overload 141

OCTOBER 2017 £3

The Historical Context of Technology

Certainty is elusive. We investigate a historical context for foundational thinking.

A magazine of ACCU

ISSN: 1354-3172

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe

OVERLOAD 141

October 2017

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

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 142 should be submitted by 1st November 2017 and those for Overload 143 by 1st January 2018.

Overload is a publication of the ACCU

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

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.

This way up!

Directions in computing can be very confusing. Frances Buontempo wonders if we know our right from left.

Following instructions can be very difficult. For example, writing an editorial every two months with no idea what to write about, simply having to fill two pages, is a regular challenge. Could I get someone to draw me a cartoon strip? (Budding artists get in touch, please). Can I use a gigantic font? (Apparently not.) Lacking any direction should be liberating, but can be confounding. Even if you are given directions, things can go awry. Have you ever told someone to go left at the next turn, only to see them indicate right? Far too often a cry of, "No, left!" meets with "Yes, I'm going left!!" The simplest solution is to try, "No, the other left" instead. That has worked on several occasions for me.

Spatial awareness is vital in many situations. Even sitting at your desk all day, you need to be able to reach out and grasp your mug of coffee – assuming you are caffeine-powered like many programmers – or move your fingers around a keyboard. Practising can help you improve; you learn where the home keys are and how much to move to hit the keys you need, rather like practising scales when you learn an instrument. Some keys are better hit with the right hand; others with the left hand. I constantly have to remind myself there are two shift and control keys. Note to self; for combinations like Ctrl-A, it makes much more sense to use the right hand Crtl key with the A.

At other times, spatial words are used arbitrarily: a binary operator can take an lhs and a rhs – left hand side and right hand side – stemming from how the code is written. The operator could be seen as taking a first and second operand, avoiding the tempting and common abbreviations lhs and rhs. Whether infix notation 3 + 4 or something like 3.operator+(4), you have a left and right operand. If you start chaining operators – 2*3+4 – you still have a left and right, though to parse the expression you will build an expression tree. If you evaluate this in Q you will get 14 [Q operators]. You may have been expecting 10, (2*3)+4, but Q evaluates from right to left. Unconventional; you need braces to ensure it follows your instincts. What about x < y < z? In some languages this would compare the result of x < y, a Boolean, with z, which may not have been the intention; you need to spell out (x < y) and (y < z). However, in Python, the operators can be chained [Python operator]. For almost all data types, you would expect transitivity to follow: that x < z. This might not happen, either due to bugs or by design, depending on the definition of less than. Unconventional approaches can lead to surprises. Of course, conventions and instincts are built on top of what has happened before. Challenging norms leads to new and interesting ideas. New branches of mathematics often sprout this way. As Kevlin Henney recently pointed out, "Tradition [noun]; A practice or belief that is repeated and passed on without rationale, except that 'We've always done it this way.'" [Henney]

Sticking to conventions and repeating the way you do things is not always a bad idea. Typing with your left hand on the left of a keyboard and your right on the right is a good thing. Doing touch typing drills can be a good thing, if like me you frequently end up with a mess of typos when you write. Building up a muscle memory makes so many things become instinctive. Rather than sitting at your desk all day, it is good to get some physical exercise. Going for a short walk round the block counts, though some people choose more formal sets of exercises, possibly in a gym. Calisthenics are a way of using your own body's weight to exercise, without needing kit, so you can do this in a gym or outdoors. Think sit ups or similar. If you search, you will find lists of exercises to try that work out your whole body, claiming to improve balance, muscle tone and the rest. If you'd rather sit at your desk and code, try out Object Calisthenics by Jeff Bay. These are included in the Thoughtworks anthology [Thoughtworks] and can be found online [Object Calisthenics]. He points out that many object-oriented examples "encourage poor OO design in the name of performance or simple weight of history". Tradition, as ever. They consist of nine rules of thumb to nudge your OO code towards seven code qualities; cohesion, loose coupling, no redundancy, encapsulation, testability, readability, and focus. Rule six is "Don't abbreviate", or "Do not abbreviate". So, lhs and rhs mentioned above are bad parameter names, and "Ctrl" was just careless. I apologise. In terms of code, I'm sure you can think of many examples of abbreviations such as `getNoChildren` rather than `getNumberOfChildren` or even `numberOfChildren`. Of course, rule nine says no getters, so the problem is nipped in the bud.

Left and right seem like obvious ideas, but I recall lots of problems trying to remember which was which when I was a small child. Parents often resort to writing on shoes, gloves, etc. to help reinforce the words. Can you recall how you learnt which side was which? Do you remember when you discovered whether you were left or right handed? Can you write with both hands? Since more people are right handed, many things are set up for right handed people. To scan a travel pass you usually hold it to your right, unless you get on a bus in the UK. You may have spotted people scanning the card on the left and then walking into a closed barrier. Right-handed scissors don't work if you use them in your left hand. Other setups are purely driven by convention, including driving; the gear stick in a car is by your left if you have a right hand drive car, and by your right if you have a left hand drive car. The hand you use depends on which side of the road you are supposed to be driving on. One will seem more natural depending on your experience.

The idea of handedness, or chirality, applies to molecules as well as people. Thalidomide, a drug produced to combat morning sickness, caused malformation of limbs in the babies. I am not a chemist, but I understand small batches were tested which appeared to be fine, but when enterprise sized production batches were produced, slightly different 'enantiomers' or optical isomer of the chemical were manufactured. One is a mirror image of the other; they contain the same elements arranged similarly. The only difference is the chirality. The molecules can then slot into different places and cause different outcomes in vivo. A warning

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

against enterprise-grade endeavours? Maybe, maybe not. Turning to physics, when Feynman discusses anti-matter in *Six Not So Easy Pieces* [Feynman] he identifies the chiral opposite of normal matter as antimatter. Starting with beta-decay, he considers reflective symmetry in nature and recounts an experiment which shows decaying cobalt atoms emitting electrons which tend to go up, under some specific conditions. You could describe this experimental setup to a Martian, and then have a way to describe right and left (or up and down) without pointing. A genuine problem, assuming you can speak Martian, and find a Martian to talk to. How do you describe left or right? In words, without pointing. Cobalt; check. Martian; check. So far, so good. However, the 'broken symmetry' – that sometimes electrons tend to spin left, or go up, with no good reason – is curious. He encourages you to imagine antimatter, which annihilates matter on contact. If the Martian is made of antimatter and you talk him through the cobalt experiment to communicate left and right, and arrange to meet, what will happen? His anti-matter cobalt will send electrons down, so his left and right will be the other way round to yours. Feynman says,

> What would happen when, after much conversation back and forth, we each have taught the other to make space ships and we meet halfway in empty space? We have instructed each other on our traditions, and so forth, and the two of us come rushing out to shake hands. Well, if he puts out his left hand, watch out! [Feynman online]

Perhaps you can't describe left and right in exact terms. Gisin gives a short, though maths-heavy discussion of the same topic [Gisin], called 'Quantum gloves', starting "Assume that two distance partners like to compare the chiralities of their Cartesian reference frames." It talks through frames of reference and information encoding, and leaves me none-the-wiser as to which way is up; highly recommended if you like your maths, though. Pointing is sometimes the only way. Of course, that is something of a human convention. Some animals can understand it, but a cat will often just look at your finger rather than what you are pointing to.

Arbitrary assignments, such as left or right-handed applied to molecules, apply in computing too. Algorithms, as well as design, processing, investing or parsing can be described as top down or bottom up. The internet tells me that top down parsing starts at the "highest level of a parse tree" [Wikipedia]. Without following the link, I am certain the highest part of this tree will be a root. Trees in computing are almost always upside down. Top-down starts with the big picture moving on to details, while bottom up figures out the details, and pieces the parts together to make the big picture or final answer. Bottom-up can also mean capsizing a yacht, though figuring out which way up a boat should go should be straightforward enough, even for a Martian. Markup and markdown is a different matter. I presume we draw the root at the top of a page to add the details as we go down since we tend to start writing at the top of a page. This leads to another source of confusion, at least for me. Many drawing packages are based on printing to a screen, and tends to have 0 as the top of the screen. In contrast, 0 is at the bottom when you plot data on axes; for many plotting packages, going up gives bigger numbers. Watch out for different idioms, conventions and scales as you switch contexts.

If you are told to turn something round, what do you do? Does 'flip' suggest something different to 'rotate'? Can I describe what I mean clearly using only words? Many software applications come with some form of documentation, frequently of varying quality, and part of that often includes some form of diagrams. I personally tend to find drawing an architecture diagram of a complicated system by hand more useful than attempting to decipher a diagram with hundreds of parts. Where do you look first? A simple diagram, with a couple of blobs indicating different parts of the system and one or two arrows showing connections, either as data flow or processing can be helpful. More than that is often confusing. Making clear diagrams is as much of skill as writing well. You probably don't need a detailed map of your software landscape, at least initially, though a spot clearly marked 'Here be dragons' is useful. There is no shame in a simple sketch on a white board, if you have a whiteboard. Failing that a hand-drawn sketch on paper will do.

Some of the confusion we have considered is inherent in language, while some is through ambiguities, careless abbreviations, or assumptions about experience. Some of the issues arise from pedantry or precision. One language will parse a statement left to right, while another might go right to left, or special case operators. Humans do the same too. Children can have difficulty with ambiguous statements: "the man fished from the bank" [Rowland]… Some children get the hang of this type of statement more quickly than others. Sometimes autism is blamed, though this book suggests autistic children's difficulties in communication are not purely down to autism. The author cites various studies including 'Barking up the wrong tree?' [Norbury] of language impairment in autistic children, considering the use of metaphor and simile comprehension in a group of children with various degrees of language impairment and autistic symptoms. The study concludes that children with autism who struggled with the metaphors also had language impairment, and were indistinguishable from those without autism. It concludes that "a number of top-down (contextual processing, world knowledge, and experience) and bottom-up (semantic analysis) processes that work synergistically to arrive at metaphor understanding." Computer programmers are often claimed to be on the autistic spectrum or have Asperger's syndrome [Atwood]. I suspect much of the claimed evidence is inaccurate; being absorbed in something you find interesting is not the same as having autism, yet some ways of measuring autism do hint at slight differences between different groups, for example the autism-spectrum quotient [Baron-Cohen et al]. If you can't tell your left from right, never remember which is bottom up or top down, find ambiguous instructions confusing, you are not alone. If it all gets too much, step away from the keyboard and get some exercise. Or practise scales. Loudly. Or overload some operators in unconventional ways and write it up for *Overload*. And don't forget, as Tony Hoare once said,

> You cannot teach beginners top-down programming, because they don't know which end is up.

References

[Atwood] https://blog.codinghorror.com/software-developers-and-aspergers-syndrome/

[Baron-Cohen et al] The Autism-Spectrum Quotient (AQ): Evidence from Asperger Syndrome/High-Functioning Autism, Males and Females, Scientists and Mathematicians, *Journal of Autism and Developmental Disorders*, February 2001, Volume 31, Issue 1, pp 5-17 https://link.springer.com/article/10.1023/A:1005653411471

[Feynman] *Six Not-so-easy Pieces: Einstein's Relativity, Symmetry and Space-time*, first published in 1964 by Addison-Wesley but reissued in 1990s.

[Feynman online] http://www.feynmanlectures.caltech.edu/I_52.html

[Gisin] https://arxiv.org/pdf/quant-ph/0408095.pdf

[Henney] https://twitter.com/KevlinHenney/status/900598832131698688

[Norbury] 'Barking up the wrong tree? Lexical ambiguity resolution in children with language impairments and autistic spectrum disorders.' *J Exp Child Psychol*. 2005 Feb;90(2):142-71.
See https://www.ncbi.nlm.nih.gov/pubmed/15683860 or find online at http://www.pc.rhul.ac.uk/sites/lilac/new_site/wp-content/uploads/2010/04/metaphor.pdf

[Object Calisthenics] https://www.cs.helsinki.fi/u/luontola/tdd-2009/ext/ObjectCalisthenics.pdf

[Python operator] https://docs.python.org/2/reference/expressions.html#comparisons

[Q operators] http://code.kx.com/q4m3/4_Operators/

[Rowland] *Understanding Child Language Acquisition* by Caroline Rowland via https://books.google.co.uk/books?id=cVizAQAAQBAJ

[Thoughtworks] https://pragprog.com/book/twa/thoughtworks-anthology

[Wikipedia] https://en.wikipedia.org/wiki/Top-down_(disambiguation)

Letters to the Editor

Silas has some comments on Deák Ferenc's recent article:

Regarding Deák Ferenc's interesting article in *Overload* #135 (October 2016), with code at https://github.com/fritzone/obfy, on the generation of more-obfuscated binaries using C++ constructs, I think a better application is that of decoding obfuscated data rather than a simple license check.

Attackers tend to go for the weakest link. If you make the **`check_license`** function very obfuscated, then sooner or later the weakest link will be, not the license check itself, but the call to it. The attacker finds the code that calls it and jumps over that call, or replaces the first few bytes of the compiled **`check_license`** with code to return true.

Stripping out the name **`check_license`** might help, but from your code it looks like reverse-engineering the location of **`check_license`** (or a call to it) will become the simplest attack, unless we're in a context where the attacker has read-only access to the code and the only plausible attack is a counterfeit license file (but that's not the normal context if you're shipping out proprietary software to an end user).

Therefore, I suggest this code could more appropriately be used in programs that need to ship with an obfuscated copy of proprietary data. For example, consider a dictionary program whose publishers wish to allow the end-user to look up individual words for display on screen, but not to copy out the entire dictionary and print their own (at least, not without going to the trouble of manually writing it down from the screen). Such a publisher might wish to store their dictionary entries encrypted, with the decryption algorithm obfuscated using your code. Unless the attacker can figure out how the decryption algorithm works, they'd be restricted to using the provided program to display the entries, which is what the publisher wants.

For the dictionary example, there might still be other weak points: it might be possible to feed keystrokes to the program, causing it to display each entry in turn, and automatically copy the text off the screen. The program might try to protect against this by limiting the speed at which entries can be displayed and/or the total number of entries per session. These limits could then be targeted in an attack. And so on. But this is all more difficult than bypassing a yes/no license check.

