not. If you are frequently told, "Normal people don't do that", remember you are extraordinary. Find some like-minded people to chat to or work with, or join an organisation of like-minded programmers such as the ACCU, and enjoy yourself. If you then end up feeling like a fraud because everyone around you has more experience than you, or seems to know more details than you, don't let the imposter syndrome take over. Even if you don't believe in magic, or miracles or guardian angels, find a way to say "Expecto Pratronum!" at the Dementors trying to undermine you.

References

[Buontempo12] *Overload* 110, Aug 2012, 'Allow me to introduce myself', https://accu.org/index.php/journals/1904

[Computerworld13] https://www.computerworld.com/article/2483690/it-careers/india-to-overtake-u-s--on-number-of-developers-by-2017.html

[etymonline] https://www.etymonline.com/word/pattern

[LRB] https://www.lrb.co.uk/v30/n18/donald-mackenzie/whats-in-a-number

[Pottermore] https://www.pottermore.com/features/what-is-a-patronus

[Quora] https://www.quora.com/Is-Machine-Learning-just-glorified-curve-fitting

[Shelley17] *Frankenstein: Annotated for Scientists, Engineers, and Creators of All Kinds*, Shelley, Guston, Finn, Robert, Robinson. June 2017, MIT Press.

[Stackoverflow17] https://insights.stackoverflow.com/survey/2017

[Wikipedia-a] https://en.wikipedia.org/wiki/Hapax_legomenon

[Wikipedia-b] https://en.wikipedia.org/wiki/True_name]

[Wikipedia-c] https://en.wikipedia.org/wiki/Libor

[Wikipedia-d] https://en.wikipedia.org/wiki/Black_swan_theory

[Wikipedia-e] https://en.wikipedia.org/wiki/Of_Miracles

A Wider Vision of Software Development

Is code a hopeful arrangement of bytes? Charles Tolman brings his Organising Principles series to a close.

In this concluding article, I will explore the idea of the 'Organising Principle' further, relating it to the patterns work of Christopher Alexander and, given the relevance of perceiving such principles to software development, I suggest how we can improve our perception of such principles.

Be aware that it is possible that trying to explain the idea of an Organising Principle more clearly is a fool's errand since its reality cannot be fixed. As I mentioned in the previous article, If you fix it: You Haven't Got It. It is a far more experiential (or phenomenological) concept.

However, Organising Principles have a number of characteristics. They:

- Embody a living wholeness.
- Have a high degree of ambiguity.
- Are never static.
- Lie behind the Parts of a Whole.

I explicitly mention the separation of Whole and Parts here since it is a key aspect of any competent approach to software design. For the programmer this begs the question of how well he or she understands the problem in terms of its wholeness and yet also sees how the parts need to work therein.

Experienced people in a given domain will have a sense of the whole and yet will be able to identify the risk points in the parts. They can simultaneously see the whole picture and know the essence of what needs to happen in the parts. This is what you pay for when you employ an expert. For example, in the design example in the previous article, an expert will know to check your allocation strategies and if they see out of order allocations anywhere will be able to connect this to the failure of the system as a whole.

Alexander & patterns revisited

At this point it is useful to look at some of Christopher Alexander's ideas [Wikipedia-a] about the perception of beauty that links to what I have been saying about the idea of Cognitive Feeling.

Alexander started with defining a Pattern Language [PatternLanguage] to help foster good architectural design – what he called Living Structure. This metamorphosed into his masterwork, The Nature of Order [NatureOfOrder] where he tried to get a better understanding of why we find certain structures beautiful.

In the *Nature of Order*, Volume 1 Chapter 5, he identified the following 15 properties of Living Structure:

- Levels of scale
- Contrast
- Strong centres
- Boundaries
- Alternating repetitions
- Positive space
- Good shape
- Local symmetries
- Deep interlock and ambiguity
- Gradients
- Roughness
- Echoes
- The void
- Simplicity and inner calm
- Not-separateness

Charles Tolman earned a degree in Electronic Engineering in the 70s, and then moved into software; progressing through assembler to Pascal, Eiffel and eventually C++. He's now involved in large scale C++ development in the CAE domain. Having seen many silver bullets come and go, his interest is in a wider vision of programmer development that encompasses more than purely technical competence. You can contact him at ct@acm.org

If we look at this solely as a list of items, it can be difficult to understand how they may be useful in design, apart from using them as heuristic guidelines. Though useful, if we consider them in the light of the dynamic concept of the Organising Principle, they make a lot more sense.

A pointer to why this may be so is in Alexander's use of the word: Living. *Livingness* implies ambiguity, and therefore these 15 properties can also be seen as Organising Principles. Thus when we try and fix them in order to come to a better understanding, we will only be seeing one way of looking at each principle and by definition will have come away from the actual thing itself.

But can we better relate such principles to software development?

Requirements implicitly contain Organising Principles

Embedded within a set of requirements is something that we need to find and embody in our implementation. As with high-level schematics, there is a high degree of ambiguity surrounding this 'something' and I can usually hand wave to my heart's content about structure and architecture, but it also needs to be connected to the details of an actual implementation as I tried to do in the previous article.

Generally requirements will implicitly, not explicitly, embody dynamic specifications of what needs to happen. The conversations between users and developers, properly managed, are a key activity. Much of my job when I talk to any user is to try and understand:

- What they want to do.
- What I think can be built.
- How they can be brought together.

As a developer, I need to understand what core dynamic principles are embedded in the requirements. By definition if you are doing a useful piece of software the user won't know what they actually *need*. They will be able to talk about what they *want*, but that is most likely going to be based on extrapolations from their present experience. Any conversation between user and developer will generate new knowledge about future experience, so my job as a developer is to help us both map out this area of new knowledge and its attendant Organising Principles.

Architecture references the Organising Principle

If we take the point that an Organising Principle is not a fixed, discrete idea then we can see that it is much like comparing a film of an activity

with its reality. The film is just a set of discrete frames, but the reality is continuous.

In the same way, the various possible architectures and implementations are different views of a specific Organising Principle. This is difficult to grasp and needs a far more mobile thinking than we normally use and is the core of why good software development is so difficult. Certainly design patterns have helped but we need to go further and understand that the development of personal perception skills is more useful than coming up with abstract lists. I believe this is why it can take 15 years to develop such competence.

Code is the precipitate of the Organising Principle

If we manage to perceive this dynamic principle, referencing it by designing a software architecture that we embody into a specific code implementation, we are embarking on a process of precipitation. If we don't do this then – indeed – as the original title of my ACCU2016 talk suggested, we only have a 'Hopeful Arrangement of Bytes' without any coherent design.

Sometimes playing around with code from the 'bottom up' may be a valid thing to do as a piece of 'software research' just to see how the code works out and will be especially relevant if there is a blank sheet, or 'green field' site where you have to start somewhere. In this case you need to identify the highest risk requirements and just implement some proof of concept to check your understanding.

A point worth noting here is that it will not be ideal for a novice or journeyman programmer to follow the example of an experienced one. Be warned: It is possible that a master programmer can work at the keyboard and seemingly design as they go, creating from internalized mature ideas. Yet this is something that is definitely not recommended for inexperienced developers. They will need to spend much more time maturing and externalizing their ideas first before those ideas become part of their 'cognitive muscle memory' from which they can work direct to the keyboard. It is to be hoped that the experienced developer will realise earlier when they need to step away from the keyboard.

Perceiving Organising Principles

The above comments are all very well but just how can we develop such a 'living and mobile' thinking perception? Unfortunately as programmers we are at a disadvantage.

We work in a domain where much of our thinking needs to be fixed into a rule-based structure in order to imagine how a computer program will function. This can have unwanted side effects of making it difficult to think in a mobile, living way. Hence Ted Nelson's [Wikipedia-b] rant in his book *Geek's Bearing Gifts* about techies only being able to think in terms of hierarchies.

If we personally want to develop this other way of seeing, we need to engage in some activities that foster such a mode of cognition. Perceiving livingness, almost by definition, requires that we need to handle ambiguity. This is what is required when we are working in the 'gap', or whenever we are dealing with human situations. Logical thinking can cope with known and static issues, but as programmers we need to be very aware of the boundaries of our knowledge, more so than the lay person due to the inherent fixity of the domain of computer programming.

My thesis is that in order to develop a mobile dynamic cognition that can better perceive Organising Principles, we need to take up some artistic pursuit in a disciplined and self-aware way since it is the artistic process that can allow us to move outside the boundaries of what we already now.

Disciplined artistic process

In developing a balanced perception though an artistic approach, do whatever appeals to you. For me I find painting and dance work well. An example of how the artistic process parallels software development can be seen in an early experience I had with painting.

The following image is a watercolour painting of my daughter done at an early stage of my painting hobby (so please be gentle with any criticism!).

As one of my first forays into the painting world and like the good novice artist I was, I decided to draw the picture first, using a photograph as a reference. It took me 4 hours!

The first effort took 2 hours. The next took 1 hour and the last two each took half an hour. I had intended the final result to be the basis for the final painting. But being the worried novice that I was, all too aware of my lack of experience, I decided to perform a 'colour check' painting freehand, away from the drawing, before doing the final version. To my complete surprise this became the final painting I have shown here. I found that afterwards when I tried to paint into the final drawing it did not have the same life as the freehand painting.

This is an example of the difference between the 'master' freehand approach as compared to the 'journeyman' drawn approach. Of course I do not consider myself to be a master painter, but this example illustrates the perceptual self-developmental dynamic inherent in the artistic process.

We can also see here the need to do the foundational, 'analytic' work, in this case the drawing; followed by the 'gap' of putting the drawing away and using the developed freehand skill to come up with the 'solution idea'.

A final, and slightly frivolous, example of an artistic pursuit is that of the improvised dance form that is Argentine Tango, notable for the response it elicited from John Lakos in my original talk where he asked me to teach him some moves! I teach and dance Tango as a hobby and this particular dance form is strongly founded on being far more conscious about the primary human activity of walking, the signature movement in Tango. (For those interested, see my post on dance as True Movement. [Tolman16])

Here there is a need for structure, and a mobile process of interpretation and improvisation, both founded on a disciplined form of the dance. It can take years to learn how to 'walk' again but if followed in a disciplined manner can lead to sublime experiences of artistic 'Living Structure' as the 'team' of two people dance from a common centre of balance.

The author giving John Lakos an impromptu Tango lesson at ACCU2016. Photograph courtesy of Mogens Hansen.