Of course, if enough publishers started using Ferenc's code, then sooner or later somebody might try to write a decompiler for it. Such a decompiler would be dependent on the C++ compiler that had been used, and would be very difficult indeed to write, but once written could be used to attack many products. The publishers might however be able to stay ahead for a while by randomising the exact set of optimisations they allow their compiler to use. (The use of volatile prohibits pre-computation optimisations, but other types of optimisation could be switched on and off for more variations.)

Thanks.

Silas S. Brown

Ferenc replies:

Hi,

I would like to thank Silas for his constructive comments. The suggestions he made are valid: indeed, the weakest link in license checking during the application's lifetime is the actual call to the license checking routine. The attacker can always 'NOP' out the calls to any specific routine in the application, once that routine is identified as the one checking the license, or just patch the method to return a valid license.

To mitigate this patching, I could recommend that several routines perform the task of checking the license, each of them written with a different sub-set of the obfuscation framework to generate different code. Also, different parts of the application could perform license checking in seemingly unrelated scenarios (for example, opening a dialog box could trigger the license check, as well as saving the current progress) and react accordingly. This would certainly lead to an increase of the delivered binary, however.

The call to license checking, however, is a different problem entirely. Gone are the DOS days when we could (easily) dynamically patch our executable in memory by decrypting a sequence of binary commands and jumping to them to perform operations which the disassembler cannot see, or simply by constructing a new sequence of commands by jumping into the middle of a carefully crafted binary opcode sequence, thus making debugging and disassembly a real nightmare (both of these techniques were widely used in viruses 20 years ago).

Without a further research, right now I'm only aware of a few ways (some standard and some non-standard) to call methods in an indirect way, but in the end they all end up in the generated binary code. For example, one could store the addresses of several 'facade' methods (which call the license checking method) in a structure, and randomly select one from them in order to confuse the wanna-be attacker (of course, more than one license checking methods can be there too). Another (pretty hackish, non-portable, non-standard and definitely not recommended) method is to carefully engineer a local stack overflow which will end up in the license checking method, without entirely destroying the stack of the application (again: not recommended).

In order to circumvent the Data Execution Prevention policy of some operating systems, it is possible to execute constructed binary code by creating specific memory areas which will be marked as executable, where we can generate code outside of the binary and make the call to the license checking algorithm invisible to the stored binary, thus nothing to patch. But this topic is so wide that it deserves a dedicated article, and it was outside the scope of the 'Obfuscation Frameworks' article.

Regarding the usage of code the way you suggested (ie: as a data storage component): indeed, it is a very appropriate use case for this framework, so please feel free to experiment with it and let me know the results.

Thank you again for your constructive ideas,

Kind regards,

Ferenc

Silas also noticed an error in the Editorial in Overload 135:

I just noticed a typo in *Overload* 135's editorial.

It says A⇒B is identical to !A⇒!B.

This is incorrect. If A implies B, that does not mean lack of A implies lack of B. (Fire implies smoke, but if there's no fire there might still be smoke from dry ice or something.)

Instead, A⇒B is identical to !A | B.

(Either there's no fire, or there's fire and smoke with it, but we're not saying anything about the value of smoke when there's no fire.)

Silas S Brown

And Fran agrees that Silas is right...

As you say, A⇒B is the same as !A | B

I meant !B⇒!A (I think).

A⇒B	**!B⇒!A**
0 1 0	1 1 1
0 1 1	0 1 1
1 0 0	1 0 0
1 1 1	0 1 0

Marking Benches

Too slow! Russel Winder reminds us that benchmarking code performance is a complicated issue.

In the article 'Mean Properties' [Winder17] I, hopefully humorously, but definitely contentiously, stated in a footnote:

> Of course we have no real data on this hypothesis without undertaking some sensible benchmarking activity. Which we will not be doing in this article since it is far too much effort.

It may have been too much effort in the context of that article for that article, but clearly measuring performance is extremely important when performance of code execution becomes an issue.

A short rant

Far too regularly I, and I am sure all of you reading this, hear people saying things like "but this code is much faster than that code" and leave it at that as though saying something is the case makes it true[1]. If you are involved in a conversation with someone making these sort of sweeping judgements, you might want to ask them to rephrase. If the person rewords to something along the lines of "I believe that, on appropriately measuring, this code will execute faster than that code in the measurement context" then you know they initially just mis-phrased things in the heat of the moment, and actually have a reasonable view of software performance. If the person persists with "but this code is much faster than that code, just look at the code" then you may just want to shun the person publicly till they re-educate themselves.

Repairing a previous mis-phrasing

In [Winder2017] I stated "…this code is likely to be much slower than using NumPy." Not an obviously outrageous phrase since it is not claiming a performance fact, but it is raising a performance question, a question that really should be answered.

Questions such as this, as with any hypotheses in scientific method, lead to setting up an experiment to obtain data, experiments that are reproducible and with good statistical backing. In essence this means running the codes a goodly number of times in consistent and stable settings. Doing this sort of experimentation manually is, at least for all programmers I know, seriously tedious, leading either to short-cuts or no experimentation. There have to be frameworks to make this sort of experimentation easy.

The Python approach

In [Winder2017] I used [pytest] and [Hypothesis] as frameworks for writing tests for correctness of the implementation code. The goal was to emphasise property-based over example-based testing for correctness. pytest also has support for benchmarking, i.e. checking the performance of code. It is not built in to pytest as distributed, but is a trivially installable plugin [pytest-benchmark].

1. OK, as from the second half of 2016 in the new 'post-truth' world, the norm in society in general is for things stated to be true to be true, but we must not let software development and programming be polluted by this unscientific posturing.

pytest-benchmark provides a fixture to be used with test functions to create benchmark code. Whilst benchmarking can (and sometimes is) intermingled with correctness testing code, it is far more normal and idiomatic to separate correctness testing and benchmarking.

As an example let us address the code from the previous article. Here we will just use the Numpy version:

```
# python_numpy.py
import numpy
mean = numpy.mean
```

and the corrected pure Python version:

```
#python_pure_corrected.py
def mean(data):
  if len(data) == 0:
    raise ValueError \
      ('Cannot take mean of no data.')
  return sum(data) / len(data)
```

omitting the original pure Python version. The hypothesis we are testing is that: the Numpy version is faster than the pure Python version.

So how to collect data? We write what appear to be pytest tests but use the benchmark fixture provided by pytest-benchmarking to create benchmark executions of the functions instead of the correctness testing functions we wrote for correctness testing.

```
import pytest

from random import random

from python_numpy import mean as mean_numpy
from python_pure_corrected import mean as \
  mean_pure

data = [random() for _ in range(0, 100000)]

def test_mean_numpy(benchmark):
  benchmark(mean_numpy, data)

def test_mean_pure(benchmark):
  benchmark(mean_pure, data)
```

Listing 1

Russel Winder Ex-theoretical physicist, ex-UNIX system programmer, ex-academic, ex-independent consultant, ex-analyst, ex-author, ex-expert witness, and ex-trainer, not yet an ex-human being. Mostly now interested in the ACCU conference and writing DVB-T and DAB software. Still interested in programming, programming languages and build. May yet release the next version of GPars. See https://www.russel.org.uk

It appears that we can conclude that the Numpy version is 5-ish times slower given the mean and median values of the time taken to execute the function

Name	test_mean_pure	time in µs	test_mean_numpy	time in µs
Min	856.063	1	4461.969	5.21
Max	1079.325	1	4731.414	4.38
Mean	872.459	1	4476.2517	5.13
StdDev	8.2584	1	22.6862	2.75
Median	871.892	1	4472.543	5.13
IQR	1.4317	1	9.9422	6.94
Outliers[*]	45;342		7;10	
Rounds	1055		165	
Iterations	1		1	

*. Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.

Table 1

In Listing 1, pytest is imported on the assumption that the pytest-benchmark plugin is available. the module random is imported as a dataset has to be generated: data is a random collection of numbers in the range (0.0, 1.0), here 100,000 items are in the dataset. Since both mean function implementation are called mean, the alias feature of the import statement is used to distinguish the Numpy implementation and the pure Python implementation. Then there are the two test functions, which use the benchmark fixture to run the benchmarks of the functions using the dataset provided. (The output has been tabulated – see Table 1 – which omits some information but makes it much easier to read.)

Well, that was unexpected[2]. It appears that we can conclude that the Numpy version is 5-ish times slower given the mean and median values of the time taken to execute the function.

Oh come on, this cannot be...

So we have actual data that pure Python is much faster at calculating means on datasets than Numpy mean. It's reproducible, and thus incontrovertible. The original hypothesis is disproved[3].

Yet our expectation remains that pure Python is interpreted Python bytecodes, and thus slow, whereas Numpy is implemented in C and thus fast. It is unbelievable, yet provably true, that pure Python is faster than Numpy.

2. No-one expects the Spanish Inquisition.
   It's a Python thing, a Monty Python thing. https://en.wikipedia.org/wiki/The_Spanish_Inquisition_(Monty_Python) https://www.youtube.com/watch?v=7WJXHY2OXGE
3. Technically we need to do some analysis of variance and significance testing to make this statement. However, for the purposes of this article, I shall gloss over this point, even though it pains me to do so: ANOVA, F-tests and t-tests are so much fun.

Ah, but… the input data was a Python list, Numpy works on Numpy arrays. Mayhap we can unpack this idea with a new benchmark. Listing 2 adds constructing a Numpy array form of the data and then running the Numpy mean function on the array. Note we keep the two original tests (see Table 2).

OK, now this looks more like it: pure Python on a Python list is 8-ish times slower than Numpy on a Numpy array. Numpy mean on Python list remains as slow as previously.

We have a new Quasi-Hypothesis

Passing Python data structures to Numpy functions is a really bad idea, thus the mean implementation of the Numpy version is a very poor implementation.

By making a Numpy function appear to be a pure Python function, the conversion of the Python list to a Numpy array is hidden. This conversion clearly has performance implications and so must be made explicit in performance sensitive code. Essentially, when using Numpy, always use Numpy data structures: do not make it convert Python ones.

Proper benchmarking brought this to light. Now to change all the code to avoid the problem. And then benchmark again to make sure the suppositions and hypotheses are vindicated.

Endnote

I have glossed over many important points of data collection, samples, statistics, and normality, in this article – the goal being to enthuse people

```
import pytest

from numpy import array
from random import random

from python_numpy import mean as mean_numpy
from python_pure_corrected import mean as \
  mean_pure

data = [random() for _ in range(0, 100000)]
data_as_array = array(data)

def test_mean_numpy_list(benchmark):
  benchmark(mean_numpy, data)
def test_mean_numpy_array(benchmark):
  benchmark(mean_numpy, data_as_array)
def test_mean_pure(benchmark):
  benchmark(mean_pure, data)
```

Listing 2

pure Python on a Python list is 8-ish times slower than Numpy on a Numpy array

into using benchmarking frameworks for gathering real, reproducible data to back up claims of performance. Using a framework such as pytestbenchmark has some assumptions built in that arguably may not be formally valid, and so the results presented may not be ‘correct’ in a formal statistical sense. However, the framework gives us data on our code’s performance that is valid enough for us to make deductions, inferences and corrections. Thus it is an incredibly useful tool for ‘rough and ready’ performance checking.

Oh that all programming languages had such useful benchmarking frameworks. Some have, e.g. Python and Java. This is an interesting article on benchmarking frameworks for C++ [Filipek16]. ■

References

[Filipek16] Micro benchmarking libraries for C++. http://www.bfilipek.com/2016/01/microbenchmarking-libraries-for-c.html

[Hypothesis] http://hypothesis.works/

[pytest] http://pytest.org/

[pytest-benchmark] http://pytest-benchmark.readthedocs.io/en/stable/ https://github.com/ionelmc/pytest-benchmark

[Winder17] ‘Mean Properties’, *Overload* 137, February 2017. https://accu.org/var/uploads/journals/Overload137.pdf https://accu.org/index.php/journals/2340

Name	test_mean_numpy_array	time in µs	test_mean_pure	time in µs	test_mean_numpy_list	time in µs
Min	100.484	1	841.832	8.38	4572.266	45.5
Max	153.946	1	945.041	6.14	5019.503	32.61
Mean	101.8516	1	848.8424	8.33	4626.392	45.42
StdDev	3.5793	1	9.227	2.58	49.3561	13.79
Median	101.004	1	846.704	8.38	4623.92	45.78
IRQ	0.406	1	0.9293	2.29	44.3635	109.28
Outliers*	324;417		62;308		26;5	
Rounds	3314		1011		175	
Iterations	1		1		1	

*. Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.

Table 2

The Historical Context of Technology

Certainty is elusive. Charles Tolman provides a historical context for foundational thinking.

An enlightening aspect that surprisingly pertains to the issue of software design is the philosophical history that has led us to our current technological society.

Looking back, we can see some origins in the Thirty Years War [Wikipedia_01] that took place between 1618 and 1648. Some commentators have drawn parallels with the impact of WW1 and WW2 between 1914 and 1945, saying that they could also be seen as a thirty year war [Toulmin92]. The Thirty Years War of 1618 was a terrible war over much of Europe that resulted in the death of a third of the German population. It was a religious war between Protestants and Catholics, i.e. one religion, two factions – and raised serious concerns about the subjectivity of religious faith and the human condition. It was this that brought the quest for certainty to a head. The underlying question was: How can we be certain of what is happening in the world around us? And for the faithful in the 1600s, how can we be certain of God's plan?

Descartes

It was during this time that René Descartes [Wikipedia_02] produced his 'Discourse on Method' in 1637. He was the father of analytical geometry and, of course, coined the famous phrase "I think, therefore I am". But this was predicated on the fact that we first doubt, thus the more correct phrase should be "I doubt, I think, therefore I am". He concluded that, because of our subjectivity, we cannot trust our senses and what they are telling us about the world, so he returned to the point of doubting. Since there was doubt, there must be a being that is doubting. This being, this 'I' who is doubting, is thinking about this so therefore I am thinking. Since I am thinking I must exist in order to do that thinking.

Because the church was looking for certainty and because Descartes was able to couch his thought in terms that they could accept, this provided the foundation for the Scientific Revolution. This was followed by the Industrial Revolution which has led us to our current modern technological society. It is interesting to consider the fact that all that we take for granted today represents the end of 300 years or so of work based on Descartes' philosophical premise: "I doubt, I think, therefore I am" where the aim was to try and eradicate subjectivity.

It is ironic that, although the aim was to be objective, his Cartesian coordinate system can be considered to be based on the structure of the human being! I stand up, and my head could be considered as the Y axis. I stretch my arms out to the side, there you have the X axis. I walk forward and there you have the Z axis.

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

This points to the difficulties that are implicit in the struggle to eradicate subjectivity – an objective (pun intended!) which I do not consider possible.

Kant

I usually refrain from mentioning Immanuel Kant [Wikipedia_03] since I am not a Kantian scholar, but his thinking has formed much of the basis of modern thought. He produced the *Critique of Pure Reason* in 1781, and there is one quote I wish to highlight here from his considerable body of work. He said that "The world in itself is unknowable". and this strengthened Descartes' approach of not trusting our senses. It has given our modern scientific and technological society the excuse to allow our thinking to run ahead of the phenomena of the world.

This activity may sound familiar if you think back to the 'Path of the Programmer' [Tolman16]. It is a characteristic of the Journeyman phase.

With regard to my previous workshop on imagination [Tolman14], an area dealing with educating our subjectivity, it is interesting to see that one commentator, Mark Johnson, has noted that Kant had difficulty with imagination – Johnson states that he was "not able to find a unified theory of imagination in Kant's writings" [Johnson87].

Goethe

The third person I want to mention, and the one I feel most drawn to, is Goethe [Wikipedia_04]. It was Goethe who raised the warning flag to say that there was a problem with the underlying philosophy and practice of the scientific method. He pointed out that there was too much over-hypothesizing and that the thinking was going ahead of the phenomena of the world. Observation was not being given enough time.

This should ring alarm bells for any programmer because it is exactly what happens when someone takes an undisciplined approach to debugging.

Goethe, however, was particularly interested in understanding the growth of plant life. He wrote the *Metamorphosis of Plants* in 1788 and identified two very important activities. The first one is Delicate Empiricism (or "Zarte Empirie" in German), i.e. carefully collecting the data, carefully observing the world without overly disturbing its processes. [Wahl05]

The second activity, which is what gave me the impetus to give my previous workshop on imagination in 2014, is Exact Sensorial Imagination. This is NOT fantasy, but exact, grounded imagination congruent with the observed phenomena. Goethe was trying to understand how plants grew and how their forms changed during growth.

For me this links to how software projects grow over time, as if they have a life of their own. A programmer needs to have a grasp of how the current software forms may change over time within such a context if they are to minimise future bugs.

The key difference between Descartes and Goethe is that Descartes was trying to *eradicate subjectivity* whereas Goethe was wanting to *educate subjectivity*.

The next important phase in philosophical thought is the advent of phenomenology in the 1900s [Wikipedia_05]. The realization that the process of coming to know something is crucial to, and as important as, the conclusion. Goethe is not considered a phenomenologist as he focused on specific phenomena rather than the philosophy behind what he was doing, but he definitely prefigured some of their ideas and so could be called a proto-phenomenologist.

We need to understand that phenomenology is a sea-change in philosophical thought. Here we are, living in a modern technological society based on 300 years of progress initiated by Descartes and his subject/object duality, and now the underlying foundational thinking has changed significantly.

The discipline of software development is in the forefront of trying to understand what this change of thinking means in practice, though it may not have realised it. We need to understand how we develop our ideas and we need to understand our own cognitive biases, the subject of Dr. Marian Petre's keynote 'Balancing Bias in Software Development' [Petre16]. The point here is that we can do a certain amount in teams but there is also some personal work to do in understanding our own learning processes.

There is a wonderful quote by Jenny Quillien who has written a summary of Christopher Alexander's *Nature of Order* books [Wikipedia_06]. She says in a preface:

> Wisdom tells us not to remain wedded to the products of thought but to court the process. [Quillien08]

I think this is a lovely way of putting it. The process needs courting, it has to be done carefully as with Goethe's Delicate Empiricism.

For those who wish to understand Goethe's work and the philosophical issues around phenomenology, a primary source is Henri Bortoft [Wikipedia_07]. His writing is very understandable, particularly his book *Taking Appearance Seriously* [Bortoft12] and he draws on the work of Gadamer [Wikipedia_08], one of the more recent phenomenologists. ■

References

[Bortoft12] Bortoft, Henri (2012) *Taking Appearance Seriously: The Dynamic Way of Seeing in Goethe and European Thought*. See http://www.florisbooks.co.uk/book/Henri-Bortoft/Taking+Appearance+Seriously/9780863159275

[Johnson87] Johnson, Mark (1987) *The Body in the Mind: The Bodily Basis of Meaning, Imagination and Reason*, published by University of Chicago Press
See also https://philosophy.uoregon.edu/profile/markj/

[Petre16] 'Balancing Bias in Software Development', talk delivered at the ACCU conference 2016. See https://accu.org/index.php/conferences/accu_conference_2016/accu2016_sessions#Balancing_Bias_in_Software_Development and http://mcs.open.ac.uk/mp8/

[Quillien08] Quillien, Jenny (2008) *Delight's Muse*, see http://www.culicidaepress.com/2010/11/09/quillien-delights-muse/

[Tolman14] Workshop delivered at the ACCU conference in 2014, https://charlestolman.com/2014/04/21/accu2014-workshop-imagination-in-software-development/

[Tolman16] Tolman, Charles (2016) 'The Path of the Programmer', talk at the ACCU 2016 conference. See also https://charlestolman.com/2016/05/01/accu2016-talk-on-software-architecture-design/

[Toulmin92] Toulmin, Stephen (1992) *Cosmopolis: The Hidden Agenda of Modernity*, ISBN 978-0226808383
See also https://en.wikipedia.org/wiki/Stephen_Toulmin

[Wahl05] Wahl, Daniel C. (2005) '"Zarte Empirie": Goethean Science as a Way of Knowing' *Janus Head* 8(1) pp 58–76 Trivium Publications, Amherst, NY. Available at http://www.janushead.org/8-1/wahl.pdf

[Wikipedia_01] 'Thiry Years' War' https://en.wikipedia.org/wiki/Thirty_Years'_War

[Wikipedia_02] https://en.wikipedia.org/wiki/Ren%C3%A9_Descartes

[Wikipedia_03] https://en.wikipedia.org/wiki/Immanuel_Kant

[Wikipedia_04] https://en.wikipedia.org/wiki/Johann_Wolfgang_von_Goethe

[Wikipedia_05] https://en.wikipedia.org/wiki/Phenomenology_%28philosophy%29

[Wikipedia_06] https://en.wikipedia.org/wiki/Christopher_Alexander

[Wikipedia_07] https://en.wikipedia.org/wiki/Henri_Bortoft

[Wikipedia_08] https://en.wikipedia.org/wiki/Hans-Georg_Gadamer

Courses:

Moving Up to Modern C++
An Introduction to C++11/14/17 for experienced C++ developers. Written by Leor Zolman.
3-day, 4-day and 5-day formats.

Effective C++
A 4-day "Best Practices" course written by Scott Meyers, based on his Legacy C++ book series. Updated by Leor Zolman with Modern C++ facilities.

An Effective Introduction to the STL
In-the-trenches indoctrination to the Standard Template Library. 4 days, intensive lab exercises, updated for Modern C++.

Live on-site C++ Training
by Leor Zolman

Mention ACCU and receive the U.S. training rate for any location in Europe!

www.bdsoft.com • bdsoftcontact@gmail.com • +1.978.664.4178

'Speedy Gonzales' Serializing (Re)Actors via Allocators

More speed! Sergey Ignatchenko completes his (Re)Actor allocator series with Part III.

Start the reactor. Free Mars…
~ Kuato from Total Recall

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.

As we briefly discussed in Part I of this mini-series [NoBugs17a], message-passing technologies such as (Re)Actors (a.k.a. Actors, Reactors, ad hoc FSMs, and event-driven programs) have numerous advantages - ranging from being debuggable (including post-factum production debugging), to providing better overall performance.

In [NoBugs17a] and [NoBugs17b], we discussed an approach to handling allocations for (Re)Actors, and then were able to achieve a safe dialect of C++ (that is, as long as we're following a set of well-defined local rules). Now, let's take a look at another task which can be facilitated by per-(Re)Actor allocators - specifically, at the task of serializing (Re)Actors that are later going to be de-serialized by the same executable. While one solution for this task was provided in [Ignatchenko-Ivanchykhin16], the proposed 'ultra-fast' serialization is rather cumbersome to maintain, and in most cases it can be beaten performance-wise by serializing at the allocator level.

#define (Re)Actors

To make this article self-contained and make sure that we're all on the same page with terminology, let's repeat the definition of our (Re)Actors from [NoBugs17a].

Let's name a common denominator for all our (Re)Actors a `GenericReactor`. `GenericReactor` is just an abstract class – and has a pure virtual function, `react()`:

```
class GenericReactor {
  virtual void react(const Event& ev) = 0;
  virtual ~GenericReactor() {}
}
```

Let's name the piece of code which calls `GenericReactor`'s `react()` *Infrastructure Code*. Quite often this call will be within a so-called 'event loop' (see Listing 1).

```
std::unique_ptr<GenericReactor> r
  = reactorFactory.createReactor(...);

while(true) { //event loop
  Event ev = get_event();
    //from select(), libuv, ...
  r->react(ev);
}
```

Listing 1

Let's note that the `get_event()` function can obtain events from wherever we want; from `select()` (which is quite common for servers) to libraries such as **libuv** (which is common for clients).

Also let's note that an event loop such as the one above is by far *not* the only way to call `react()`: I've seen implementations of Infrastructure Code ranging from the one running multiple (Re)Actors within the same thread, to another one which deserialized the (Re)Actor from a database (DB), then called `react()`, and then serialized the (Re)Actor back to the DB. What's important, though, is that even if `react()` can be called from different threads, it ***must*** be called *as if* it is one single thread. In other words, if necessary, all thread sync should be done ***outside*** of our (Re)Actor, so `react()` doesn't need to bother about thread sync regardless of the Infrastructure Code in use.

Finally, let's name any specific derivative from Generic Reactor (which actually implements our `react()` function) a `SpecificReactor`:

```
class SpecificReactor : public GenericReactor {
  void react(const Event& ev) override;
};
```

In addition, let's observe that whenever the (Re)Actor needs to communicate with another (Re)Actor then – adhering to the 'Do not communicate by sharing memory; instead, share memory by communicating' principle – it merely sends a message, and it is only this message which will be shared between (Re)Actors. In turn, this means that we can (and *should*) use single-threaded allocation for *all* (Re)Actor purposes – except for allocation of those messages intended for inter-(Re)Actor communications.

Task: Serialization for the same executable

Now, let's define what we're going to do with our (Re)Actor in this article (and *why*). Basically, as discussed in [Ignatchenko-Ivanchykhin16], when dealing with (Re)Actors, we often need to serialize the current state of our (Re)Actor so that we can deserialize it in *exactly the same executable* (though this executable can run on a completely different computer etc.). Applications of such 'serialization for exactly the same executable' are numerous; in particular, it is useful for (see, for example, [NoBugs17c]):

- Migrating our (Re)Actors (for example, to load-balance them)
- Low-Latency Fault Tolerance for (Re)Actors

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

One very important property of such serialization is that it is extremely difficult to beat this kind of serialization performance-wise

- Production post-mortem debugging (serializing the state, plus all inputs after that state, of the (Re)Actor, and then transferring them to developer's box in case of a crash)

'Speedy Gonzales' (Re)actor-level serialization

Now, as we have both our (Re)Actors and our task at hand more-or-less defined, let's see how well we can do with our per-(Re)Actor allocator.

Actually, it is fairly simple:

- We re-define global allocator (for example, `malloc()`/`free()`, though depending on compiler specifics and options YMMV) so that it acts as a bunch of per-(Re)Actor allocators (or at least per-thread allocators) – more on it below

  This means that within our (Re)Actor, we do ***not*** need to specify allocators for each call to 'new' and for each collection <phew! />
- Within our per-(Re)-Actor allocator, we allocate/deallocate OS pages ourselves (via calls such as `VirtualAllocEx()` or `mmap()`); of course, we also keep a list of all the OS pages we're using.
- Whenever we need to serialize our (Re)Actor, we simply dump all the pages used by the allocator of this (Re)Actor (with an address of each page serialized) – that's it!
- When we need to deserialize our (Re)Actor, we try to allocate OS pages at exactly the same (virtual) addresses as they were originally allocated

  If such allocation **succeeds** for all our serialized pages (which is common – though strictly speaking, not guaranteed – when we're deserializing a (Re)Actor into a dedicated-for-this-(Re)Actor process, which in turn is common for debugging), we just need to copy pages back from the serialized form into allocator (that's it)

  If allocation at the same address **isn't possible** for whatever reason, we have to use a process which is essentially similar to relocation discussed in [NoBugs17a]. Very briefly:
  - We have a *relocation map*, which gives the mapping between 'old' page addresses and 'new' page addresses.
  - At OS level, we make sure the pages with 'old' addresses are unreadable.
  - We run *traversing* (as discussed in [NoBugs17a]) over the state of our (Re)Actor. During this traversing process, we merely try to access all the elements of the state of our (Re)Actor. Whenever any pointer within our (Re)Actor state happens to point to the 'old' page, such access will fail (causing an *access violation* exception). We catch each such an exception, and update the pointer which caused the exception to the 'new' value within the corresponding 'new' page (calculating the 'new' value using the relocation map).

    Note that while the traversing of our *own* collections can easily be handled along the same lines as above, traversing and fixing *standard* collections can be outright impossible without adding a few lines to them ☹. How to handle this in practice? It depends, but one way to do it is to take a *cross-platform* STL implementation (such as EASTL), and to add the few lines implementing traversing for each collection you require (it is NOT rocket science for any *specific* STL).

Bingo! After such traversing of the *whole* state of our (Re)Actor is completed, we can be sure that *all* the pointers to the heap within our (Re)Actor are updated with the new values. In turn, it means that *all* the pointers to the state are already updated, so all the relocations due to serialization are already handled, and we can proceed with normal (Re)Actor processing.