A wider vision

In conclusion, I hope I have conveyed the implicit yet substantial link between art and technology. My wish is that it will enable us to widen our vision of software development, and help us realise that we cannot separate it from perceptual development, with its attendant need for a balanced development of the Self. ■

References

[NatureOfOrder] https://www.natureoforder.com

[PatternLanguage] https://www.patternlanguage.com

[Tolman16] https://charlestolman.com/2016/07/18/tango-thoughts-true-movement/

[Wikipedia-a] https://en.wikipedia.org/wiki/Christopher_Alexander

[Wikipedia-b] https://en.wikipedia.org/wiki/Ted_Nelson

An MWSR Queue with Minimalist Locking

Multithreaded queues come in many flavours. Sergey Ignatchenko describes his implementation of a multiple writer single reader queue.

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.

In [NoBugs17], we discussed the theory behind using CAS (Re)Actors to build multithreaded non-blocking primitives. A very brief recap:

- Whenever we want non-blocking processing, we ***have*** to use CAS (Compare and Swap) operations
- The idea of CAS (Re)Actors is to treat a CAS block (up to 128 bits in size on modern CPUs) as a state of the (Re)Actor; in other words, all we're doing within one specific (Re)Actor always fits into the following pattern:
  - We read the state
  - We modify it if necessary
  - We write it back
- In the context of CAS (Re)Actors, writing the state back is tricky: it requires CAS operations, which can fail due to some other thread interfering with us. If we fail, we simply drop the whole result and start anew; actually, this is a very typical pattern in CAS-based primitives. Another way to look at it is to consider it an incarnation of optimistic concurrency control.

As was mentioned in [NoBugs17], the benefit provided by CAS (Re)Actors is ***not*** about the sequence of CPU operations we're issuing; in theory, *exactly* the same thing can be produced without (Re)Actors at all. The key benefit is about *how we're thinking about our multithreaded primitive*, which tends to provide significant benefits in the complexity that we can handle. Today we'll demonstrate a practical use of CAS (Re)Actors using one very specific example.

The task at hand

In quite a few rather serious real-world interactive distributed systems (such as stock exchanges and games), I found myself in need of a really fast queue, which had to have to have the following characteristics:

- It should be an MWSR queue (allowing Multiple Writers, but only a Single Reader).
- It should have flow control. In other words – if queue writers are doing better than queue readers – the queue ***must not*** grow infinitely. Instead, at *some* point (when the queue grows over a certain pre-defined limit), our queue ***must*** start blocking writers.
- This means that our queue *cannot possibly be* 100% non-blocking. However, it should *only* be blocking. In other words, unless the queue is full, write should be non-blocking, and unless the queue is empty, read should be non-blocking too. *NB: as this is a performance requirement, complying with it a mere 99.9% of the time is good enough.*
- As a side – but occasionally rather important – property, the queue should be at least almost-fair. 100% fair queues (the ones which *guarantee* first come, first served behaviour) are rather difficult to achieve, but an *almost*-fair property (~='there won't be *too much* reordering in most of the scenarios') is much easier to obtain. In particular, if we're speaking about re-orderings caused by CAS failures (followed by an immediate retry along the lines of CAS (Re)Actors), then in practice, in 99.(9)% of the cases, unfairness will be limited to single-digit microseconds, which is OK for the vast majority of the use cases out there. In fact, most of the time, usually there are *much* more severe and unpredictable delays than those single-digit microseconds caused by CAS reorderings, so overall system behaviour will be pretty much indistinguishable from the behaviour of a perfectly fair system.

> **Optimistic Concurrency Control**
>
> OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. [Wikipedia]

As much as I was in need of such a queue, it turned out to be an *extremely* difficult task, so that I wasn't able to devise a system which avoids *all* the races. I did try to do it three or four times, but each time I found myself going into a vicious cycle of solving one race merely to create another one, and going into this 'robbing Peter to pay Paul' mode until I realized the futility of my efforts in that direction ☹. It was even worse as I was *sure* that a solution did exist – it is just that I wasn't able to find it.

Enter CAS (Re)Actors

After several attempts at it, writing this queue-with-flow-control became a kind of personal obsession of mine, so there is no surprise that last year I took another shot at it. And as by that time I was spending quite a bit of time formalizing and generalizing my experiences with (Re)Actors, the idea of CAS-size (Re)Actors fortunately came to my mind. Let's see how the CAS (Re)Actors did allow me to write that elusive MWSR queue with flow-control (we have to be sketchy here, but the whole supposedly-working implementation is available on Github [NoBugs18]).

Sergey Ignatchenko has 20+ years of industry experience, including being an architect of a stock exchange, and the sole architect of a game with hundreds of thousands of simultaneous players. He currently writes for a software blog (http://ithare.com), and translates from the Lapine language a 9-volume book series 'Development and Deployment of Multiplayer Online Games'. Sergey can be contacted at sergey.ignatchenko@ithare.com

our locking primitives must unlock properly regardless of potential races between a thread being locked and unlock()

(Re)Actors

Let's say that our queue is based on two CAS (Re)Actors, `EntranceReactor` and `ExitReactor`:

- The primary goal of `EntranceReactor` is to handle writers, providing slots in the queue and instructing them to block when no such slots are available.

  `EntranceReactor`'s state consists of:

  - `firstIDToWrite` – the first ID in the queue which is available for writing. *NB: we consider all IDs as non-wrappable because (as was discussed in [NoBugs17]) it would take hundreds of years to wrap-around a 64-bit counter). The position of the first ID in the queue buffer can be calculated as a simple* `ID%QueueSize`.
  - `lastIDToWrite` – the last ID which is available for writing (of course, we *do* want to allow more than one concurrent write to our fast-performing queue).
  - `lockedThreadCount` – the number of writers which are currently locked (because the queue is full).

  In accordance with CAS (Re)Actor doctrine, all operations over (Re)Actor are inherently atomic. For `EntranceReactor`, we define the following atomic operations:

  - `allocateNextID()` – allocates the next ID to the caller, *and* indicates whether the caller should lock for a while.
  - `unlock()` – indicates that the thread is unlocked.
  - `moveLastToWrite(lastW)` – here we're telling our `EntranceReactor` that reader has already read everything up to `lastW`, so that it can allow more writes. `moveLastToWrite()` returns whether we should unlock one or more writers (which is an expensive operation so we want to avoid it as long as possible).

- Our second (Re)Actor is `ExitReactor`, which handles our only reader; in particular, it maintains information about completed writes, so it can tell when information is available for reading.

  The state of `ExitReactor` consists of:

  - `firstIDToRead` – the first ID which is not read yet.
  - `completedWritesMask` – as there can be several concurrent writes, they can finish in an arbitrary order, so we have to account for them with a mask.
  - `readerIsLocked` – a simple flag, with semantics similar to `lockedThreadCount` (but as we have only one reader, a simple boolean flag is sufficient here).

  As for `ExitReactor`'s atomic operations, we define them as follows:

  - `writeCompleted(ID)` – indicates that the write of specific ID is completed.
  - `startRead()` – called by reader to start read, and either returns an ID to read, or indicates that we should lock instead.
  - `readCompleted(ID)` – indicates that the reader is done with reading; as it usually frees some space in the buffer, it returns a new ID where the write can be done.

For sizes of the fields, please refer to [NoBugs18]; however, it should be noted that, under the CAS (Re)Actors paradigm, we can easily use bit fields, so the only thing we care about is the *combined bit size* of the fields, which shouldn't exceed the magic number of 128 (that is, for modern x64 CPUs which support the CMPXCHG16B instruction – and that is pretty much any Intel/AMD CPU produced over last 10 years or so).

On the ABA problem

As is well-known in MT programming, and was briefly discussed in [NoBugs17], the so-called ABA problem is one of those things which can easily kill the correctness of a multithreaded primitive. However, it *seems* that our (Re)Actors are free from ABA-related issues; very briefly:

- As our IDs are monotonically increased and wraparound-free, all the writes which update *at least one* of the IDs are free from ABA problems (see also discussion on it in [NoBugs17]).
- The fields `lockedThreadCount` and `readerIsLocked` have semantics with the property 'it is ***only*** the current value which matters, and no history is relevant', which also means that their updates are ABA-problem free.
- This leaves `completedWritesMask` as the only potentially ABA-dangerous field. However, we can observe that if we consider the *tuple* (`firstIDToRead`,`completedWritesMask`) and take into account the logic behind these fields, *this whole tuple* is monotonically increased and wraparound-free; this, in turn, means that there is no potential for ABA problems here either <phew />.

Overall, from what I can see, our (Re)Actors are ABA-problem free; still, as an ABA problem is one of those bugs which can sit there for ages before manifesting itself, I would certainly appreciate somebody more skilled than me having another look at it.

Locking primitives

In addition to (Re)Actors, we have two locking primitives, both built more or less along the lines of Listing 1.

It is a rather simple (but quite interesting) primitive, with the idea being that whenever some of our (Re)Actors return, we should lock. The corresponding thread calls `lockAndWait()` and waits on a conditional variable until some other thread calls `unlock()`. It is important to note that our locking primitives ***must*** unlock properly regardless of potential races between a thread being locked and `unlock()`. In other words, it should work regardless of whether `unlock()` comes *before* or *after* the thread scheduled to be locked reaches `lockAndWait()`.

```
class LockedSingleThread {
private:
  int lockCount = 0;//MAY be both >0 and <0
  std::mutex mx;
  std::condition_variable cv;
public:
  void lockAndWait() {
    std::unique_lock<std::mutex> lock(mx);
    assert(lockCount == -1 || lockCount == 0);
    lockCount++;
    while (lockCount > 0) {
      cv.wait(lock);
    }
  }
  void unlock() {
    std::unique_lock<std::mutex> lock(mx);
    lockCount--;
    lock.unlock();
    cv.notify_one();
  }
};
```

Listing 1

Putting it all together

Having all four building blocks (two (Re)Actors and two locking primitives), we can write our **MWSRQueue** (see Listing 2).

As we can see, after we have defined our (Re)Actors (including their operations), the whole thing is fairly simple. Within our **push()** function, we merely:

- request an ID from **EntranceReactor** (and **lockAndWait()** if we're told to do so)
- write to the slot which corresponds to the ID. Note that while we're actually writing, we're *not holding any* locks, which is certainly a Good Thing™ concurrency-wise.
- Inform **ExitReactor** that the write is completed (so reader can start reading the ID we just wrote)

As for our **pop()** function, it is only marginally more complicated:

- We ask **ExitReactor** whether it is OK to read; if not, we're locking (and re-trying from scratch later)
- We read the data (again, at this point we're *not* holding any kind of locks(!)).
- We're telling our **ExitReactor** that we're done reading – and in response it may want to inform **EntranceReactor** that there is some room available. (This is implemented via a **newLastW** variable, but actually corresponds to sending a message – containing this one variable – from **ExitReactor** to **EntranceReactor**.)