One very important property of such serialization is that it is extremely difficult to beat this kind of serialization performance-wise. Deserialization is a slightly different story due to potential relocation, but it is still pretty fast. Also, for several of the use cases mentioned in the 'Task: Serialization for the same Executable' section above, it is only the performance of *serialization* which really matters. Indeed, all we need to do is `memcpy()` for large chunks of RAM, and with `memcpy()` speeds being of the order of 5-10 gigabytes/second at least for x64 (see, for example, [B14]), this means that even to serialize a (Re)Actor which has 100MB state, we're talking about times of the order of 10-20ms. Serializing the same thing using conventional serialization methods (even really fast ones, such as the one discussed in [Ignatchenko-Ivanchykhin16]) is going to be significantly slower. The exact numbers depend on the specifics of the organization of the data, but if we have a randomly filled 100MB `std::map<int>`, just iterating over it without any serializing is going to take the order of 100ms, almost an order of magnitude longer (!).

Parallel serialization aided by (kinda-)copy-on-write

For those apps where even 10-20ms of additional latency per 100MB of state is too much, it might be possible to reduce it further.

One of the implementations would work along the following lines (which are ideologically similar, though not identical to, classic Copy-on-Write):

When we want to serialize, we (in our 'main' (Re)Actor processing thread):

- create a list of pages to be serialized
- pre-allocate space in some other area of RAM where we want to serialize
- for all the pages to be serialized, set an 'already serialized' parameter to `false`
- mark all the pages to be serialized as 'no-write' (using `VirtualAllocEx()` or `mprotect()`).
- start another 'serialization' thread (use an always-existing dedicated serialization thread, take it from thread pool, etc.)
- continue processing the (Re)Actor messages in the main thread

The 'serialization' thread just takes pages from the list to be serialized one by one, and for each such page:

we do have to copy some of the pages within the main thread (causing some latency), for typical access patterns, this will happen relatively rarely

- checks if it is already serialized: if yes, skips; and if not, marks it as 'already serialized' (which should be done using an atomic CAS-like operation to prevent potential races with the main thread)
- if it wasn't already serialized:
  - serializes the page into pre-allocated space
  - removes the 'no-write' protection from the page

Whenever we have write access to one of the 'no-write' pages, we catch the appropriate CPU-level exception, and within the exception handler:

Check if the page being accessed is already being serialized (this can happen due to races with the 'serialization' thread); this should be done in an atomic manner similar to the 'serialization' thread as described above

If the page isn't serialized yet:

- serialize it into pre-allocated space
- remove the 'no-write' protection from the page, so future writes no longer cause any trouble.

That's it. While with such a processing we *do* have to copy *some* of the pages within the main thread (causing *some* latency), for typical access patterns, this will happen relatively rarely, significantly reducing overall serialization latency observed within the 'main' (Re)Actor thread. For example, if out of our 100MB (~=25K pages) (Re)Actor state, only 1000 pages are modified during our 20ms serialization – then the latency cost of the serialization will drop approximately by a factor of 25x, bringing observable latency to around 1ms (which is acceptable for the *vast majority* of the systems out there, even for first-person-shooter games).

Per-(re)actor allocators in a usable manner

Up to now, we are merely assuming that our allocators can be made per-(Re)Actor; one obvious way of doing this is to have our (Re)Actor code specify our own allocator for each and every allocation within our (Re)Actor (we'll need to cover both explicit calls to new, and all implicit allocations such as collections).

While such a naïve approach would work in theory, it is way too inconvenient to be used in practice. Fortunately, changing an allocator to a per-(Re)Actor one happens to be possible *without any changes to the (Re)Actor code*. In particular, it can be done along the following lines.

First, we replace **`malloc()`**/**`free()`** (***Important***: make sure that your global **`::operator new`**/**`::operator delete`**, and your default **`std::allocator`** also use the replaced functions (!). The latter might be rather tricky unless your **std** library already uses **`::operator new()`**/**`::operator delete()`**, but usually it can be take care of; in particular, for GCC, see [GCC] and the **`--enable-libstdcxx-allocator`** option for **`./configure`** of **libstdc++.**)

To implement our own **`malloc()`**, we're going along the lines of Listing 2. (Of course, **`free()`** should go along the same lines.)

The point here is that our Infrastructure Code (the one which calls our (Re)Actor) sets the **`current_allocator`** pointer before every call to **`GenericReactor::react()`** (see Listing 3).

```
thread_local OurAllocator* current_allocator
  = nullptr;

void* malloc(size_t sz) {
  if( current_allocator )
    return current_allocator->malloc(sz);
  else
    return non_reactor_malloc(sz);
}
```

Listing 2

Of course, this is a kind of trick – but it will work. Very briefly: first, we confine our **`current_allocator`** variable to the current thread by using **`thread_local`**, and then within this single thread, we can easily control which allocator is currently used by simple assignments within our Infrastructure Code. One thing to remember when using this way is to make sure that we set **`current_allocator`** before *each and every* method call of our (Re)Actor (including its constructor and destructor(!)).

That's it: we've made our (Re)Actor use a per-(Re)Actor allocator – and without changing a single line *within* our (Re)Actor's code too ☺.

Summary

To summarize this part III of the mini-series on 'Allocators for (Re)Actors':

- Allocator-based serialization for (Re)Actors is both
  - Easy to implement in a very generic manner, and
  - Extremely fast (for x64 – around tens of ms per 100MB of state)

  If necessary, parallel serialization may further reduce latencies (in some cases, down to – very roughly – a single digit ms of latency per 100MB of the state).

```
current_allocator = create_new_allocator();
std::unique_ptr<GenericReactor> r
  = reactorFactory.createReactor
  (current_allocator,...);
current_allocator = nullptr;

while(true) { //event loop
  Event ev = get_event();
    //from select(), libuv, ...
  current_allocator = r->allocator;
  r->react(ev);
  current_allocator = nullptr;
}

current_allocator = r->allocator;
r.reset();
current_allocator = nullptr;
```

Listing 3

- The Allocators can be replaced to make them per-(Re)Actor, without any changes to the (Re)Actor code(!)

And as this part III *concludes* this mini-series, let's summarize all our findings (from all three parts).

Part I

- (Re)Actor-based allocators allow for very efficient allocation, with three potential modes:
  - 'Fast' mode (no protection, but faster than regular `malloc()`)
  - 'kinda-Safe' mode (with protection from *some* of the memory corruptions)

    Here, we introduced a potentially novel method of implementing 'dangling' pointer detection in runtime – the one based on ID comparison. Compared to traditional 'tombstones' it has better locality, and will usually outperform it.
  - 'kinda-Safe with Relocation' mode, with added ability to relocate heap objects (this, in turn, allows to avoid dreaded external fragmentation, which tends to eat *lots* of RAM in long-running programs).
- Simple 'traversing' interface is sufficient to ensure that all the pointers in the (Re)Actor state are updated

Part II

By adding a few more of easily understood guidelines, we can extend our 'kinda-Safe' mode from Part I into 'really safe' C++ dialect.

All the guidelines/rules we need to follow are *local*, which enables reasonable tool-aided enforcement and allows to keep code maintainable.

Part III

- Custom (Re)Actor-based allocator can be used for the all-important for (Re)Actors serialization for the same executable. It is (a) very easy to maintain for (Re)Actor code, and (b) extremely fast.
- Per-(Re)Actor allocators can be implemented without *any* changes within (Re)Actor itself (i.e. all the necessary changes can be confined to Infrastructure Code).

Phew. It was rather long mini-series, but I hope I have made my points about the significant advantages of allocators specialized for (Re)Actor purposes reasonably clear. ■

References

[B14] Thomas B, 'Single-threaded memory performance for dual socket Xeon E5-* systems', https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/509237

[GCC] 'The GNU C++ Library Manual', Chapter 6, https://gcc.gnu.org/onlinedocs/libstdc++/manual/memory.html#allocator.ext

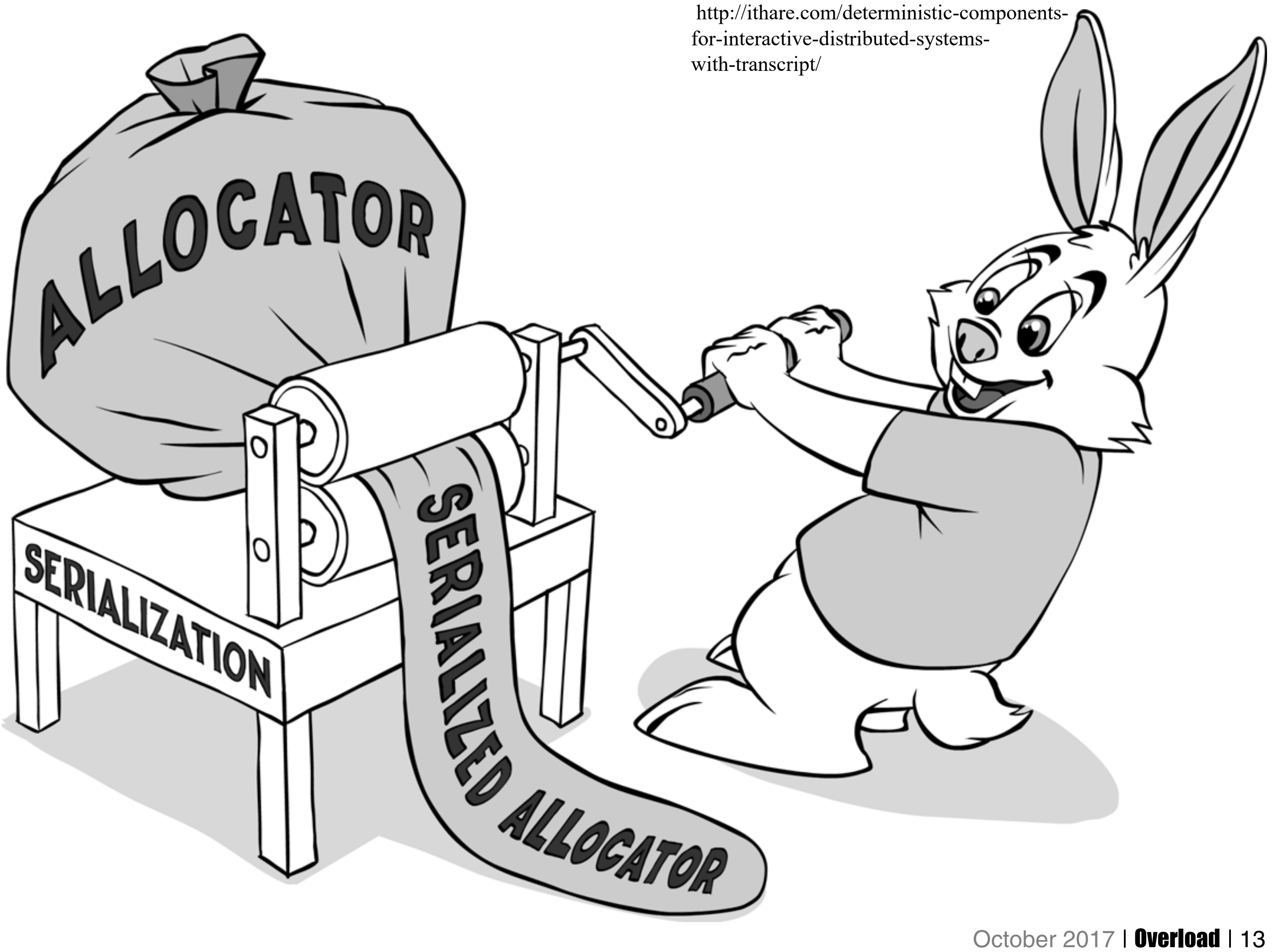
[Ignatchenko-Ivanchykhin16] Sergey Ignatchenko and Dmytro Ivanchykhin, 'Ultra-fast Serialization of C++ Objects', *Overload* #136, Dec 2016

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

[NoBugs17a] 'No Bugs' Hare, 'Allocator for (Re)Actors with Optional Kinda-Safety and Relocation', *Overload* #139, Jun 2017

[NoBugs17b] 'No Bugs' Hare, 'A Usable C++ Dialect that is Safe Against Memory Corruption', *Overload* #140, Aug 2017

[NoBugs17c] 'No Bugs' Hare, 'Deterministic Components for Interactive Distributed Systems', ACCU2017, available at http://ithare.com/deterministic-components-for-interactive-distributed-systems-with-transcript/



Polymorphism in C++ – A Type Compatibility View

Polymorphism can happen in many ways. Satprem Pamudurthy compiles an exhaustive matrix of approaches.

Polymorphism is the provision of a single interface to entities of different types [Stroustrup]. While there is a single interface, there can be many possible implementations of the interface and the appropriate implementation is selected (either explicitly by the programmer, or by the compiler) based on the types of the arguments. C++ supports multiple kinds of polymorphism and classifying them based on certain common characteristics makes it easier for us to reason about design choices. One way to classify them is in terms of when the implementation is selected i.e. compile-time vs. runtime. In this article, we will look at the relationship between type compatibility and polymorphism, and see that the different kinds of polymorphism can also be classified based on whether they require nominative or structural compatibility between the types of arguments (actual parameters) and parameters (formal parameters). This gives us a framework for arguing about design choices in two dimensions.

Type compatibility and polymorphism

An important characteristic of type systems concerns the notion of type compatibility i.e. whether the type of an expression is consistent with the type expected in the context that the expression appears in [Wikipedia-1]. To understand the importance of type compatibility to polymorphism, consider the following definition of polymorphism from *On Understanding Types, Data Abstraction and Polymorphism* by Luca Cardelli and Peter Wegner:

> Similarity relations among type expressions that permit a type expression to denote more than one type, or to be compatible with many types, are referred to as polymorphism [Cardelli85].

Basically, the notion of type compatibility is at the heart of polymorphism. The strongest similarity relation between types is equivalence (under some rules for equivalence). Types that are equivalent are compatible with each other. In languages with subtyping, such as C++, a subtype is compatible with its super types. The forms of type compatibility that are of interest to us are nominative and structural.

Nominative typing is the one that is immediately familiar to C++ programmers. In nominative typing, two variables are type-compatible if and only if their declarations name the same type [Wikipedia-2]. Basically, types are not equivalent unless they have the same name and are in the same scope. In C++, aliases defined using `typedef` are equivalent to the original type. In nominative subtyping, such as inheritance in C++, super type – subtype relationships are explicitly declared. The C++ type system is primarily nominative.

In a structural type system, on the other hand, type equivalence is determined by the type's actual structure or definition, and not by other characteristics such as its name or place of declaration [Wikipedia-3]. Super type – subtype relationships are also determined based on the structures of the types. One type is a subtype of another if and only if it contains all the properties of that type. Conversely, any type whose properties are a subset of those of another type is a structural super type of that type. In contrast to nominative subtyping, we can define a structural super type of an existing type without modifying the definition of that type. Thus, structural typing thus allows us to treat concrete types that are otherwise unrelated by inheritance in a polymorphic manner. Duck typing is a special form of structural typing where type compatibility is only based on the subset of a structure used within a particular context. Two types that are considered compatible in one context might not be compatible in another context. C++ exhibits structural typing with respect to type template parameters. In this article, I use structure to mean a type's public interface, and property to mean a type's public method or data.

```
class Dog {
public:
  const std::string& name() const;
  void moveTo(const Point& point);
};

class Cat {
public:
  const std::string& name() const;
  void moveTo(const Point& point);
};

class ThingWithName {
public:
  const std::string& name() const;
};
```

Listing 1

In Listing 1, while both the `Cat` and the `Dog` classes have the same structure, they are not considered equivalent in a nominative type system such as C++. For instance, you cannot pass an object of type `Dog` to a function that expects a `Cat`. However, they are structurally equivalent. Also, in a structural type system, `ThingWithName` would be a super type of both the `Cat` and the `Dog` classes. Now, let's see the forms of type compatibility required by the different kinds of polymorphism in C++.

Polymorphism at compile-time

C++ is a statically typed language, which means that variables have a type (whether declared or deduced) at compile-time and they retain that type throughout their life. Compile-time polymorphism, also known as static polymorphism, involves selecting an implementation based on static types of the arguments. C++ allows two kinds of compile-time polymorphism – ad hoc and parametric.

Satprem Pamudurthy works in the financial services industry and has been programming professionally for over 10 years. His main tools are C++ and Python but he will use anything that lets him get the job done. In the past, that has meant Java, C# and even VBA. You can reach him at satprem@gmail.com.

Runtime polymorphism, also known as dynamic polymorphism, involves selecting an implementation based on the runtime type of one or more arguments (dynamic dispatch)

```
void printName(const Dog& v)
{
  std::cout << v.name() << std::endl;
}

void printName(const Cat& v)
{
  std::cout << v.name() << std::endl;
}

int main()
{
  Dog d;
  printName(d);

  Cat c;
  printName(c);
  return 0;
}
```

Listing 2

Ad hoc polymorphism [Strachey67] in C++ is implemented using overloaded functions. Function overloading allows us to define two or more functions with the same name in the same scope [Wikipedia-4]. Overloaded functions are distinct and potentially heterogeneous implementations over a range of specific types. The compiler selects the most appropriate overload based on the number and types of the arguments [cpp-reference]. Overload resolution requires either nominative compatibility or implicit convertibility between the types of the arguments and the parameters. Consider the overloaded functions in Listing 2.

In ad hoc polymorphism, we need to provide a separate implementation for each type. This, however, leads to code duplication if we want to uniformly apply an algorithm to values of different types. Parametric polymorphism, on the other hand, allows us to write generic functions and generic data types that can handle values identically without depending on their type [Strachey67]. In C++, parametric polymorphism is implemented using templates – generic functions are defined using function templates, and generic data types are defined using class templates. For each type parameter, a template implicitly specifies the minimum set of properties that the corresponding type argument must have. We can use SFINAE [Vandevoorde02] and Concepts (which have recently been merged into the C++20 draft) to explicitly add additional requirements on the type arguments. Consider the example in Listing 3.

```
template<typename T>
void printName(const T& v)
{
  std::cout << v.name() << std::endl;
}

int main()
{
  Dog d;
  printName(d);

  Cat c;
  printName(c);
  return 0;
}
```

Listing 3

The **`printName`** template has one type parameter and can be instantiated with **`Dog`**, **`Cat`**, or any type that has a **`name()`** method, irrespective of any other methods they might have. In other words, a template can be instantiated with any types that are structurally equivalent to, or are structural subtypes of its type parameters. Note that this is also true of generic lambdas. After all, the call operator of a generic lambda's compiler-generated type is a template.

CURIOUSLY RECURRING TEMPLATE PATTERN [Coplien95] is a technique that uses templates and inheritance to simulate subtype polymorphism at compile-time. In CRTP, the base class template is implemented in terms of the derived class's structural properties. Consider the example in Listing 4.

Compile-time polymorphism in C++ is quite powerful and expressive, but because implementations are selected at compile-time, it cannot depend on information that is available only at runtime. In fact, often we only know the type of objects to create at runtime. Let us now see how polymorphism works at runtime.

Polymorphism at runtime

Runtime polymorphism, also known as dynamic polymorphism, involves selecting an implementation based on the runtime type of one or more arguments (dynamic dispatch). In C++, it is implemented using subtyping and the most common form of subtyping for dynamic dispatch is inheritance with virtual functions. Overrides of a virtual function essentially overload it on the type of the implicit **`this`** argument. Virtual function calls are resolved based on the runtime type of **`this`** (which must be nominatively compatible with the base class) and the static types of the rest of the arguments. Consider the example in Listing 5.

Inheritance is a form of nominative subtyping, and the **`printName()`** method can be called with any object whose type is a nominative subtype of **`Animal`**. However, because inheritance requires explicit declaration of super type – subtype relationships, is not always a viable solution for runtime polymorphism. Some of the types we need to abstract over might be in a third-party library that we cannot modify, and thus cannot be made to derive from a common base class. The types might not even be related – forcing a set of unrelated types to derive from a common base class is intrusive and does not necessarily express an is-a relationship. Inheritance while there is **no language support** in C++ for **runtime polymorphism** based on structural compatibility, it **can be simulated** using type erasure

```
template<typename T>
class Animal {

protected:
  Animal() { }

public:
  const std::string& getName() const
  {
    return static_cast<const T *>(this)->name();
  }
};

class Dog : public Animal<Dog> {
public:
  const std::string& name() const;
  void moveTo(const Point& point);
};

class Cat : public Animal<Cat> {
public:
  const std::string& name() const;
  void moveTo(const Point& point);
};

template<typename T>
void printName(const Animal<T>& v)
{
  std::cout << v.getName() << std::endl;
}

int main()
{
  Dog d;
  printName(d);

  Cat c;
  printName(c);
}
```

Listing 4

```
class Animal {
public:
  virtual ~Animal() {}
  virtual const std::string& name() const = 0;
};

class Dog : public Animal {
public:
  const std::string& name() const override;
  void moveTo(const std::string& std::string);
};

class Cat : public Animal {
public:
  const std::string& name() const override;
  void moveTo(const std::string& std::string);
};

void printName(const Animal& v)
{
  std::cout << v.name() << std::endl;
}

int main()
{
  Dog d;
  printName(d);

  Cat c;
  printName(c);
}
```

Listing 5

also implies a tight coupling between the base and the derived classes, which might cause scalability issues. For more details about the various types of inheritance and their implications, please refer to John Lakos's presentation on inheritance [Lakos16a]. The video of his presentation is available on the ACCU YouTube channel [Lakos16b].

The ability to select implementations based on structural compatibility of runtime types would help overcome some of the drawbacks of using inheritance, but how do we do that? The answer is type erasure. Type erasure is used to create non-intrusive adapters that are implemented in terms of the structural properties of the adapted object's type, and some of those adapters behave just like structural super types. Consider the example in Listing 6.

The container (`ThingWithName`) can be instantiated with an object of any type as long as it has the `name()` method, irrespective of any other methods it may have i.e. any type that is structurally equivalent to or is a structural subtype of `ThingWithName`. Because it is not a class template, clients of `ThingWithName` do not have to know the underlying type at compile-time. Thus, while there is no language support in C++ for runtime polymorphism based on structural compatibility, it can be simulated using type erasure. Runtime structural subtype polymorphism is widely used in C++, even though we might not have thought of it as such. For example, `std::any` can be seen as the structural counterpart of an empty base class, and the `std::function` template can be seen as generating structural super types of callable types (any type with an explicit or an implicit `operator()`).

```
class ThingWithName {
public:
  template<typename T>
  ThingWithName(const T& obj)
  : inner_(std::make_unique<Holder<T> >(obj))
  {

  }
  const std::string& name() const
  {
    return inner_->name();
  }

private:
  struct HolderBase {
    virtual ~HolderBase() { }
    virtual const std::string& name() const = 0;
  };

  template<typename T>
  struct Holder : public HolderBase {
    Holder(const T& obj)
    : obj_(obj)
    {
    }
    const std::string& name() const override
    {
      return obj_.name();
    }
    T obj_;
  };
  std::unique_ptr<HolderBase> inner_;
};

void printName(const ThingWithName& v)
{
  std::cout << v.name() << std::endl;
}

int main()
{
  ThingWithName d((Dog()));
  printName(d);

  ThingWithName c((Cat()));
  printName(c);
}
```

Listing 6

Nominative vs. structural typing – the trade-offs

Structural typing enables unanticipated re-use i.e. it frees us from having to anticipate all possible types that we want to apply an algorithm to, and from having to anticipate all possible algorithms that we want to apply to a type. While it is more flexible than nominative typing, there are certain drawbacks. Just because two types are structurally equivalent does not mean they are semantically equivalent. The advantage of a nominative system is that type names convey contracts and invariants that are not necessarily apparent from the structure of type alone. It allows the programmer to explicitly express her design intent, both with respect to contracts and how the various parts of the program are intended to work together. By allowing us to also use the ‘meaning’ of the type to select implementations, nominative typing allows for stronger type-safety than structural typing.

A matrix of polymorphism choices

We have seen that we can classify compile-time and runtime polymorphism in terms the form of type compatibility they require. It is helpful to represent these in a two-dimensional matrix. Bear in mind that these are just building blocks and that real-world design patterns do not neatly fall into one of these categories. For example, in the Curiously Recurring Template Pattern the base class template requires structural compatibility whereas the derived classes are nominative subtypes of the base class.

	Compile-time	Runtime
Nominative Typing	Overloaded functions	Inheritance with virtual functions
Structural Typing	Templates and generic lambdas	Type erasure

The matrix shows us that the dichotomy that exists between nominative and structural type compatibility at compile-time also exists at runtime. The choice of the kind of polymorphism to use in C++ is often phrased as a choice between templates and inheritance. However, as we can see from the matrix, the journey from templates to inheritance requires two hops. If we need runtime polymorphism but want to retain the flexibility of structural typing, type erasure is a more natural choice. We should, however, choose inheritance if we also want the stronger type-safety of nominative typing. The matrix provides a framework for understanding the implications and trade-offs of our design choices as they relate to polymorphism.

I should point out that there are exceptions to the type-compatibility view of polymorphism. Type casting, whether implicit or explicit, can make a monomorphic interface appear to be polymorphic. This is sometimes referred to as coercion polymorphism. Also, C++ allows non-type template parameters, meaning templates can be instantiated with values in addition to types. ■

References

[Cardelli85] L. Cardelli and P. Wegner, *On Understanding Types, Data Abstraction and Polymorphism*, 1985

[Coplien95] J. Coplien, Curiously Recurring Template Patterns, *C++ Report*: 24–27, February 1995

[cpp-reference] Overload Resolution http://en.cppreference.com/w/cpp/language/overload_resolution

[Lakos16a] Proper Inheritance, John Lakos (https://raw.githubusercontent.com/boostcon/cppnow_presentations_2016/master/00_tuesday/proper_inheritance.pdf)

[Lakos16b] Proper Inheritance, John Lakos, ACCU 2016 (https://www.youtube.com/watch?v=w1yPw0Wd6jA)

[Strachey67] C. Strachey, *Fundamental concepts in programming languages*, 1967

[Stroustrup] Bjarne Stroustrup’s C++ Glossary http://www.stroustrup.com/glossary.html#Gpolymorphism

[Vandevoorde02] D. Vandevoorde, N. Josuttis (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional

[Wikipedia-1] Type System (https://en.wikipedia.org/wiki/Type_system)

[Wikipedia-2] Nominal Type System (https://en.wikipedia.org/wiki/Nominal_type_system)

[Wikipedia-3] Structural Type System (https://en.wikipedia.org/wiki/Structural_type_system)

[Wikipedia-4] Overloading (https://en.wikipedia.org/wiki/Function_overloading)

Open Source – And Still Deceiving Programmers

Malware can hijack the normal flow of your program. Deák Ferenc walks through the ELF format to avoid malicious code injection.

Computer viruses, trojan horses, rootkits and other pieces of malicious software have been around for a very long time. Since the first application that could be classified as a 'classic' computer virus (based on the theory presented in [Neumann66]) appeared in 1971, countless variations of the same construct have appeared, with more or less destructive intentions, varying from harmless jokes to highly specialized pieces of malicious software targeting industrial processes and machines, while the number of them as per [BBC] has passed 1000000 (by 2008).

The threat presented by these noxious pieces of code is so significant that a new word has emerged in order to classify them: 'malware'. This is short for malicious software. Malware is a collective category encompassing several types of harmful applications from the classic virus (ie. an application which replicates itself by modifying already existing files or data structures on a computer) through worms (applications which move through the network infecting computers) to trojan horses (applications posing as something other than they are, very often disguised as legitimate applications). Malware also covers spyware and keyloggers (these spy on your activities, frequently registering your keystrokes which then are sent to malicious parties), rootkits (very low-level applications, more often found at the hardware/OS level, hidden from user-level access) and various ransomware applications, which hold your computer hostage by encrypting your data until you pay a 'ransom'.

Recently, the trend in the propagation of these damaging applications has changed. Due to the increase in the level of security features in Operating Systems which were more traditionally affected by viruses, the number of classical 'viruses' which have multiplied themselves via modifying existing system files and spread via execution is in recess; however, there is a sharp increase in the sighting of other types of malware which are propagating via email attachments, malicious downloads or by simply utilizing vulnerabilities in operating systems [Wikipedia_2].