That's it! We've got our **MWSRQueue**, and with all the desired properties too. In particular, it *is* an MWSR queue, it *does* provide flow control, it locks *only* when it is necessary (on the queue being empty or full), and it *is* almost-fair (as IDs are assigned in the very first call to the **allocateNextID()**, the most unfairness which can possibly happen is limited to CAS retries, which are never long in practice).

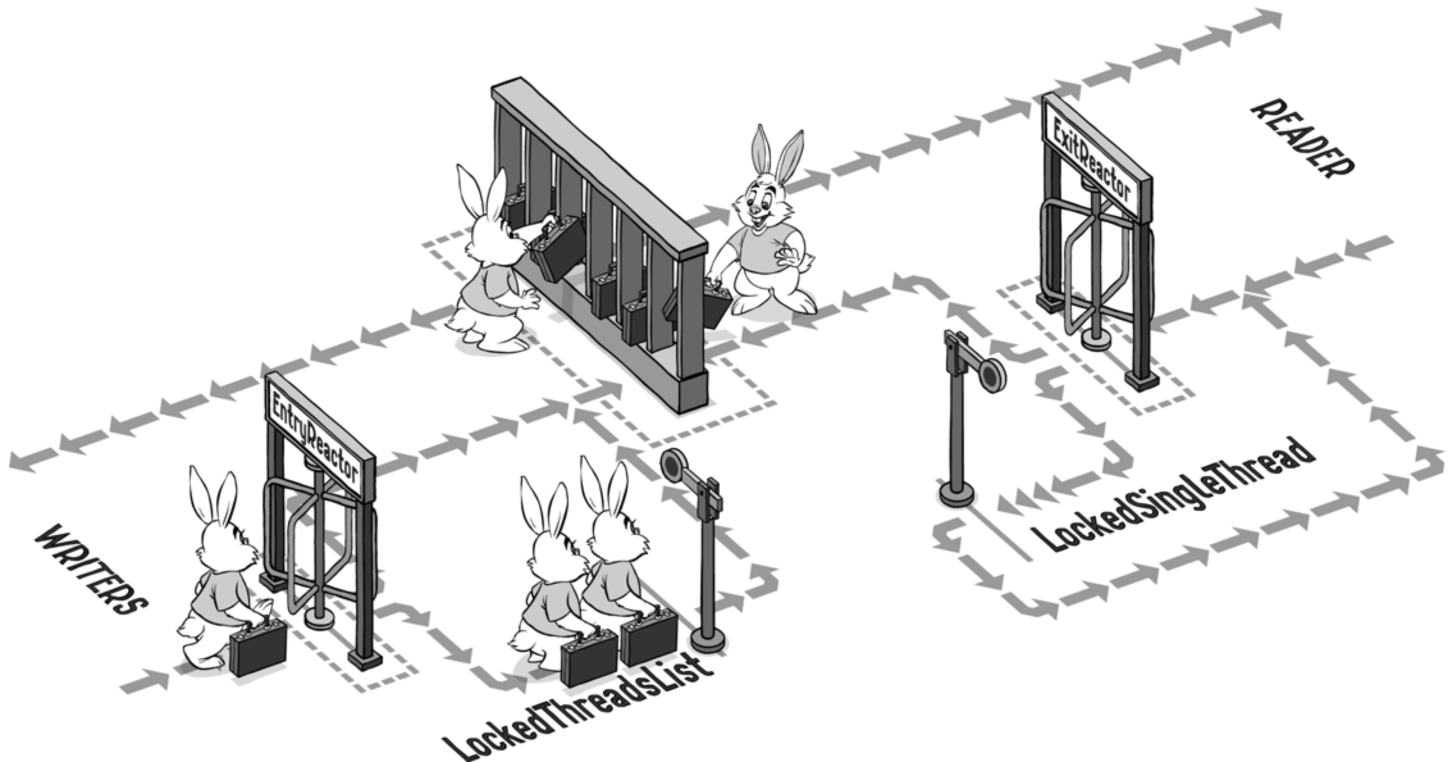
However, IMNSHO the most important property of the queue is that it was observed to be easily debuggable. After I finished writing the code (which is around 700 LoC of heavily-multithreaded code, and is next-to-impossible to test until the whole thing is completed) and ran the simplistic tests found in the `/test/` folder within [NoBugs18], there were, of course, bugs (like a dozen of them). And for multithreaded programs in general, debugging is a well-known nightmare (in particular, because (a) a bug manifests itself in a different place on different runs, and (b) adding tracing can *easily* change things too much so the bug won't

```
template<class QueueItem>
class MWSRQueue {
  static constexpr size_t QueueSize = 64;

private:
  QueueItem items[QueueSize];
  MT_CAS entrance;
  MWSRQueueFC_helpers::LockedThreadsList
    lockedWriters;
  MT_CAS exit;
  MWSRQueueFC_helpers::LockedSingleThread
    lockedReader;

public:
  MWSRQueue();
  void push(QueueItem&& item) {
    EntranceReactorHandle ent(entrance);
    std::pair<bool, uint64_t> ok_id =
      ent.allocateNextID();
    if (ok_id.first) {
      lockedWriters.lockAndWait(ok_id.second);
      ent.unlock();
    }
    size_t idx = index(ok_id.second);
    items[idx] = std::move(item);
    ExitReactorHandle ex(exit);
    bool unlock =
      ex.writeCompleted(ok_id.second);
    if (unlock)
      lockedReader.unlock();
  }
```

Listing 2

```
  QueueItem pop() {
    while (true) {
      ExitReactorHandle ex(exit);
      std::pair<size_t, uint64_t> sz_id =
        ex.startRead();
      size_t sz = sz_id.first;
      assert(sz <= QueueSize);
      if (!sz) {
        lockedReader.lockAndWait();
        // unlocking ex is done by
        // ex.writeCompleted()
        continue;//while(true)
      }
      uint64_t id = sz_id.second;
      size_t idx = index(id);
      QueueItem ret = std::move(items[idx]);
      uint64_t newLastW =
        ex.readCompleted(sz,id);
      EntranceReactorHandle ent(entrance);
      bool shouldUnlock =
        ent.moveLastToWrite(newLastW);
      if (shouldUnlock)
        lockedWriters.unlockAllUpTo(id +
          sz - 1 +QueueSize);
      return ret;
    } //while(true)
  }
private:
  size_t index(uint64_t i) {
    return i % QueueSize; //should be fast as
            // long as QueueSize is power of 2
  }
};
```

Listing 2 (cont'd)



manifest itself anymore (!)). However, this specific queue happened to be debuggable very easily:

> *I was able to debug it within half a day.*

I contend that anybody who has tried to debug multithreading programs of comparable complexity will realize *how fast half-a-day is for this kind of not-so-trivial multithreading*.

Maintainability

After it started to work, I ran some experiments, and found that with one single writer, it performed great: *with real 128-bit CAS, I measured an upper bound performance of this queue at about 130 nanoseconds per* `push()`+`pop()` *pair*. However, with more than one writer, performance was observed to degrade very severely (around 50×(!)).

After thinking about it for a few hours, I realized that, actually, the code used in the examples above can be improved *a lot* – in particular, we can (and ***should***) avoid going into thread-sync stuff *on each and every call to* `pop()`. Indeed, as our queue can handle up to 64 slots at a time, we can read all of them into some kind of a 'read cache' (with proper synchronization), but then in subsequent calls to `pop()` we can easily read all the cached values without any thread sync involved. This optimization allowed me to improve performance in tests with two writers by over 50× (so that performance with two writers became about the same as performance with one single writer). BTW, if you want to see the code with this 'read cache' optimization, it is a part of current implementation in [NoBugs18].

However, my main point here is *not* about the performance of this particular queue. What I want to emphasize is that:

- In spite of the queue being rather complicated, it was *easy* to reason about it
- After I realized what I want to do, implementing the whole thing (writing + debugging) took less than two hours(!). Once again, this is *extremely* fast for writing/debugging a significant change for *reliably-working multithreaded programs*.

Of course, a lot of further optimizations are possible for this queue (in particular, I am thinking of introducing 'write caches' along the lines of the 'read cache' above); still, even the current (not perfectly optimized) version in [NoBugs18] *seems* to perform pretty well under close-to-real-world usage patterns. On the other hand, *please treat the code in [NoBugs18] as* ***highly experimental****, and be sure to test it very thoroughly before using it in any kind of production; multithreading bugs are sneaky, and there is always a chance that one of them did manage to hide within, in spite of all the reliability improvements provided by CAS (Re)Actors.*

Conclusions

We have demonstrated how the real-world task of 'creating an MWSR queue with flow control and minimal locking' can be implemented using the concept of CAS (Re)Actors (which was discussed in detail in [NoBugs17]).

In the process, it was also observed that

> *Not only do CAS (Re)Actors allow us to write multithreaded programs very easily (by the standards of multithreaded programs, that is), but also CAS (Re)Actor-based programs are easily maintainable and easily optimizable.*

As a nice side effect ☺, we also wrote a practically-usable MWSR queue with flow control and minimalistic locking, which can take as little as 120 nanoseconds per `push()`+`pop()` pair ☺. ■

References

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

[NoBugs17] 'No Bugs' Hare, CAS (Re)Actor for Non-Blocking Multithreaded Primitives, *Overload* #142, December 2017

[NoBugs18] 'No Bugs' Hare, mtprimitives, https://github.com/ITHare/mtprimitives/tree/master/src

[Wikipedia] Optimistic concurrency control (OCC) https://en.wikipedia.org/wiki/Optimistic_concurrency_control

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague

Testing: Choose the Right Level

Testing can be easy. Andy Balaam considers levels to keep your focus just right.

When someone complains that their tests are not providing the benefits that they were promised, or are more trouble than they're worth, some of us may be inclined to nod wisely and muse that testing is a skill that must be learned.

Testing is a skill, bad tests are barely better than no tests at all, and sometimes writing tests is painful.

But sometimes, testing is easy and good, and we feel productive and safe in the knowledge that our code must be correct, because the tests specify our requirements simply, and unambiguously.

This article will argue that the most important difference between tests being a pain or a joy is the level at which we have chosen to write them. We will begin by defining what we mean by a level, then cover how to choose what level to test at, and the importance of testing at multiple levels. Finally we will look at some examples of times the author has seen the practical effects of the choice of level on the experience of testing software.

What is a level?

Each test will run the code under test by calling it via an interface, and may also insert test code via 'seams' [Feathers04]. Choosing a 'level' for your tests simply means deciding what group of interfaces and seams you will use to invoke the code under test.

Examples of test levels include:

- calling individual methods and checking return values
- constructing a few independent classes, calling methods on them, and checking object state or return values
- instantiating a large group of classes backed by mocks and checking the inputs and outputs via high-level method calls
- expressing tests as expected output when certain input is provided to the executable under test
- exercising full running systems via external interfaces e.g. HTTP.