Most of these vicious applications have, however, one thing in common: we rarely see the original source code which led to their creation (except for some high-profile leaks such as [TheIntercept]). Indeed, it would be kind-of silly to ask fellow programmers to "please compile and run this code, it is a virus, it will infect your computer and it will multiply itself in countless pieces before rendering your machine unusable" … this would be similar to the old joke of receiving an email with the content "Hi, this is a manual virus. Since I am not so technically proficient as to be able to write a real virus, please forward this email to everyone in your contact list and delete all your files. With kind regards, Amateur Virus Writer".

This certainly does not mean that everything you download from the internet and compile yourself is guaranteed to be clean and harmless. The open source community focused around various free products goes to great lengths in order to provide a high quality application without backdoors (ie: unofficial ways which makes access to certain systems possible) and, considering the backdoor attempt of 2003 targeting Linux [LinuxBackdoor], their effort invested in this direction is more than welcome.

Considering this introduction, you might envision that this article will be a tutorial on how to write viruses, Trojan horses and other maleficent pieces of code in order to achieve world domination, or just simply for fun. You couldn't be further from the truth. On the contrary. This article will present practical ways for defending your open source application against hideous interventions by programmers with hidden intentions, who would like to hijack the normal flow of your code by various means we will discuss later.

We will see different ways of executing code, we will dig deep in the binary section of executable files, and we will have the chance to examine the source code of a true shape-shifting application. All of these can be present in real life situations when you are merging code from your contributors into the final product, and if you don't pay attention some not so well intended modifications will end up in the final product.

Running an application

The simple and mundane task of starting an executable application compiled for your platform in fact involves a long list of processes from the operating system's side. Since the details of this topic in itself are worth a small book, I will just provide a very high-level overview of what happens when you start an application.

Starting a process in Linux

Applications in Linux use the so called 'ELF' format (Executable and Linkable Format [Wikipedia_1], [O'Neil16]). This is a format adopted by various Unix-like operating systems, but more recently the Linux subsystem of Windows 10 also shows support for this type of executable.

A short overview of the steps taken when a new application is launched (either from a shell or from somewhere else) is as follows:

1. The **`fork`** system call is used to create a new process. The fork will create a 'copy' of the current process and set up a set of flags reflecting the state of the new process [fork].
2. The **`execve`** system call is executed with the application to be executed [execve].
3. Down in the Linux kernel, a **`linux_binprm`** [linux_binprm] structure is being built in order to accommodate the new process, which is passed to:
   ```
   static int load_elf_binary(struct linux_binprm
   *bprm)
   ```
   in **`fs/binfmt_elf.c`** [load_elf_binary]
4. **`load_elf_binary`** does the actual loading of the ELF executable according to the specifications and at the end it calls **`start_thread`**, which is the platform dependent way of starting the loaded executable.

Deák Ferenc Ferenc has wanted to be a better programmer for the last 15 years. Right now he tries to accomplish this goal by working at FARA (Trondheim, Norway) as a system programmer, and in his free time, by exploring the hidden corners of the C++ language in search for new quests. fritzone@gmail.com

The open source community focused around various free products goes to great lengths in order to provide a high quality application without backdoors

For those expressing a deeper interest in this field, the excellent article [HowKernelRuns] or Robert Love's outstanding book *Linux Kernel Development* [Love10] will provide all the details required.

The ELF format

The ELF binary in itself is a complex subject – a full description can be found in *Learning Linux Binary Analysis* [O'Neil16], and a shorter one on Wikipedia [Wikipedia_1] – so let's summarize it briefly:

The ELF headers

The file header of the ELF file starts with a few magic numbers for correctly identifying this as being a valid ELF file: 0x7F followed by the characters `E`, `L` and `F` (0x45, 0x4c, 0x46). The architecture of the file is specified (whether 32 or 64 bit) and whether the encoding of the file is big endian or little endian.

In the header, a special field denotes the target operating system's Application Binary Interface and the instruction set of the binary. ELF files can be of different types, such as relocatable, executable, shared or core. This information is also stored in the elf header.

There are several fields in the header dealing with the length and format of the ELF sections, described below.

The file header of the ELF file is followed by a program header, which describes to the system how to create a process image, and several section headers.

ELF sections

There are several sections in an ELF file, each containing various data, vital to correctly understand and run the application. Among these sections is:

- the `.text` section, which contains the actual code of the application,
- the `.rodata` section, containing the constant strings from the application
- the `.data` section, which contains for example the initialized global variables
- the `.bss` section, which contains uninitialized global data
- various other sections describing how this application handles shared libraries and also…
- sections describing application startup and destruction steps in the `.ctors` and `.dtors` sections (correspondingly `.init_array` and `.fini_array`). These sections contain function pointers to the methods, which will be called on application startup and shutdown, and we will have a more detailed look at them in later paragraphs of this article.

There is a handy Linux utility called `readelf`,which displays information about a specific ELF file's structure. We will refer to it in this article and will present the output of it frequently.

Deceiving techniques

So, after this short but necessary, background introduction, we have finally arrived at the focus point of the article. We will present here various techniques you have to carefully observe in order to keep your source healthy and free of unwanted side effects.

Mainless application

The `main` function in a 'normal' application is the place where the application starts [main_function]. However, be aware that C and C++ compilers handle `main` very differently. For example, Listing 1 will compile flawlessly using `gcc` with default compilation flags even though `void main(int a)` is strictly not standard compliant, but `g++`, being more picky, will refuse to compile it. For `gcc`, you need to use `-pedantic` to warn you about the return type of `main` not being `int` as required by the standard.

But what if an attacker does not wish to provide a main function (since he wants to pose the source code as being a part of a library)?

With `gcc`, there is always the possibility of using the `-e <entrypoint>` switch to specify a different entry point for your application. This is very observable in the build files, and there will be more to be dealt with, such as:

1. The need to specify `-nostartfiles` to the `gcc` command line in order to avoid the linking error: `(.text+0x20): undefined reference to 'main'`. As a short explanation to this, in the background `gcc` always links to some architecture specific files (such as `crtstuff.c`), which provide the application with the required startup functionality, which will end up calling the main [linuxProgramStartup]
2. Explicitly use the function `exit(<CODE>);` to properly exit the application in order to avoid the segmentation fault at exit.
3. There is no access to the common argc and argv values passed in to your application.

With these in mind, the source file in Listing 2, compiled with the command below should work as expected, by totally avoiding the `main` function thus deceiving you into believing the validity of the application.

Compiled with:

```
gcc mainless.c -o mainless -e my_main
  -nostartfiles
```

```
#include <stdlib.h>
#include <stdio.h>
void main(int a)
{
  void (*fp[])(int) = {main,exit};
  printf("%d\n",a++);
  fp[a/101](a);
}
```

Listing 1

the **initialization phase** of the application is a **preferred place** among wanna-be virus writers to **place ptrace related code**

```
#include <stdio.h>
#include <stdlib.h>

int my_main()
{
  printf("Mainless\n");
  exit(2);
}
```

Listing 2

It is interesting to note that comparing the binary file produced from compiling a 'mainless' program with a more standard binary – with 'main' and the linked-in `gcc` startup files – gives us the (not so) surprising result that the 'mainless' file is smaller, with a difference of up to 2000 bytes. Also, the elf structure, analyzed with `readelf`, gives a much simpler layout. A few differences in the header of the ELF file can also be observed:

Header	mainless	with main
Entry point address	0x400390	0x400430
Start of section headers	5184	6624
Number of section headers	20	31

This all proves that the 'mainless' file is, indeed, much smaller that the corresponding one with main.

Running code before main

When your targeted compiler is a C compiler, it can be really difficult to run code before `main` (or as we have seen, its replacement) starts. I have to emphasize that C++ has a dynamic initialization phase where arbitrary code is executed in order to initialize non-local variables. However, that comes with the well-known static initialization order fiasco: In C++, it is unpredictable which non-local is initialized before which other, so if one depends on the other one, the application in question may work flawlessly in some situations, while other reincarnations may suffer from this dependency with an uninitialized variable.

```
#include <stdio.h>

void my_main(int argc, char* argv[],
  char* envp[])
{
  printf("my main: %d parameters\n", argc);
}
int main(int argc, char* argv[])
{
  printf("main: %d parameters\n", argc);
}
__attribute__((section(".init_array"))) void
  (* p_my_main)(int,char*[],char*[]) = &my_main;
```

Listing 3

Fortunately, for C compilers, the ELF format provides extra support for running code before application start in the so called 'constructor' section, and it is also possible to hijack the `.init` section in order to execute code we want using special assembly syntax.

The .init_array section of the ELF binary

The `.init_array` (and the `.preinit_array`) section of the ELF binary contains a list of pointers (addresses of functions) called by the code initializing the application. This code (the one calling the functions in the `.init_array` section) usually resides in the `.init` section. The difference between `.preinit_array` and `.init_array` is that code in the `.preinit_array` is called before the `.init_array`.

The `gcc` compiler has a non-standard C extension to provide support for defining various dedicated elf sections with user specified code via the usage of the `__attribute__` syntax. An example of is in Listing 3.

Analyzing a debug session of this application will reveal interesting insights on the working of libc and the application startup procedure (see Figure 1).

For further information, evaluating this information, while combining it with the relevant section of assembly code (as the result of `objdump -h -S init_array`) will give details of how the code is actually executed,

```
(linux)$ gdb ./init_array
(gdb) break my_main
Breakpoint 2 at 0x40052a
(gdb) run
Starting program: .../init_array

Breakpoint 2, 0x000000000040052a in my_main(int, char**, char**) ()
(gdb) bt
#0  0x000000000040052a in my_main(int, char**, char**) ()
#1  0x00000000004005cd in __libc_csu_init ()
#2  0x00007ffff7a2d7bf in __libc_start_main (main=0x400550 <main>, ... ) at ../csu/libc-start.c:247
#3  0x0000000000400459 in _start ()
```

Figure 1

It is important that you carefully observe not only your C and C++ files, but also the accompanying build instructions

```
#include <stdio.h>
#include <stdlib.h>

int my_main()
{
  __asm__ (".section .init \n call my_main \n
.section .text\n");
  printf("my_main\n");
}

int main()
{
  printf("main\n");
}
```

Listing 4

and it is easily traceable based on how the `.init_array` section is handled on application startup.

If we don't require access to the command line parameters, `gcc` also has support as an extension for specifying a function to be called before main using a much less cryptic syntax: `__attribute__ ((constructor))`, which has similar consequence to the section initializer syntax:

```
void __attribute__ ((constructor)) premain()
{
  printf("premain called\n");
}
```

An even more obscure way of defining a method to be called before main is to rely on the '.init' section of the ELF format and do some assembly level magic (see Listing 4).

This will directly instruct the compiler to assemble the code `call my_main` into the section '.init' by using the assembly directive `.section .init`.

And, last but not least, some proprietary compilers targeting commercial environment (but not `gcc`) have support for a special compiler specific `#pragma` directive: `#pragma init`, which has the same effect as having `__attribute__((constructor))` for the `gcc`.

Process tracing

Just a side note: the initialization phase of the application is a preferred place among wanna-be virus writers to place ptrace related code. ptrace is a mechanism offered by Linux making it possible for a parent process to observe and influence the execution of other processes. It is mainly used in debugging and examining the state of other processes; however, certain anti debugging features – if compiled in into an executable – will make the analysis of compiled application difficult.

You should be looking for the `PTRACE_TRACEME`, which detects if the current application is traced by a debugger, and will act accordingly.

Running code after main

Very similar to the above scenario where we want to run code before the main, the ELF binary comes again to our help by making it possible to run C code after `main` has finished its lifetime. The 'destructors' section of the ELF is named the `.fini_array` and we can get access to it via the following code construct:

```
void end_app(void)
{
  printf("after main\n");
}
__attribute__((section(".fini_array"))) void
(* p_end_app)(void) = &end_app;
```

or there is also support for the `__attribute__ ((destructor))` syntax, if we find the above one very cryptic. This section comes very handy in case we are in the situation of running some 'last minute' cleanup jobs.

Similarly to the `.init` section, we can instruct the assembler to generate code into a `.fini` section, and also some proprietary compilers support the `#pragma fini` directive, to mark some identifiers as a finalization function.

A shape shifting application

It is important that you carefully observe not only your C and C++ files, but also the accompanying build instructions. Today, there are several tools which facilitate the management of build scripts (such as CMake, SCons, etc...) and these tools often use very complex files, which makes it easy to hide a few unwanted pieces of code, so be sure to check those too – most of the time that is the location a deception begins.

So, let's consider the following situation, where you are working on a free and open source budget management application, and one of your contributors submits the code in Listing 5.

Surely, this is a short unit test for some functions (supposedly `OPEN_INTEREST`, `READ_INTEREST`, `INTEREST_VALUE` being the function we 'want' to test), albeit a pretty poorly written one. However, it

```
include <stdio.h>
#include <stdlib.h>
// TODO: This is still work in progress, more
// months in the test plan are required and some
// code cleanup is necessary to remove the clutter.
// Will FIX ASAP!!!
#define INTEREST void*

#ifndef DEBUG_INTEREST
  #define L (int*)
  #define TEST void
  #define OPEN_INTEREST(a) \
    printf("Opening: %s\n", #a);
  #define READ_INTEREST(a) \
    printf("Reading: %s\n", #a);
  #define CLOSE_INTEREST(a) \
    printf("Closing: %s\n", #a);
  #define INTEREST_VALUE(a,b) b
#endif
```

Listing 5

```
TEST interest_calculator_test(char const *period)
{
  static INTEREST array[] = {
    L(26), L(2), 0,  // January
    L(5), L(26), 0,  // February
    L(8), L(2), 0,   // March
    L(11), L(2), 0,  // April
    L(14), L(14),0,  // May
    L(14), L(17), 0, // June
    L(20), L(20),0,  // July
    L(20), L(23), 0, // August
    L(2), L(2)};     // September

  INTEREST earning = array, *entry = array;
  char interest[1024] = {0}, *q, type = 3;
     // type 3 = Recurring
  char* c = interest; const char* name
    = "Monthly";

  OPEN_INTEREST(earning);
  if(earning == 0) READ_INTEREST(entry);
  if(entry != 0)  // Do we have an interest at
                  // the current point?
  {
    if(INTEREST_VALUE(entry,type) == 4)
      if((*INTEREST_VALUE(entry,name) != 46
                  // 11822 minutes ~ 8 days
        && *(int16_t*)
          (INTEREST_VALUE(entry,name)) != 11822))
        c = interest; q = (char*)period;
  }
  if(*q) *c++ = *q++;   // move to next period
  if(*q) {
    *c++ = 47;          // 47 - a check value,
                        // comes from test data
    q = (char*)INTEREST_VALUE(entry,name);
                        // get its value, save it
    while(*q) *c++ = *q++; // skip period
  }
  if(*q) {
    *(int16_t*)c = 0x000a;
    printf("Current value: %s", interest);
    *c = 0;
    interest_calculator_test(interest);
                   // advance month to next one
  }
  if(*q) CLOSE_INTEREST(entry);   // Done
}
int main()
{
  const char interest_period[] = {47, 0};
  interest_calculator_test(interest_period);
  return 0;
}
```

Listing 5 (cont'd)

seems to be harmless for the moment, and the comment on top clearly says it needs improvement, you decide to keep it in your source code base, hoping that the developer who submitted the patch just had a bad day, and he will come back with clarification lately. The code compiles, so it represents no harm and it does not really disturb anything in the normal flow of the application.

Soon a new patch comes in from the developer (maybe a different one, just to cause some more confusion), intended to have been a fix for something in the build system of the application, concerning the unit test, it looks just like the lines of code in Listing 6.

Yes, seemingly it patches something in compiling of the unit tests, but it's highly complex, difficult to read (intentionally), and regardless this is not a significant part of the application since the unit tests are just run on your computer.

The first sign of suspicion should have come from the forced include of `/usr/include/dirent.h` directly from the compiler's command line... So, this basically makes it possible for the malicious code writer to include a file into the compilation process from the command line, without appearing in the source file, thus avoiding suspicion. If we look further the malicious make entry contains some entries, which disturbingly resemble some commands used to handle directory structure in linux: **`opendir`**, **`readdir`** and **`closedir`**… (And I intentionally left it in this half-baked stage to raise awareness of this kind of issue, and the word 'label' was left intentionally in the defines too...)

Other signs of malevolence are the forced redefinitions of the **`if`** and **`while`** keywords. Unfortunately, there is nothing in the compiler to stop you from doing this, so all this will compile and is considered valid. Although it will only affect this file, there are numerous **`if`**s in it so let's dig a bit more. Soon you realize there is something fishy going on, so you decide to look at the preprocessed source code of this innocent looking unit test. It is in Listing 7.

To your horror, the source code of the application has changed into something incomprehensible, full of **`goto`**s and linux system calls accessing directories, and it seems to be doing something totally different now: it browses the directory structure of your computer (I took the liberty of beautifying some of the preprocessed output, and stripped out **`main`**, which is the same) and it prints the directories found to the console.

The evil programmer has used some of the not so well known **`gcc`** extensions, such as storing the addresses of labels, and with a carefully constructed array of labels and indexes, he has been able to abuse the usage of the **`__COUNTER__`** macro (the one which gives an increasing sequence of numbers) in order to calculate various jump locations together with generating labels in a coherent way to achieve his real intentions: traversing your filesystem and performing operations on it (for this specific scenario, just printing names).

A few very strange lines of code appear, such as **`*(int16_t*)( ((struct dirent*)entry)->d_name) != 11822`**. However, after some thought, this is nothing but a comparison of the **`d_name`** field of the **`struct dirent`** structure **`entry`** to "..". Because 11822 = 0x2E2E = "..".

In order to facilitate a continuous sequence provided by the **`__COUNTER__`** in the array indexes, and also the label counters, the

```
all:
  ${CC} -UL -UOPEN_INTEREST -UREAD_INTEREST -UCLOSE_INTEREST \
    -UINTEREST_VALUE -DL\(n\)=\&\&n_label\#\#n -DD=__COUNTER__ \
    -DT\(x\,y\)=x\#\#y -include /usr/include/dirent.h -DT2\(x\,y\)=T\(x\,y\) \
    -DDO=T2\(n_label\,D\)\: \
    -DOPEN_INTEREST\(entry\)=entry\=T2\(open\,dir\)\(period\)\; \
    -Dif\(x\)=T2\(go,to\)\ \*array\[x\?D\:D\]\;\ DO -include /usr/include/sys/types.h -Dwhile=if \
    -DREAD_INTEREST\(entry\)=entry\=T2\(read\,dir\)\(\(DIR\*\)earning\)\; \
    -DCLOSE_INTEREST\(entry\)=T2\(close\,dir\)\(\(DIR\*\)earning\)\; -include /usr/include/unistd.h \
    -DINTEREST_VALUE\(INTEREST_VALUE\,t\)=\(\(struct\ T2\(dir\,ent\)\*\)INTEREST_VALUE\)\-\>d_\#\#t \
    -DTEST=void -DDEBUG_INTEREST ${SOURCE}
```

Listing 6

```
void interest_calculator_test(char const *period)
{
  static void* array[] = {
    &&n_label26, &&n_label2, 0,
    &&n_label5,  &&n_label26, 0,
    &&n_label8,  &&n_label2, 0,
    &&n_label11, &&n_label2, 0,
    &&n_label14, &&n_label14,0,
    &&n_label14, &&n_label17, 0,
    &&n_label20, &&n_label20,0,
    &&n_label20, &&n_label23, 0,
    &&n_label2,  &&n_label2};

  void* earning = array, *entry = array;
  char interest[1024] = {0}, *q, type = 3;
  char* c = interest;
  const char* name = "Monthly";

  earning=opendir(period);

  goto *array[earning == 0?0:1];
n_label2:

  entry=readdir((DIR*)earning);;

  goto *array[entry != 0?3:4];
n_label5:

  goto *array[((struct dirent*)entry)->d_type
    == 4?6:7];
n_label8:

  goto *array[(*((struct dirent*)entry)->d_name
  != 46 && *(int16_t*)( ((struct dirent*)entry)->
  d_name) != 11822)?9:10];
n_label11:

  c = interest; q = (char*)period;

  goto *array[*q?12:13];
n_label14:

  *c++ = *q++;

  goto *array[*q?15:16];
n_label17:

  *c++ = 47;
  q = (char*)((struct dirent*)entry)->d_name;

  goto *array[*q?18:19];
n_label20: *c++ = *q++;

  goto *array[*q?21:22];
n_label23:

  *(int16_t*)c = 0x000a;
  printf("Current value: %s", interest);
  *c = 0;
  interest_calculator_test(interest);

  goto *array[*q?24:25];
n_label26:

    closedir((DIR*)earning);;
}
```

Listing 7

array contains a set of unused elements, such as zeroes; however, those values can be anything.

At this point, I have stopped, since it is not the intention of this article to publish destructive code but to raise awareness of its existence and provide meaningful ways for detecting and combating them.

Just a side note, for those wanting to do experiments on the code please find below the evil defines to make your experiments easier:

```
#define INTEREST void*
#define L(n) &&n_label##n
#define D __COUNTER__
#define T(x,y) x##y
#define T2(x,y) T(x,y)
#define DO T2(n_label,D):
#define OPEN_INTEREST(entry) \
  entry=T2(open,dir)(period);
#define if(x) T2(go,to) *array[x?D:D]; DO
#define while if
#define READ_INTEREST(entry) \
  entry=T2(read,dir)((DIR*)earning);
#define CLOSE_INTEREST(entry) \
  T2(close,dir)((DIR*)earning);
#define INTEREST_VALUE(INTEREST_VALUE,t) \
  ((struct T2(dir,ent)*)INTEREST_VALUE)->d_##t
```

Conclusion

As we have seen, the threats are real, and this article can by no means offer a full overview of all the software menaces that are present in our everyday life. Since we focused on an open source approach to deceiving techniques, we have tried to make the article as informative as possible without actually turning it into a 'how to write your own virus' essay. Please note that besides of presenting a few non-destructive scenarios, there could be several more that have not yet been identified... or that have been omitted intentionally. ■

References

[BBC] http://news.bbc.co.uk/2/hi/technology/7340315.stm

[execve] http://man7.org/linux/man-pages/man2/execve.2.html

[fork] http://man7.org/linux/man-pages/man3/fork.3p.html

[HowKernelRuns] https://0xax.gitbooks.io/linux-insides/content/SysCall/syscall-4.html

[linux_binprm] http://elixir.free-electrons.com/linux/v4.6.7/source/include/linux/binfmts.h#L14

[LinuxBackdoor] https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/

[linuxProgramStartup] http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

[load_elf_binary] http://elixir.free-electrons.com/linux/latest/source/fs/binfmt_elf.c#L682

[Love10] Robert Love, *Linux Kernel Development*, Addison-Wesley Professional, 2010

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

[Neumann66] von Neumann, John and Arthur W. Burks. 1966. *Theory of Self-Reproducing Automata*, Univ. of Illinois Press, Urbana IL.

[O'Neil16] Ryan 'elfmaster' O'Neill, *Learning Linux Binary Analysis*, Packt, 2016

[TheIntercept] https://theintercept.com/2017/04/14/leaked-nsa-malware-threatens-windows-users-around-the-world/

[Wikipedia_1] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[Wikipedia_2] https://en.wikipedia.org/wiki/Timeline_of_computer_viruses_and_worms

C++11 (and beyond) Exception Support

C++11 introduced many new exception related features. Ralph McArdell gives a comprehensive overview.

C++11 added a raft of new features to the C++ standard library and errors and exceptions were not left out.

In this article, we will start with a quick overview of the new exception types and exception related features. While the nitty gritty details are not covered in great depth, in most cases a simple usage example will be provided. The information was pulled together from various sources – [cppreference], [Josuttis12], [N3337] – and these, along with others, can be used to look up the in depth, detailed, specifics.

Following the lightning tour of C++11 exception support, we will take a look at some further usage examples.

Example code

The example code is available with a `Makefile` for building with GNU g++ 5.4.0 and MSVC++17 project files from: https://github.com/ralph-mcardell/article-cxx11-exception-support-examples

It may be useful to at least browse the source as the full example code is not always shown in the article.

For g++ each was built using the options:

```
-Wall -Wextra -pedantic -std=c++11 -pthread
```

For MSVC++17, a Win32 console application solution was created and each example source file added as a project using the default options, or with no pre-compiled header option selected if a project ended up with it turned on.

New standard library exception types

So let's start with a brief look at the new exception types. They are:

- `std::bad_weak_ptr` (include `<memory>`)
- `std::bad_function_call` (include `<functional>`)
- `std::bad_array_new_length` (include `<new>`)
- `std::future_error` (include `<future>`)
- `std::system_error` (include `<system_error>`)
- `std::nested_exception` (include `<exception>`)

Additionally, starting with C++11, `std::ios_base::failure` is derived from `std::system_error`.

`std::bad_weak_ptr` and `std::bad_function_call` are derived from `std::exception`.

`std::bad_weak_ptr` is thrown by the `std::shared_ptr` constructor that takes a `std::weak_ptr` as an argument if the `std::weak_ptr::expired` operation returns true (see Listing 1), which should produce output similar to:

```
Expired or default initialised weak_ptr:
bad_weak_ptr
```

Ralph McArdell has been programming for more than 30 years with around 20 spent as a freelance developer predominantly in C++. He does not ever want or expect to stop learning or improving his skills.

```
std::weak_ptr<int> int_wptr;
assert( int_wptr.expired() );
try{
  std::shared_ptr<int> int_sptr{ int_wptr };
}
catch ( std::bad_weak_ptr & e ){
  std::cerr
  << "Expired or default initialised weak_ptr: "
  << e.what() << "\n";
}
```

Listing 1

```
std::function<void()> fn_wrapper;
assert( static_cast<bool>(fn_wrapper)==false );
try{
  fn_wrapper();
}
catch ( std::bad_function_call & e ){
  std::cerr << "Function wrapper is empty: "
  << e.what() << "\n";
}
```

Listing 2

If a `std::function` object has no target, `std::bad_function_call` is thrown by `std::function::operator()` – see Listing 2 – with expected output:

```
(g++):     Function wrapper is empty:
           bad_function_call
(msvc++):  Function wrapper is empty:
           bad function call
```

`std::bad_array_new_length` is derived from `std::bad_alloc`, which means it will be caught by existing catch clauses that handle `std::bad_alloc` exceptions. It is thrown if an array size passed to new is invalid by being negative, exceeding an implementation defined size limit, or is less than the number of provided initialiser values (MSVC++17 does not seem to handle this case). It should be noted that this only applies to the first array dimension as this is the only one that can be dynamic thus any other dimension size values can be checked at compile time. In fact MSVC++17 is able to check the validity of simple literal constant size values for the first dimension as well during compilation, hence the use of the `len` variable with an illegal array size value in the example in Listing 3.

When run, we should see output like so:

```
(g++):     Bad array length:
           std::bad_array_new_length
(msvc++):  Bad array length:
           bad array new length
```

`std::future_error` is derived from `std::logic_error` and is used to report errors in program logic when using futures and promises,

error codes are lightweight objects encapsulating possibly implementation-specific error code values, while error conditions are effectively portable error codes

```
int len=-1; // negative length value
try{
  int * int_array = new int[len];
  delete [] int_array;
}
catch ( std::bad_array_new_length & e ){
  std::cerr << "Bad array length: "
            << e.what() << "\n";
}
```

Listing 3

for example trying to obtain a future object from a promise object more than once (see Listing 4), which should display:

```
     (g++):  Error from promise/future:
             std::future_error: Future already
             retrieved
 (msvc++):  Error from promise/future:
             future already retrieved
```

`std::system_error` is derived from `std::runtime_error` and is used to report operating system errors, either directly by our code or raised by other standard library components, as in the example in Listing 5, which on running should produce output along the lines of:

```
     (g++):  System error from thread detach:
             Invalid argument
 (msvc++):  System error from thread detach:
             invalid argument: invalid argument
```

```
std::promise<int> int_vow;
auto int_future = int_vow.get_future();
try{
  int_future = int_vow.get_future();
}
catch ( std::future_error & e ){
  std::cerr << "Error from promise/future: "
            << e.what() << "\n";
}
```

Listing 4

```
try{
  std::thread().detach(); // Oops, no thread to
}                         // detach
catch ( std::system_error & e ){
  std::cerr << "System error from thread detach: "
  << e.what() << "\n";
}
```

Listing 5

`std::nested_exception` is not derived from anything. It is a polymorphic mixin class that allows exceptions to be nested within each other. There is more on nested exceptions below in the 'Nesting exceptions' section.