Note that several of the examples above concern object-oriented code, but the idea of choosing a level applies to other coding styles too.

Choosing a level

Often the best way to choose a test level is to try and write some tests at different levels and discover which test combines simplicity and power, where by power we mean ability to express meaningful test cases.

Not too wide

The best level to test your code will not be too wide – your tests should not be prone to failure because of the behaviour of uninteresting areas of code. Uninteresting areas could simply be different components, or third-party libraries or services.

You can tell tests are too wide when:

- there are many possible reasons why a test has failed
- it is hard to debug failures because lots of different systems need to be observed
- tests run too slowly to give useful feedback.

When tests are too wide you should consider whether there are more direct interfaces that may be used to exercise the code under test. An important advantage of the practice of test-driven development [TDD] is that it tends to lead us to build more such interfaces into our design.

Not too narrow

Tests should not be too narrow: the interesting and difficult parts of the code must not be left out at test time. Where we are implementing some pure logic it is often easy to write tests that cover that logic, but where we are making use of an external component, and the most common problems will be caused by misunderstandings of that component's behaviour, it is more difficult to write a wide enough test.

You can tell tests are too narrow when:

- production bugs are greeted with a chorus of "but the tests passed!"
- code that mocks external components contains encoded assumptions about how those components work.

Not too much setup

Our tests often consist of 'given' (set-up), 'when' (doing something) and 'then' (assertions) phases. When the 'given' part is long, error-prone and complex, we know we have chosen a level that requires too much setup.

A particularly troublesome variant of this situation is when we must instantiate large numbers of interconnected mock classes just to create an instance of a class under test. This causes problems when the code under test changes, and tests fail because the mocks no longer express the true behaviour of related classes. More description of how to avoid complex mocks may be found in [Balaam].

Note that this item concerns the size of the code actually written within the test code. If the runtime cost of setup code is high, that might indicate a test that is too wide, but not one with too much setup in this sense.

You can tell tests have too much setup when:

- each test has a long, complex 'given' phase at the beginning
- it becomes necessary to write test fixture classes just to hold on to all the state needed to start a test
- the tests are dependent on the details of multiple interconnected mock classes.

Choose multiple levels

Normally, a single level of testing will not provide sufficient coverage of the interesting parts of our code. It is almost always necessary to write

Andy Balaam is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

tests at the full-system level in order to have confidence that the system works, but the full-system level is usually too wide to allow efficient testing of all the details of behaviour.

Writing a few simple, real-world tests at the system level (ensuring build and integration issues are discovered early), and large numbers of detailed correctness tests at the code level (written as the code is written) can often be sufficient. In more complex systems it often makes sense to write tests of larger chunks of logic at an intermediate level, and here is where choosing the right level can be tricky. The best level will allow writing 'acceptance' tests where we can express genuine customer requirements as individual tests, and run those tests in a reasonable time, without excluding the code that is most likely to contain bugs.

Examples

The following sections describe examples of real-world projects that involved choosing a level at which to test, and an assessment of the success of that choice.

Reporting metrics

In a high-availability messaging product written in Java that emitted a large set of metrics for real-time monitoring, the team found that the metrics repository was becoming overwhelmed, so we needed to reduce the number of individual metrics being emitted.

Metrics reporting was provided by a third-party library which directly makes HTTP requests in response to method calls, and provides an API to filter the list of metrics emitted.

Testing the filtering logic we provided to the library was tricky because it was highly interlinked with the behaviour of the library, and there was no mechanism available to ask which metrics would be emitted, or provide a mock HTTP client to track the outcome without actually making HTTP requests.

We were left with a choice between unit testing the filtering logic, accepting that the assertions we were making were making sweeping assumptions about the library behaviour (too narrow), or testing the logic in an environment that actually allowed and tracked the real HTTP requests (too wide).

We eventually added assertions to a test within our full system test environment that actually included the metrics repository as well as the messaging component, checking that the repository had received only the correct subset of metrics.

This seemed most unsatisfactory, and emphasises the importance of Michael Feathers' Golden Rule of API Design: "It's not enough to write tests for an API you develop – you have to write unit tests for code that uses your API." [Feathers07]. Ideally we would be able to provide the filtering logic to the metrics library and ask it to list which metrics will be emitted.

Command line tools

In a team maintaining a suite of auxiliary utility tools used on the command line, we settled on a strategy of writing tests that invoked the actual executable commands with pre-specified input and expected output. The tools were written in a variety of technologies, and the tests were written in Python, executing the commands using the subprocess module.

One of the design goals of these tools was to ensure each executable had a simple, well-defined job so that tools could be combined in flexible pipelines to handle unexpected scenarios.

The choice of using only system-level testing encouraged us to follow this design, since our team of experienced TDD-ers were uncomfortable writing too much code without a direct test, so they tended to break complex code into multiple tools to allow coherent testing.

System level testing made it easy to test features that could easily be overlooked such as command-line argument processing and formatting of output.

We found this level to be ideal for testing this kind of tool, and rarely felt concerned by the lack of a pure unit test layer. Sometimes when writing Python code we wanted to write unit tests, but this was often addressed by breaking complex code into multiple executables. When writing tools in Bash and other less naturally testable environments, we felt liberated by the fact that our test setup already made writing tests easy.

Most of the tools involved consumed standard input and emitted results to standard output, but where they interfaced with external systems such as the file system or the network, our approach made it hard to test. We limited such tools to be as simple as possible and did not include them in our test coverage, which was not ideal, but rarely caused problems in practice.

Usually, testing only at the system level will cause tests to run too slowly to be useful, but in this case our suite of hundreds of tools completed its test run in under 10 seconds, which was good enough for our development process.

Game logic

In a rabbit-focussed Android puzzle game [RabbitEscape] we needed to test large numbers of scenarios in the game model, including interactions between different rabbit behaviours and user actions (e.g. if I give a rabbit the 'climbing' ability while it's building a bridge, will it stop building?).

Since the game model operates in discrete time steps, using a coarse grid of spacial positions, we chose to represent game states in ASCII art-style text grids. This allows us to express tests of behaviour as sequences of text 'pictures' of the successive game states where e.g. a rabbit is represented as an 'r' character, and a floor block is represented by '#'.

Writing tests of the game-model component at this level, encoded in this way, has been remarkably successful. It is easy to read and understand old tests, and turning a bug report into a failing unit test is a satisfying and clear process.

The code base contains plenty of 'normal' unit tests, but where we need to test the in-game behaviour we always turn to this method for its clarity and ease of writing. There is some run-time overhead in parsing and rendering text representations of the game state, but this has not caused us a performance problem so far (hundreds of tests run in about 5 seconds).

One concern in this approach is that tests might become too wide – when testing a single behaviour or bug we instantiate a whole game world containing all the game logic, as well as the text parsing and rendering, meaning our tests could fail for reasons irrelevant to the logic we want to test. In practice, although tests do sometimes start failing unexpectedly, more often than not this is because of an interaction we did not anticipate, that we are glad to find out about, so the extra coverage of the tests generally works to our advantage.

Compared with most test frameworks we have seen, this one is a joy to work with: we are very happy with the way it works – almost no test feels pointless, and newcomers can read and understand the tests very easily.

Object model

In a JavaScript object model representing domain objects for an interactive charting application, we needed to implement logic to synchronise changes between client and server.

We chose to test purely at the unit level, where we took 'unit' to mean a piece of functionality, rather than an object or module. Most test modules contained multiple tests exercising the same 1–3 objects from the code under test, exercising different usage of the same code. Each test expressed a single idea, and each test module served as a specification for that behaviour.

Writing the tests and code in this way was a pleasure, and the functionality was well-covered and easy to understand, partly because of this approach. One advantage of viewing units in this way is that it pushes towards better design: if a test module began to involve more than about three objects, we took this as a prompt to revisit the design and see whether the behaviour could be implemented in a more coherent, simpler structure.

One result of choosing this level was that the interface between client and server was not covered by these tests. Separately, we built tests that instantiated a multi-language environment to test the client and server

interactions, effectively running a JavaScript interpreter within the server platform. These tests were unreliable and failures were hard to understand and debug.

In retrospect, our full system test environment (including a full web browser and a server environment) might have been a more pragmatic way to ensure the client-server communication worked, instead of trying to cover it explicitly using headless tests. Certainly, separating the pure logic into narrow tests that touched no file system or network resources proved highly effective for giving us confidence in the logic, but did not cover enough to provide full confidence in the system as a whole.

Web service

Working in a large, unfamiliar legacy Java code base for a set of web APIs, we found an attempt had been made to test at a component level.

Most of the tests were JUnit methods that relied on a complex stack of mock objects that, when correctly instantiated, replaced the code that made external connections (e.g. TCP sockets, files) in the production code.

Because the system's observable behaviour consisted mostly of network traffic, we knew we would need plenty of system-level tests to convince ourselves that it was behaving correctly. However, the existing mock-based tests were a perfect of example of testing at the wrong level: they were not wide enough to cover the real-world behaviour (e.g. what happens when we encounter a poorly-configured DNS server), but they had all the down-sides of system-level tests: they were unreliable, timing-dependent and slow, and depended on the system being in the correct starting state to run correctly.

At the same time, creating new tests was slow and error-prone due to the large number of mock classes needed, and tests often broke when the assumptions encoded in the mock structure became invalid due to changes in the code under test. Effectively we were testing the mocks more than the interesting code.

Furthermore, many of the components in this system were tightly integrated with other components, meaning it was difficult to be sure they were working correctly without a true system test that ran lots of parts simultaneously.

Our team replaced the heavily-mocked component tests one by one with true system-level tests that instantiated different subsets of the full production set-up inside repeatable Docker-based environments, and exercised the system through its real network interfaces. Meanwhile, we gradually increased the coverage of true unit-level testing by writing unit tests whenever we changed the code. Within months, our test runs were more reliable, tests were effective at finding real problems, and the failure rate of the production software reduced.

By changing the level at which we were testing, from the complex Java interfaces of the external components to the simple and relatively slow changing external HTTP interfaces, we simplified the job of testing, making it much clearer what the expected input and output were. Where fake or mock services were needed, they were simple independent code bases, or often could be implemented to provide hard-coded HTTP responses, using the Python http.server module.

The Docker-based tests were slow – even slower than the old mock-based tests – and they were not 100% reliable, but the significant improvement in reliability, and the much better coverage of real-world scenarios, was well worth the extra time. The far better comprehensiblity of the tests was perhaps the key advantage longer term, as we worked to understand this complex legacy system.

Tree merge

In a large C++ UI application, functionality existed to merge two existing models based on a large set of rules for how to combine potentially clashing hierarchical trees. Building up models in code was complex and verbose, and deciding whether the produced model was actually correct was difficult.

We took the decision to describe object models in a custom domain-specific language (DSL). This allowed us to write tests that clearly described the input and output conditions without the noise of boilerplate code interfering. The DSL took only one line to describe each object in the tree, meaning most of our tests became only one or two screens of code, and the expected behaviour was clear.

Using a custom DSL has many potential disadvantages, such as the lack of a debugger, but we had great success using one to describe object models, perhaps because instantiating objects is a simple enough process that it does not need to be stepped through line by line. We could have taken the route of writing simple functions to create objects, and stayed inside the main language, but the key advantage of the DSL in this case was that test failures produced a clear diff (in the notation of the DSL) showing what object model was expected and what was actually seen. This made interpreting test failures so much simpler that we could write in a test-driven development style, writing a test and using the failure to drive changes in the code under test. This way of working was an order of magnitude more productive than debugging individual assertion failures which gave no overall picture of the difference between expected and actual behaviour.

By changing the level of testing to a wider level (complete input and expected output, instead of hand-coded variations on an input model and expected features of the resulting output) we greatly simplified our tests and made them much more useful. The use of a DSL was helpful, but less important than the shift in testing level to one that naturally suited the problem.

Conclusion

When deciding how to test your code, it is important to consider what level makes sense for the project. You should try to choose a level where you can:

1. express your requirements simply.
2. run the important parts of the code (not mock them out).
3. easily interpret test failures.

You can tell tests are at the right level when:

- No test feels pointless – each test verifies some real part of the spec.
- There are no gaps where the really interesting stuff happens but can't be tested.
- It is easy to write the next test.

You can tell the level is wrong when:

- Tests consist of more setup than actual test code.
- Getting your mock working is harder than writing the real code.
- Tests are unreliable.
- The real interesting behaviour is not tested.
- Adding another similar test is hard.
- It takes too long to run your tests.

Often you will need two levels to cover specific code units and whole-system behaviour. Large code bases may need more than two levels – where this is needed, try to find a level that lets you view a component as having clear inputs and outputs. ■

References

[Balaam] Balaam, A.J. (2015) 'Mocks are Bad, Layers are Bad' In F. Buontempo, editor, *Overload* 127, pages 8–11.

[Feathers04] Feathers, M. (2004) *Working Effectively with Legacy Code* Prentice Hall

[Feathers07] Feathers, M (2007) 'API Design as if Unit Testing Mattered', presentation at SD West 2007 https://www.scribd.com/document/60239205/As-if-Unit-Testing-Mattered

[RabbitEscape] Rabbit Escape, http://artificialworlds.net/rabbit-escape

[TDD] Beck, K.(2002) *Test-Driven Development by Example*, Addison Wesley

CTAD – What Is This New Acronym All About?

What is class template argument deduction? Roger Orr elucidates this new C++17 feature.

C++17 has now been shipped and the dust is settling. There are a number of new features in the language; one of the last to be added before the final cut goes by the snappy acronym of CTAD. The full name is Class Template Argument Deduction, which may not tell you a great deal more than the acronym does.

Example with std::pair

When using class templates you've always had to provide the template arguments even when their type was obvious from the use:

C++98 code

```
void test(int id, std::string const &name)
{
  std::pair<int, std::string> p(id, name);
  // ...
}
```

I've put the types involved in bold to make the duplication clear. The template arguments have to be provided although it's pretty clear what they are.

Things changed slightly with the introduction of **auto** in C++11; it became possible to use the (pre-existing) helper function **make_pair** to create the variable and so avoid duplication of the types:

C++11 code

```
void test(int id, std::string const &name)
{
  auto p{std::make_pair(id, name)};
  // ...
}
```

However, this relies on the existence of the **make_pair** function template so if you wished to provide a similar facility for your own class template you had to ensure a helper function was available. It was simply an idiom to enable using the language rules which do allow template arguments to be deduced when calling *function* templates.

Class template argument deduction allows us to avoid duplicating type names even when using the constructor syntax:

C++17 code using CTAD

```
void test(int id, std::string const &name)
{
  std::pair p(id, name);
  // ...
}
```

The compiler detects that **pair** names a class template but no template arguments are supplied and deduces the arguments from the types used in the call of the constructor. (Hence the name of class template argument deduction.)

Use of CTAD makes the class type of the variable **p** *explicit*. It removes the need to define a helper function such as **make_pair** – and it is a better technique as the use of a helper function relies on a naming convention to specify the type to the reader of the code.

```
// Existing C++ class
template <typename T>
struct point
{
  T x;
  T y;
};

int main()
{
  // New C++17 use
  point pt{0L,0L};
}
```

Listing 1

As the paper containing the wording [P0091R3] put it in the summary: "If constructors could deduce their template arguments 'like we expect from other functions and methods,' ..." This is pretty much what CTAD does.

Since CTAD is now a standard language feature it is available for any existing class templates without any additional changes (see Listing 1).

Potential problems

First of all, note that the addition of CTAD to C++17 does not break any *existing* code as it simply allows some formerly ill-formed code to become valid.

However, before you race to your code-base and remove all explicit specification of class template arguments on variable declarations, there are a few corner cases that might be troublesome.

First of all, the deduction process will only work if there are constructors for the target type that *use* the template arguments; the process cannot magically guess the type from the arguments. So, for example, CTAD is of no use for the following class as the template argument is not part of the constructor signature:

```
template <typename T>
class collection
{
  public:
  collection(std::size_t size);
  // ...
};
```

Roger Orr Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

The wording for class template argument deduction includes the option of providing explicit deduction guides

It can also be a problem when the constructor you desire to invoke uses types derived from the template arguments as there is no way by default to work 'backwards' to deduce the underlying template argument.

For example, if you wish to construct a **vector** from a pair of iterators and hence invoke the constructor:

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
const Allocator& = Allocator());
```

Doing this directly is problematic as the template argument type for the target collection is *implicit* in the **value_type** of the supplied iterators. (We'll see below one work around.)

Secondly, if there are several constructors, the constructor you want may not be the one that the deduction will find. This may be a problem if you are trying to use CTAD with classes that were written before C++17 as designing in usable constructors will not have been necessary then. Even small details of the way classes are written can render CTAD inoperable.

Thirdly, in some cases, you *do* want to use a helper function to create instances of the class for other reasons. One obvious example from the standard library is **std::make_shared**. As Scott Meyer's *Effective Modern C++* puts it in Item 21: "Prefer **std::make_unique** and **std::make_shared** to direct use of **new**."

Consider the following example:

```
#include <memory>
int main()
{
  std::shared_ptr<int> p(new int(10));
  auto q{std::make_shared<int>(10)};
  std::shared_ptr r(new int(10)); // C++17
}
```

The construction of **p** and **r** differ in whether the type is explicit or deduced, but in both cases the shared pointer will need to allocate an additional piece of data to manage the shared object. In the construction of **q**, the target object and the associated management structure can be created using a single allocation.

Helping choose the right constructor

The wording for class template argument deduction includes the option of providing *explicit* deduction guides.

The syntax is similar to that of a function template, except as the declaration is for a constructor-like entity there is no return type.

For example, as we saw above for the case of a vector, the language will not deduce a template argument for the vector constructor taking a pair of iterators. The *deduction guide* in Listing 2 was added to the standard library [P0433R2]. Omitting the (defaulted) allocator argument for ease of understanding we get Listing 3.

This instructs the compiler when it sees a constructor call taking a pair of iterators, to construct a vector using the value_type of the iterator type, such as:

```
void foo(std::set<int> const &c)
{
  std::vector v(c.begin(), c.end());
  // ...
```

and **v** will be deduced as **std::vector<int>**, as expected.

```
template<class InputIterator, class Allocator
   = allocator<typename   iterator_traits<InputIterator>::value_type>>
vector(InputIterator, InputIterator, Allocator = Allocator())
-> vector<typename iterator_traits<InputIterator>::value_type,
     Allocator>;
```

Listing 2

```
template<class InputIterator>
vector(InputIterator, InputIterator)
-> vector<typename iterator_traits<InputIterator>::value_type>;
```

Listing 3

(Note that this is more restrictive than the range of options available using the full syntax, as in this case the vector to be constructed can, in general, be of *any* type that **int** can be converted to.)

Helping prevent inappropriate choices

There are times when you may with to prevent use of CTAD for a class, for example when the constructor chosen is unlikely to be the one that the user expected.

One way to remove the possibility of using class template argument deduction is to change the constructor to take a type *derived* from the template arguments (see Listing 4).

The compiler is not allowed to 'work upstream' and deduce a possible value of **T** (i.e. **int**) that would make **no_ctad_t<T>** match the supplied argument. This is the same rule that already applies to regular template argument deduction on function calls.

(Note: Timur Doumler is proposing standardising a class similar to **no_ctad** – this use case is one of the motivating examples)

A second way, detailed in [P0091R4], would be to extend the usage of deleted functions (**"=delete"**) to also allow for deleted deduction

your ability to make use of class template argument deduction in your code will depend on which compilers your project is targetting

```
template <typename T>
struct no_ctad { using type_t = T; };

template <typename T>
using no_ctad_t = typename no_ctad<T>::type_t;

template <typename T>
class test
{
public:
  test(no_ctad_t<T>);
};

int main()
{
   test t(1); // Error
}
```

Listing 4

guides. This has not yet been adopted into the working paper, but approval for this direction has been given by the Evolution working group.

Primary template and explicit specialisations

Class template argument deduction applies to the primary template. If an explicit specialization of this template defines different constructors these will *not* be found by CTAD. See Listing 5, for example.

Future directions

When CTAD was first proposed it was suggested that you could provide *some* template arguments and deduce the others. This is not currently in the working paper as it was felt safest to start with an 'all or nothing' approach where there are fewer opportunities for confusion or ambiguity and to consider a future extension if one is provided.

Compiler support

CTAD is part of C++17 and so will eventually be provided by any compiler offering support for the current C++ standard.

However, at the time of writing (end of 2017) not all the mainstream compilers have yet released versions implementing this part of the language.

So for example while both gcc 7.1 and clang 5.0 support the feature, the latest version of MSVC does not [C++17 progress], nor does it appear that Intel's compiler does. The status may of course have changed by the time you read this article.

So your ability to make use of class template argument deduction in your code will depend on which compilers – and which version of those compilers – your project is targetting.

Summary

Class template argument deduction does not let you write any new code that you couldn't already write, albeit with slightly more syntax.

However, the reduction in syntax does reduce the burden for the reader of the code and also ensures that, since the target types are deduced, the code automatically changes if the type of the supplied arguments change.

I consider CTAD a useful technique for reducing cognitive overhead and improving readability. ■

References

[C++17 progress] https://blogs.msdn.microsoft.com/vcblog/2017/12/19/c17-progress-in-vs-2017-15-5-and-15-6/

[P0091R3] http://wg21.link/p0091r3

[P0091R4] http://wg21.link/p0091r4

[P0433R2] http://wg21.link/p0433r2

```
template <typename T>
class myclass {};

template <>
class myclass<int>
{
public:
   myclass(int);
   // ...
};

int main()
{
   myclass m(1); // Error: primary template only
                 // has a default ctor
}
```

Listing 5

C++ with Meta-classes?

Meta-classes will allow us to detail class requirements. Francis Glassborow compares them to developments of C++ in the 1990s.

Back in the early 80s Bjarne Stroustrup produced an upgrade to C that was originally called 'C with classes'. Superficially it was nothing special because it did not add anything that could not be done in C by a sufficiently competent programmer. At that time some were brave enough to call it a 'better C'. C was in the early stages of being standardised both by ANSI (the US Standards body) and by ISO. The US Standard was published 1989 as ANSI C. A year later ISO effectively adopted ANSI C with a bit of editorial work to get it into the form required by ISO. The result was an international standard for C.

The C purists were not exactly on board with the idea that there could be an improvement to their chosen language. Indeed, some still feel that C is the best of breed and obstinately ignore the existence of C++. C still has a place in the world of programming but I am unconvinced that it deserves the time and effort that goes into evolving it. Modern optimisation technology can make good use of the added expressiveness of C++ to deliver small footprint, high performing executables from well written C++ source code.

It is worth noting that for most of the 80s 'C with Classes' or C++ as it came to be called was implemented via CFront. That was basically a translator that converted raw C++ into C which could be compiled by any suitable (effectively standard) C compiler. That allowed interested people to use C++ on any platform that had a C compiler.

Most significantly, CFront demonstrated that the core of C++ was C-based though additions such as templates (held with profound suspicion by many programmers for at least a decade) moved it into a realm of its own.

It took many years for writers and programmers to grasp that C++ was very different from C and that the idioms of the language were fundamentally different. Good C is largely not good C++. Unfortunately, part of the legacy that C++ has to cope with is attempting to maintain a compatible core that is also valid C. Those who program in both languages are all too aware of how difficult this can be. Just take the problem of implementing support for complex numbers. This is reasonably straight-forward for C++, even to the extent of supporting different levels of precision (float, double and long double). To do the same thing in C required considerable contortions including adding core support not only by adding new built-in types but added conversions to allow complex numbers to interact with floating point types.

C++ can often add new types as library extensions without changes to the core of the language. If you want quaternions or extended precision floats or Dr Conway's Surreal Numbers you, as an individual, can implement them in C++ in a natural way including support for operators and transparent support for mixing them with other arithmetic types.

Francis Glassborow Since retiring from teaching, Francis has edited *C Vu*, founded the ACCU conference and represented BSI at the C and C++ ISO committees. He is the author of two books: *You Can Do It!* and *You Can Program in C++*.

These days C++ has added a great deal including support for concurrency (hard because that field is littered with gotchas), support (initially via Technical Specification) for modules, contracts, concepts etc. Each of those additions has major impact for the programmer and makes heavy demands on language designers and implementers.

All this developed from a single step forward aimed at making C more expressive and usable. Classes are a powerful addition to C though the idea is relatively simple and makes relatively little demand on design and implementation. Of course, that is not true of the superstructures that have been built on top of classes. Without classes, the provision of support for generic programming via templates would have been the domain of a tiny number of master programmers (yes, you can do generic programming in C, but who would want the pain of doing so?). Templates enabled meta-programming because the implementation of them made demands on compilers that could easily support some aspects of meta-programming. More recently C++ has added more direct support for meta-programming.

Who could have foreseen the long-term consequences of introducing classes to C? Yet that step has produced the rich, vibrant living language that is C++ today. There have been and are attempts to produce a successor to C++. Some of these have some positive advantages (usually as a result of not having to worry about C compatibility). Java and D both have something to offer and yet seem, to me, to lack the vibrancy of C++.

It is time for something new

Almost everything that gets added to C++ increases the complexity either for the programmer or for the implementer. Hopefully, most complexity is for implementers but some things aimed at library designers can be daunting for the application programmer when they look at the resulting libraries (usually from curiosity but sometimes because a compilation error message takes them into library source code.)

This has remained true until April of 2017 when Herb Sutter gave the endnote to the ACCU Conference with the caveat that we would keep quiet about it until he had had a chance to present the idea to WG21 later in the year.

This new thing was 'meta-classes'. Like classes, they do not appear to add anything that cannot be achieved without them. However, to this writer, they are just as much a game changer as were classes added to C.

Let me give you a very brief overview of the idea and then leave you with some links that will allow you to get real information and discover that you can already start exploring some aspects of their use.

First note that they rely on compile time reflection: another addition to C++ that is fast coming down the line. Have a look at http://jackieokay.com/2017/04/13/reflection1.html and https://meetingcpp.com/blog/items/reflections-on-the-reflection-proposals.html. Or search for C++ reflection.

It is clear that static reflection will be added to C++ and that compilers (Clang, G++ and Visual C++) are already providing experimental support for it. Be careful because whilst these are close to what will eventually be

A meta-class specifies a set of required functions to the compiler, a set of prohibited functions and defaults for functions that will be used if the programmer does not provide a version

added to the C++ Standard you should be prepared to modify code using reflection to conform to the eventual standard specification.

So what are C++ meta-classes? The idea is that we should be able to specify what is required to produce a family of types. For example, a value type has a need for specific functions which we currently have to provide by hand. The compiler cannot assist us in either detecting that we have left one out, nor in providing a suitable default. The problem is that unless you can communicate that the class that you are writing is to implement a value type the compiler cannot use reflection to determine if you have implemented your intention. Nor can it provide suitable defaults.

A meta-class specifies a set of required functions to the compiler, a set of prohibited functions and defaults for functions that will be used if the programmer does not provide a version. Think of the way a class provides defaults for copying (assignment and cloning), construction and destruction. It wasn't until we had `=delete` that we had an easy way to prohibit copying, and default construction. Yes, you can do it but by tricks and idioms (such as declaring and not defining a copy constructor to inhibit cloning). Such things are error prone and lack transparency. If you do not know the idiom for turning off copying you probably will be confused by it.

Meta-classes will allow us to define what constitutes an object type, a value type, an arithmetic type etc. They will enable the compiler to generate a great deal of what we currently have to do by hand. If you wish to see how much work we have to do to implement an arithmetic type have a look at the implementation of a complex number. Yet much of that is just boiler plate code. Perhaps the complexity of implementing std::pair is even more surprising (shocking).

Like classes, meta-classes achieve very much more than a superficial description suggests. Reflection is a useful tool but not a game changer in itself.

I assume that the reader is not someone who wishes to bury their head in the sand like the C programmers who wanted nothing to do with C++ and so bifurcated K&R C into ISO C and ISO C++.

Some of the debates on Reddit begin to look very similar to some of the comments made about C++ in the early 90s. Assuming that readers avoid the knee-jerk reaction and want to know more before forming an opinion, look on the web for C++ meta-classes. A good starting point is: https://herbsutter.com/2017/07/26/metaclasses-thoughts-on-generative-c/. You do not need to type that in, just search for 'meta-classes in C++'.

When you have brought yourself up to date on meta-classes please start thinking about ways they would help you. There is partial support for meta-classes in Clang. You should be able to try out published examples and experiment with your own ideas. I invite you to report on your experiences both good and bad.

Have great coding experiences in 2018 and may all your bugs have six or more legs. ■

And the winners are...

In the last *Overload* we invited our readers to vote for their favourite articles of 2017 in *CVu*, which is our sibling magazine for members, and in *Overload*.

For CVu:

1st place: Sean Corfield for 'I Can't Think Fast Enough in a Coding Interview'

2nd place: Adam Tornhill for 'Beyond Functional Programming: Manipulate Functions with the J Language'

For Overload:

1st place: Simon Brand for 'Initialization in C++ is Bonkers'

2nd place: Ralph McArdell for 'C++11 (and beyond) Exception Support' and Katarzyna Macias for 'A C++ developer sees Rustlang for the first time'

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

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

Practical Scale Testing

Everyone wants scalable systems. Arun Saha explores methods for testing scalability.

Scalability is the ability of a system to handle a growing amount of work or its potential to accommodate such growth. It characterizes how resource utilization grows with increasing load [Wikipedia-a]. Scalability is not an 'add-on' item; it is a quality that determines the lifetime value of the software. It may not be apparent on day one when the focus is usually on the features, but it is very important for the eventual growth. Therefore, from early on, it is important to test the software and verify that the desired scalability properties are present.

What is a desired scalability property? Here is a textbook example. Consider a system that sorts data. In Big-O notation, the time complexity of a good sorting algorithm (for example, quicksort, merge sort, heapsort) is `O(N lg N)` where is `N` is the number of items being sorted [Wikipedia-b]. Thus, a desired scalability property of this system is that the time taken to sort `N` items is `O(N lg N)`, i.e., the time consumed is proportional to `N lg N`. The only way this can be examined is by running inputs of different sizes through the system [Orr14]. With sufficiently large input size, each time the input size is doubled, the time taken is expected to be doubled too. For example, with `T1 = k lg k`, and `T2 = 2k lg (2k)`, `T2/T1 = 2k lg (2k) / k lg k = 2 + 2 / lg k ~= 2`. If, however, the system is using a sub-optimal sorting algorithm of time complexity $O(N^2)$ (for example, bubble sort, insertion-sort) then the ratio of the times taken would be quadrupled (since $T2/T1 = (2k)^2/k^2 = 4$). How many different sizes to run? See the discussion in 'Test procedure', below.

The holy grail of scalability is **linear scaling**, i.e. the measured metric changes linearly with the changed attribute. With linear scaling, if the measurements are plotted on a graph, then they approximately form a straight line.

Scalability testing is a type of non-functional testing; tests are only conducted on systems that are proved functionally correct. In the example of sorting system above, the time complexity tests are meaningful and attempted only *after* it is verified that the system can *correctly* sort input of different sizes including sufficiently large sizes.

There are two main motivations of scalability testing: (a) identify the limits of the system; (b) verify that the system performs well up to the previously identified limits or desired limits. It is important to identify the goals precisely; for example, a system can support 1M users but it may not be able to support 1M users *together*. In this case, an important thing to know is, how many users can be supported simultaneously.

Usually, the scale tests are carried out as black box tests. In a broad sense, it involves two kinds of tests and measurements: (1) customer-driven (or deployment-driven) and (2) architecture-driven. The former involves real-life situations although not necessarily everyday situations. For example, can a system of cellular towers handle the load when 100K people gather for a special event? The latter are internal-driven tests, which keep loading the system and study its behavior. There is a connection though; the numbers obtained from the architecture-driven tests influence future deployments.

In this article, we present the key characteristics of successful and practical scale testing at the system level. These characteristics can form the basis of system or load tests. We will consider attribute selection, metric selection, and the testing process. These considerations help designing scalability tests for a system.

This article is not about microbenchmarking [StackOverflow] where a 'small' specific section of code (often a class or a function) is tested for performance.

Identify scalability attributes to explore

A system usually has many different knobs and parameters that could be scaled. Some of them are logical entities without any intrinsic limit, for example, the number of users, while others are physical entities tied to some limits, for example, hardware capacity and operating system constraints. From another perspective, some parameters are exposed externally (to the users of the software), such as maximum size of the items to be sorted, while others are internal, for example, block size for saving data.

As we saw above, gauging the scalability of a single attribute requires running a specific test multiple times. Therefore, to make the best use of available time, staff, and computing resources, it is important to identify and prioritize the attributes whose scalability properties are to be explored. For example, in a document management system, it may be more important to have 'search' scalability (since search results are needed instantly) than 'remove' scalability (since removal can be performed asynchronously through batch jobs).

There is no point exploring the scalability properties of attributes that are related, choose only those attributes that are *independent* of each other. For example, the number of simultaneous users and the number of (external) network connections are related (i.e. dependent); it may not make sense to choose both of them as attributes.

The attributes together form a scale-space where each attribute is a dimension and each combination of the attribute values is a point in that space.

Identify a set of metrics for each attribute

Some general metrics that characterize system performance are, for example, throughput, latency, CPU usage, memory usage, network usage, disk usage.

Each kind of system has its own family of metrics. For example, in a storage system, some commonly used metrics are sequential read IOPS (input-output operations per second), sequential write IOPS, random read IOPS, random write IOPS. In a system involving video, a common metric is frames per second. A database system uses queries per second; a

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

Scalability testing is a type of non-functional testing; tests are only conducted on systems that are proved functionally correct

networking system uses packets per second (for throughput) and time per packet (for latency).

There are many other metrics capturing other aspects of a system, besides performance. Examples include

- time to recover from a disk failure
- time to recover from a node failure
- time for traffic to converge after a network failure.

The recovery related tests and obtained metrics provide the expected time to recover from a failure situation. They serve as a useful reference to the customers and the operations community.

It is worth noting that certain metrics may be at odds, for example, the design for optimal (i.e. maximum) throughput may not be the same as the design for optimal (i.e. minimum) latency. Moreover, once the metrics are chosen, the system architecture and design tend to optimize in favor of them. Hence, it is essential to identify what is important to the customers.

Test procedure

Scalability exploration is NOT about running a test once under the maximum scale. Rather, it is a study of the system by running the test for **multiple iterations**, each time with a different combination of attribute values. Without multiple iterations, we can know how the system behaves at a specific load point, but we *cannot* know how the system behaves with an increase (or decrease) of load – the whole point of scalability.

It is usually not possible to obtain measurements for all possible values of an attribute in a continuous sense. So, the desired range of an attribute is broken down into multiple probe points and the test is conducted to obtain measurements for each one of them.

As an example, the sorting system mentioned above may have only two attributes, the number of input items and the size of individual items, and only one metric, the time consumed. Each iteration can run with a different value of the attribute tuple (#items, item-size), exploring different points in the scale-space. In order to explore scalability properties up to 1M items and 256-byte items, one possibility is to design the iterations as follows: 10 probe points for #items (100K, 200K, … 1M) and 3 probe points for item-size (8B, 64B, and 256B). This way, the number of iterations necessary is 10x3 = 30. In general, the number of iterations necessary is the product of the number of probe points of each attribute. The combinations are the cross product of the probe points. The results of such a test design can be organized as shown in Table 1

Depending on the time necessary to run the test, it is possible to run multiple repetitions of a specific iteration and take their mean value as the representative. Other statistical attributes such as median and variance can be considered as well [Winder17]. This provides more confidence in the observed metrics values.

If multiple attribute values are changed from one iteration to the next, then it is difficult to reason which attribute is responsible for the change in the metrics values. So, ideally, **change one attribute at a time** from one iteration to the next. However, if that makes the number of iterations too large to be practical, then some additional strategies may be considered, for example, reduce the number of probe points or increase iterations only in those areas where there are sudden jumps in observed metrics values.

Test Input Id	Attributes		Metrics
	No. of entries	Item size in bytes	Time consumed in ms
1	100000	8	
2	200000	8	
…	…		
10	1000000	8	
11	100000	64	
12	200000	64	
…	…		
20	1000000	64	
21	100000	256	
22	200000	256	
23	200000	256	
…	…		
30	1000000	256	

Table 1

Preserve history: If the software running the system is undergoing change, then the scale characteristics may change from one version to another. Hence, it is important to save the results on a per-test per-version basis and track if the characteristics change from one version to the next or over a series of versions. If things become worse in one version, then it is usually necessary and useful to compare with the results from the most recent previous version that is available.

The idea is that the (continuous) build machinery and the scale testing machinery proceeds independently. The scale test pipeline usually picks up a build, run all the scale tests, reports the results, and then goes to pick up the next build. By that time, it is possible that multiple builds have been done and that is okay. Thus, there may not be data available for all builds. If however, there is a need, it is possible to go back, pick up any build such as one that was earlier skipped and run the test on it. Table 2 contains an example of how the test data can be saved.

Since the attributes and the metrics are likely to be different, each test needs a different schema for its table. The test metrics form a time-series on how a particular metric performed over time. Thanks to the preservation, the performance degradation in Build 120 is easily identified. From the history, it can also be concluded that the degradation happened sometime after Build 111. To find out further whether the regression happened in one build or over a sequence of builds, the intermediate builds need to be tested; a binary search pattern may be used.

Scalability exploration is NOT about running a test once under the maximum scale

Test Input Id	Attribute Tuple (#items, item-size)	Build version	Date	Testbed Id	Metrics (time consumed in ms)
10	(1M, 8B)	108	Jan 2, 2018	42	5
10	(1M, 8B)	111	Jan 3, 2018	42	5
10	(1M, 8B)	120	Jan 9, 2018	42	7

Table 2

Lifecycle of a test

A test goes through multiple stages. First, the test is designed. The design involves choosing the input trigger, the output to verify, the attribute set, and the metric set. The design is reviewed by the appropriate stakeholders. Second, the test is run manually. This is a proof of concept to verify that the test design is okay. Third, test software is written to run the test automatically. This involves invoking commands to configure the testbed, load the test software, set up monitoring for the metrics, trigger the input, collect the output, verify if the actual output matches with the expected output, collecting the logs, and report the results. At this point, the test is ready to be activated. It can be used by humans running scale tests as well as the scale test harness where scale tests are invoked automatically.

Scale testing is mostly performed along with the software development, after unit testing, integration testing, and system testing. During the development cycle, the organization can run the scale tests on as many testbeds that are available and as frequently as things permit. The purpose and the nature of data collection are different after the product is released (i.e. in production) and is not covered in this article. The performance numbers obtained from the scale tests on the final released version of the software are (selectively) published externally and saved as a benchmark for later software versions.

Conclusion

Scalability testing is a broad subject and this article merely scratches the surface. It does not cover related topics such as micro-benchmarking, vertical scaling (scale up) vs. horizontal scaling (scale out). The article explains different aspects of managing scalability tests: designing the test (identifying the relevant attributes and metrics), planning the number of iterations and the attribute combinations to use in each iteration, performing the test by running those iterations, saving the results, verify expected scalability properties, and identify regressions. ■

Acknowledgements

Many thanks to the *Overload* reviewers and the editor Frances Buontempo for their suggestions that have helped improve this article.

References

[Orr14] Roger Orr, *Overload* #124 (December 2014) 'Order Notation in Practice' https://accu.org/index.php/journals/2043

[StackOverflow] StackOverflow, 'What is microbenchmarking?' https://stackoverflow.com/questions/2842695/what-is-microbenchmarking

[Wikipedia-a] Scalability Testing https://en.wikipedia.org/wiki/Scalability_testing

[Wikipedia-b] Sorting algorithm https://en.wikipedia.org/wiki/Sorting_algorithm

[Winder17] Russel Winder, *Overload* #141 (October 2017) 'Marking Benches' https://accu.org/index.php/journals/2427

The opinions expressed in this article are solely the author's – not author's employers'.

Functional Error-Handling with Optional and Expected

Exceptions should be exceptional. Simon Brand shows modern alternatives from the standard library and ways to improve them.

In software, things can go wrong. Sometimes we might expect them to go wrong. Sometimes it's a surprise. In most cases we want to build in some way of handling these misfortunes. Let's call them disappointments [Crowl15]. I'm going to exhibit how to use **std::optional** and the proposed **std::expected** to handle disappointments, and show how the types can be extended with concepts from functional programming to make the handling concise and expressive.

One way to express and handle disappointments is exceptions:

```
void foo() {
  try {
    do_thing();
  }
    catch (...) {
      //oh no
    handle_error();
  }
}
```

There are a myriad of discussions, resources, rants, tirades, debates about the value of exceptions [Douglas17] [Halder16] [Kirk15] [MusingMortoray12] [Stackoverflow], and I will not repeat them here. Suffice to say that there are cases in which exceptions are not the best tool for the job. For the sake of being uncontroversial, I'll take the example of disappointments which are expected within reasonable use of an API.

The internet loves cats. The hypothetical you and I are involved in the business of producing the cutest images of cats the world has ever seen. We have produced a high-quality C++ library geared towards this sole aim, and we want it to be at the bleeding edge of modern C++.

A common operation in feline cutification programs is to locate cats in a given image. How should we express this in our API? One option is exceptions:

```
// Throws no_cat_found if a cat is not found.
image_view find_cat (image_view img);
```

This function takes a view of an image and returns a smaller view which contains the first cat it finds. If it does not find a cat, then it throws an exception. If we're going to be giving this function a million images, half of which do not contain cats, then that's a lot of exceptions being thrown. In fact, we're pretty much using exceptions for control flow at that point, which is A Bad Thing™ [Stroustrup18].

What we really want to express is a function which either returns a cat if it finds one, or it returns nothing. Enter **std::optional**.

```
std::optional<image_view> find_cat
  (image_view img);
```

std::optional was introduced in C++17 [cppreference] for representing a value which may or may not be present. It is intended to be a vocabulary type – i.e. the canonical choice for expressing some concept in your code. The difference between this signature and the last is powerful; we've moved the description of what happens on an error from the documentation into the type system. Now it's impossible for the user to forget to read the docs, because the compiler is reading them for us, and you can be sure that it'll shout at you if you use the type incorrectly.

```
std::optional<image_view> full_view = my_view;
std::optional<image_view> empty_view;
std::optional<image_view> another_empty_view
  = std::nullopt;
full_view.has_value();  //true
empty_view.has_value(); //false
if (full_view) { this_works(); }
my_view = full_view.value();
my_view = *full_view;
my_view = empty_view.value(); //throws
bad_optional_access
my_view = *empty_view; //undefined behaviour
```

Listing 1

```
std::optional<image_view> get_cute_cat
(image_view img) {
    auto cropped = find_cat(img);
    if (!cropped) {
      return std::nullopt;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
      return std::nullopt;
    }
    auto with_sparkles =
      make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
      return std::nullopt;
    }
    return
      add_rainbow(make_smaller(*with_sparkles));
}
```

Listing 2

The most common operations on a **std::optional** are shown in Listing 1.

Now we're ready to use our **find_cat** function along with some other friends from our library to make embarrassingly adorable pictures of cats (see Listing 2).

Well this is… okay. The user is made to explicitly handle what happens in case of an error, so they can't forget about it, which is good. But there are two issues with this:

Simon Brand Simon is a GPGPU toolchain developer at Codeplay Software in Edinburgh. He turns into a metaprogramming fiend every full moon, when he can be found bringing compilers to their knees with template errors and writing posts for his blog at blog.tartanllama.xyz. Contact Simon at simonrbrand@gmail.com

```
std::expected<image_view,error_code> full_view =
  my_view;
std::expected<image_view,error_code> empty_view =
  std::unexpected(that_is_a_dog);
full_view.has_value();  //true
empty_view.has_value(); //false
if (full_view) { this_works(); }
my_view = full_view.value();
my_view = *full_view;
my_view = empty_view.value(); //throws
bad_expected_access
my_view = *empty_view; //undefined behaviour
auto code = empty_view.error();
auto oh_no = full_view.error(); //undefined
                                //behaviour
```

Listing 3

1. There's no information about why the operations failed.
2. There's too much noise; error handling dominates the logic of the code.

I'll address these two points in turn.

Why did something fail?

std::optional is great for expressing that some operation produced no value, but it gives us no information to help us understand why this occurred; we're left to use whatever context we have available, or (God forbid) output parameters. What we want is a type which either contains a value, or contains some information about why the value isn't there. This is called **std::expected**.

Now don't go rushing off to cppreference to find out about **std::expected**; you won't find it there yet, because it's currently a standards proposal [P0323r3] rather than a part of C++ proper. However, its interface follows **std::optional** pretty closely, so you already understand most of it. Listing 3 shows the most common operations.

With **std::expected** our code might look like Listing 4.

Now when we call **get_cute_cat** and don't get a lovely image back, we have some useful information to report to the user as to why we got into this situation.

Noisy error handling

Unfortunately, with both the **std::optional** and **std::expected** versions, there's still a lot of noise. This is a disappointing solution to handling disappointments. It's also the limit of what C++17's **std::optional** and the most recent proposed **std::expected** give us.

What we really want is a way to express the operations we want to carry out while pushing the disappointment handling off to the side. As is becoming increasingly trendy in the world of C++, we'll look to the world of functional programming for help. In this case, the help comes in the form of what I'll call **map** and **and_then**.

```
std::expected<image_view, error_code>
  get_cute_cat (image_view img) {
    auto cropped = find_cat(img);p
    if (!cropped) {
      return no_cat_found;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
      return cannot_see_neck;
    }
    auto with_sparkles =
      make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
      return cat_has_eyes_shut;
    }
    return
      add_rainbow(make_smaller(*with_sparkles));
}
```

Listing 4

Improvements upon improvements

As C++ programmers, we're constantly finding new ways to leverage the power of the language to make expressive libraries, thus improving the quality of the code we write day to day. Let's apply this to **std::optional** and **std::expected**. They deserve it.

With these two functions, we've successfully pushed the error handling off to the side, allowing us to express a series of operations which may fail without interrupting the flow of logic to test an optional.

If we have some **std::optional** and we want to carry out some operation on it if and only if there's a value stored, then we can use **map**:

```
widget do_thing (const widget&);
std::optional<widget> result =
  maybe_get_widget().map(do_thing);
auto result = maybe_get_widget().map(do_thing);
  //or with auto
```

This code is roughly equivalent to:

```
widget do_thing (const widget&);
auto opt_widget = maybe_get_widget();
if (opt_widget) {
  widget result = do_thing(*opt_widget);
}
```

If we want to carry out some operation which could itself fail then we can use **and_then**:

```
std::optional<widget> maybe_do_thing
  (const widget&);
std::optional<widget> result =
  maybe_get_widget().and_then(maybe_do_thing);
auto result =
  maybe_get_widget().and_then(maybe_do_thing);
    //or with auto
```

This code is roughly equivalent to:

```
std::optional<widget> maybe_do_thing
  (const widget&);
auto opt_widget = maybe_get_widget();
if (opt_widget) {
  std::optional<widget> result =
    maybe_do_thing(*opt_widget);
}
```

and_then and **map** for **expected** acts in much the same way an for **optional**: if there is an expected value then the given function will be called with that value, otherwise the stored unexpected value will be returned. Additionally, we could add a **map_error** function which allows mapping functions over unexpected values.

The real power of these functions comes when we begin to chain operations together. Let's look at that original **get_cute_cat** implementation again in Listing 2.

With map and **and_then**, our code transforms into this:

```
tl::optional<image_view> get_cute_cat
  (image_view img) {
    return crop_to_cat(img)
      .and_then(add_bow_tie)
      .and_then(make_eyes_sparkle)
      .map(make_smaller)
      .map(add_rainbow);
}
```

With these two functions we've successfully pushed the error handling off to the side, allowing us to express a series of operations which may fail without interrupting the flow of logic to test an **optional**. For more discussion about this code and the equivalent exception-based code, I'd recommend reading Vittorio Romeo's 'Why choose sum types over exceptions?' article [Romeo17].

A theoretical aside

I didn't make up **map** and **and_then** off the top of my head; other languages have had equivalent features for a long time, and the theoretical concepts are common subjects in Category Theory [Milewski14].

I won't attempt to explain all the relevant concepts in this post, as others have done it far better than I could. The basic idea is that **map** comes from the concept of a *functor*, and **and_then** comes from *monads*. These two functions are called **fmap** and **>>=** (bind) in Haskell. The best description of these concepts which I have read is 'Functors, Applicatives' And Monads In Pictures' by Aditya Bhargava [Bhargava13]. Give it a read if you'd like to learn more about these ideas.

A note on overload sets

One use-case which is annoyingly verbose is passing overloaded functions to **map** or **and_then**. For example:

```
int foo (int);
tl::optional<int> o;
o.map(foo);
```

The above code works fine. But as soon as we add another overload to **foo**:

```
int foo (int);
int foo (double);
tl::optional<int> o;
o.map(foo);
```

then it fails to compile, complaining that template arguments couldn't be inferred.

One solution for this is to use a generic lambda:

```
tl::optional<int> o;
o.map([](auto x){return foo(x);});
```

Another is a **LIFT** macro:

```
#define FWD(...) \
 std::forward<decltype(__VA_ARGS__)>(__VA_ARGS__)
#define LIFT(f) [](auto&&... xs) \
 noexcept(noexcept(f(FWD(xs)...))) -> \
 decltype(f(FWD(xs)...)) \
 { return f(FWD(xs)...); }
tl::optional<int> o;
o.map(LIFT(foo));
```

Personally I hope to see overload set lifting get into the standard so that we don't need to bother with the above solutions.

Current status

Maybe I've persuaded you that these extensions to **std::optional** and **std::expected** are useful and you would like to use them in your code. Fortunately I have written implementations of both with the extensions shown in this post, among others. **tl::optional** [GH1] and **tl::expected** [GH2] are on GitHub as single-header libraries under the CC0 [CC] license, so they should be easy to integrate with projects new and old.

As far as the standard goes, there are a few avenues being entertained for adding this functionality. I have a proposal [P0798r0] to extend **std::optional** with new member functions. Vicente Escribá has a proposal [P0650r1] for a generalised monadic interface for C++. Niall Douglas' **operator try()** paper [P0779r0] suggests an analogue to Rust's try! macro [Rust] for removing some of the boilerplate associated with this style of programming. It turns out that you can use coroutines [GH3] for doing this stuff, although my gut feeling puts this more to the 'abuse' end of the spectrum. I'd also be interested in evaluating how Ranges [N4685] could be leveraged for these goals.

Ultimately I don't care how we achieve this as a community so long as we have some standardised solution available. As C++ programmers we're constantly finding new ways to leverage the power of the language to make expressive libraries, thus improving the quality of the code we write day to day. Let's apply this to **std::optional** and **std::expected**. They deserve it. ■

References

[Bhargava13] http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

[CC] https://creativecommons.org/share-your-work/public-domain/cc0/

[cppreference] http://en.cppreference.com/w/cpp/utility/optional

[Crowl15] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0157r0.html

[Douglas17] Niall Douglas (2017) 'Mongrel Monads, Dirty, Dirty, Dirty', ACCU 2017: https://www.youtube.com/watch?v=XVofgKH-uu4

[GH1] https://github.com/TartanLlama/optional

[GH2] https://github.com/TartanLlama/expected

[GH3] https://github.com/toby-allsopp/coroutine_monad

[Halder16] Deb Haldar (2016) 'Top 15 C++ Exception handling mistakes and how to avoid them' http://www.acodersjourney.com/2016/08/top-15-c-exception-handling-mistakes-avoid/

[Kirk15] Shane Kirk (2015) 'C++ Exceptions: The Good, The Bad, And The Ugly' http://www.shanekirk.com/2015/06/c-exceptions-the-good-the-bad-and-the-ugly/

[Milewski14] Bartosz Milewski (2014) https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/

[MusingMortoray12] 'Everything wrong with exceptions' (2012) https://mortoray.com/2012/04/02/everything-wrong-with-exceptions/

[N4685] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4685.pdf

[P0323r3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r3.pdf

[P0650r1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0650r1.pdf

[P0779r0] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0779r0.pdf

[P0798r0] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0798r0.html

[Romeo17] Vittorio Romeo (2017) 'Why choose sum types over exceptions?' https://vittorioromeo.info/index/blog/adts_over_exceptions.html

[Rust] https://doc.rust-lang.org/1.9.0/std/macro.try!.html

[Stackoverflow] 'Why is exception handling bad?' (https://stackoverflow.com/questions/1736146/why-is-exception-handling-bad) and 'Are Exceptions in C++ really slow' (https://stackoverflow.com/questions/13835817/are-exceptions-in-c-really-slow)

[Stroustrup18] Bjarne Stroustrup and Herb Sutter (eds) https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#e3-use-exceptions-for-error-handling-only