Collecting, passing and re-throwing exceptions

Since C++11, there has been the capability to obtain and store a pointer to an exception and to re-throw an exception referenced by such a pointer. One of the main motivations for exception pointers was to be able to transport exceptions between threads, as in the case of `std::promise` and `std::future` when there is an exceptional result set on the promise.

Exception pointers are represented by the `std::exception_ptr` standard library type. It is, in fact, a type alias to some unspecified nullable, shared-ownership smart pointer like type that ensures any pointed to exception remains valid while at least one `std::exception_ptr` object is pointing to it. Instances can be passed around, possibly across thread boundaries. Default constructed instances are null pointers that compare equal to `nullptr` and test false.

`std::exception_ptr` instances must point to exceptions that have been thrown, caught, and captured with `std::current_exception`, which returns a `std::exception_ptr`. Should we happen to have an exception object to hand already, then we can pass it to `std::make_exception_ptr` and get a `std::exception_ptr` in return. `std::make_exception_ptr` behaves as if it throws, catches and captures the passed exception via `std::current_exception`.

Once we have a `std::exception_ptr`, it can be passed to `std::rethrow_exception()` from within a try-block to re-throw the exception it refers to.

The example in Listing 6 shows passing an exception simply via a (shared) global `std::exception_ptr` object from a task thread to the main thread. When built and run it should output:

```
     (g++):  Task failed exceptionally:
             Invalid argument
 (msvc++):  Task failed exceptionally:
             invalid argument: invalid argument
```

Of course, using global variables is questionable at best, so in real code you would probably use other means, such as hiding the whole mechanism by using `std::promise` and `std::future`.

Error categories, codes and conditions

`std::system_error` and `std::future_error` allow specifying an error code or error condition during construction. Additionally, `std::system_error` can also be constructed using an error category value.

In short, error codes are lightweight objects encapsulating possibly implementation-specific error code values, while error conditions are effectively portable error codes. Error codes are represented by

```
#include <exception>
#include <thread>
#include <iostream>
#include <cassert>

std::exception_ptr g_stashed_exception_ptr;

void bad_task(){
  try  {
    std::thread().detach(); // Oops !!
  }
  catch ( ... )  {
    g_stashed_exception_ptr =
    std::current_exception();
  }
}

int main(){
  assert( g_stashed_exception_ptr == nullptr );
  assert( !g_stashed_exception_ptr );
  std::thread task( bad_task );
  task.join();
  assert( g_stashed_exception_ptr != nullptr );
  assert( g_stashed_exception_ptr );

  try{
    std::rethrow_exception
      ( g_stashed_exception_ptr );
  }
  catch ( std::exception & e ){
    std::cerr << "Task failed exceptionally: "
              << e.what() << "\n";
  }
}
```

Listing 6

`std::error_code` objects, while error conditions are represented by `std::error_condition` objects.

Error categories define the specific error-code, error-condition mapping and hold the error description strings for each specific error category. They are represented by the base class `std::error_category`, from which specific error category types derive. There are several categories defined by the standard library whose error category objects are accessed through the following functions:

- `std::generic_category` for POSIX `errno` error conditions
- `std::system_category` for errors reported by the operating system
- `std::iostream_category` for IOStream error codes reported via `std::ios_base::failure` (which, if you remember, has been derived from `std::system_error` since C++11)
- `std::future_category` for future and promise related error codes provided by `std::future_error`

Each function returns a `const std::error_category&` to a static instance of the specific error category type.

The standard library also defines enumeration types providing nice-to-use names for error codes or conditions for the various error categories:

- `std::errc` defines portable error condition values corresponding to POSIX error codes
- `std::io_errc` defines error codes reported by IOStreams via `std::ios_base::failure`
- `std::future_errc` defines error codes reported by `std::future_error`

Each of the enumeration types have associated `std::make_error_code` and `std::make_error_condition` function overloads that convert a passed enumeration value to a `std::error_code` or `std::error_condition`. They also have an associated `is_error_condition_enum` or `is_error_code_enum` class specialisation to aid in identifying valid enumeration error condition or code types that are eligible for automatic conversion to `std::error_condition` or `std::error_code`.

In C++11 and C++14, `std::future_error` exceptions are constructed from `std::error_code` values. However, from C++17 they are constructed directly from `std::future_errc` enumeration values.

Nesting exceptions

At the end of the 'New standard library exception types' section above was a brief description of `std::nested_exception`, which can be used to allow us to nest one exception within another (and another, and another and so on if we so desire). This section takes a closer look at the support for handling nested exceptions.

While it is possible to use `std::nested_exception` directly, it is almost always going to be easier to use the C++ standard library provided support.

To create and throw a nested exception, we call `std::throw_with_nested`, passing it an rvalue reference to the outer exception object. That is, it is easiest to pass a temporary exception object to `std::throw_with_nested`. `std::throw_with_nested` will call `std::current_exception` to obtain the inner nested exception, and hence should be called from within the catch block that handles the inner nested exception.

Should we catch an exception that could be nested then we can re-throw the inner nested exception by passing the exception to `std::rethrow_if_nested`. This can be called repeatedly, possibly recursively, until the inner most nested exception is thrown where upon the exception is no longer nested and so `std::rethrow_if_nested` does nothing.

Each nested exception thrown by `std::throw_with_nested` is publicly derived from both the type of the outer exception passed to `std::throw_with_nested` and `std::nested_exception`, and so has an is-a relationship with both the outer exception type and `std::nested_exception`. Hence nested exceptions can be caught by catch blocks that would catch the outer exception type, which is handy.

The example in Listing 7 demonstrates throwing nested exceptions and recursively logging each to `std::cerr`.

The idea is that the code is performing some tasks and each task performs sub-tasks. The initial failure is caused by sub-task 4 of task 2 in the `sub_task4()` function. This is caught and re-thrown nested within a `std::runtime_error` exception by the `task2()` function which is then caught and re-thrown nested with another `std::runtime_error` by the `do_tasks` function. This composite nested exception is caught and logged in main by calling `log_exception`, passing it the caught exception reference.

`log_exception` first builds and outputs to `std::cerr` a log message for the immediate, outer most exception. It then passes the passed in exception reference to `std::rethrow_if_nested` within a try-block. If this throws, the exception had an inner nested exception which is caught and passed recursively to `log_exception`. Otherwise the exception was not nested, no inner exception is re-thrown and `log_exception` just returns.

When built and run the program should produce:

```
Outer exception: Execution failed performing tasks
  Nested exception: task2 failed performing sub
  tasks
    Nested exception: sub_task4 failed:
    calculation overflowed
```

Detecting uncaught exceptions

C++98 included support for detecting if a thread has a live exception in flight with the `std::uncaught_exception` function. A live

```
#include <exception>
#include <string>
#include <iostream>
#include <stdexcept>

void log_exception( std::exception const & e,
unsigned level = 0u ){
  const std::string indent( 3*level, ' ' );
  const std::string prefix( indent +
    (level?"Nested":"Outer") + " exception: " );
  std::cerr << prefix << e.what() << "\n";
  try{
    std::rethrow_if_nested( e );
  }
  catch ( std::exception const & ne )  {
    log_exception( ne, level + 1 );
  }
  catch( ... ) {}
}

void sub_task4(){
  // do something which fails...
  throw std::overflow_error{
    "sub_task4 failed: calculation overflowed" };
}

void task2(){
  try{ // pretend sub tasks 1, 2 and 3 are
       // performed OK...
    sub_task4();
  }
  catch ( ... ){
    std::throw_with_nested
      ( std::runtime_error{
        "task2 failed performing sub tasks" } );
  }
}

void do_tasks(){
  try{
    // pretend task 1 performed OK...
    task2();
  }
  catch ( ... ){
    std::throw_with_nested
      ( std::runtime_error{
        "Execution failed performing tasks" } );
  }
}

int main(){
  try{
    do_tasks();
  }
  catch ( std::exception const & e ){
    log_exception( e );
  }
}
```

Listing 7

exception is one that has been thrown but not yet caught (or entered `std::terminate` or `std::unexpected`). The `std::uncaught_exception` function returns true if stack unwinding is in progress in the current thread.

It turns out that `std::uncaught_exception` is not usable for what would otherwise be one of its main uses: knowing if an object's destructor is being called due to stack unwinding, as detailed in Herb Sutter's N4152 'uncaught exceptions' paper to the ISO C++ committee [N4152]. For this scenario knowing the number of currently live exceptions in a thread is required not just knowing if a thread has at least one live exception.

```
extern "C"{
  struct widget;
  enum status_t { OK, no_memory, bad_pointer,
    value_out_of_range, unknown_error };
  status_t make_widget
    ( widget ** ppw, unsigned v );
  status_t get_widget_attribute
    ( widget const * pcw, unsigned * pv );
  status_t set_widget_attribute
    ( widget * pw, unsigned v );
  status_t destroy_widget( widget * pw );
}
```

Listing 8

If an object's destructor only knows if it is being called during stack unwinding it cannot know if it is because of an exception thrown after the object was constructed, and so needs exceptional clean-up (e.g. a rollback operation), or if it was due to stack unwinding already in progress and it was constructed as part of the clean-up and so probably not in an error situation itself. To fix this an object needs to collect and save the number of uncaught exceptions in flight at its point of construction and during destruction compare this value to the current value and only take exceptional clean-up action if the two are different.

So, from C++17, `std::uncaught_exception` has been deprecated in favour of `std::uncaught_exceptions` (note the plural, 's', at the end of the name) which returns an `int` value indicating the number of live exceptions in the current thread.

Some additional usage scenarios

Now we have had a whizz around the new C++11 exceptions and exception features, let's look at some other uses.

Centralising exception handling catch blocks

Have you ever written code where you wished that common exception handling blocks could be pulled out to a single point? If so then read on.

The idea is to use a catch all `catch (…)` clause containing a call to `std::current_exception` to obtain a `std::exception_ptr` which can then be passed to a common exception processing function where it is re-thrown and the re-thrown exception handled by a common set of catch clauses.

Using the simple C API example shown in Listing 8 allows us to make widgets with an initial value, get and set the attribute value and destroy widgets when done with them. Each API function returns a status code meaning a C++ implementation has to convert any exceptions to `status_t` return code values. The API could be exercised as shown in Listing 9.

We could imagine a simple quick and dirty C++ implementation like Listing 10.

Not to get hung up on the specifics of the implementation and that I have added a `check_pointer` function to convert bad null pointer arguments to exceptions just for them to be converted to a status code, we see that the error handling in each API function is larger than the code doing the work, which is not uncommon.

Using `std::current_exception`, `std::exception_ptr` and `std::rethrow_exception` allows us to pull most of the error handling into a single function (Listing 11).

Now each function's `try` block only requires a `catch (…)` clause to capture the exception and pass it to the handling function and, for example, the `set_widget_attribute` implementation becomes Listing 12.

We can see that the implementation is shorter and, more importantly, no longer swamped by fairly mechanical and repetitive error handling code

```
int main( void ){
  struct widget * pw = NULL;
  assert(make_widget(NULL, 19u) == bad_pointer);
  assert(make_widget(&pw, 9u) ==
    value_out_of_range);
  if (make_widget(&pw, 45u) != OK)
    return EXIT_FAILURE;
  unsigned value = 0u;
  assert(get_widget_attribute(pw, &value) == OK);
  assert(get_widget_attribute(NULL, &value) ==
    bad_pointer);
  assert(value == 45u);
  assert(set_widget_attribute(pw, 67u) == OK);
  assert(set_widget_attribute(NULL, 11u) ==
    bad_pointer);
  assert(set_widget_attribute(pw, 123u) ==
    value_out_of_range);
  get_widget_attribute(pw, &value);
  assert(value == 67u);
  assert(destroy_widget(pw) == OK);
  assert(destroy_widget(NULL) == bad_pointer);
}
```

Listing 9

translating exceptions into error codes, all of which is now performed in the common **handle_exception** function.

We can reduce the code clutter even more, at the risk of potentially greater code generation and call overhead on the good path, by using the execute-around pattern [Vijayakumar16] (more common in languages like Java and C#) combined with lambda functions. (thanks to Steve Love [Love] for mentioning execute around to me at the ACCU London August 2017 social evening).

```
namespace{
  void check_pointer( void const * p )  {
    if ( p==nullptr )
      throw std::invalid_argument("bad pointer");
  }
}

extern "C"{
  struct widget{

  private:
    unsigned attrib = 10u;

  public:
    unsigned get_attrib() const { return attrib; }
    void set_attrib( unsigned v ){
      if ( v < 10 || v >= 100 )
        throw std::range_error
          ( "widget::set_widget_attribute:
            attribute value out of range
            [10,100)" );
      attrib = v;
    }
  };

  status_t make_widget( widget ** ppw,
  unsigned v ){
    status_t status{ OK };
    try{
      check_pointer( ppw );
      *ppw = new widget;
      (*ppw)->set_attrib( v );
    }
```

Listing 10

```
    catch ( std::invalid_argument const & ){
      return bad_pointer;
    }
    catch ( std::bad_alloc const & ){
      status = no_memory;
    }
    catch ( std::range_error const & ){
      status = value_out_of_range;
    }
    catch ( ... ){
      status = unknown_error;
    }
    return status;
  }

  status_t get_widget_attribute
  ( widget const * pcw, unsigned * pv ){
    status_t status{ OK };
    try{
      check_pointer( pcw );
      check_pointer( pv );
      *pv = pcw->get_attrib();
    }
    catch ( std::invalid_argument const & ){
      return bad_pointer;
    }
    catch ( ... ){
      status = unknown_error;
    }
    return status;
  }

  status_t set_widget_attribute( widget * pw,
  unsigned v ){
    status_t status{ OK };
    try{
      check_pointer( pw );
      pw->set_attrib( v );
    }   catch ( std::invalid_argument const & ){
      return bad_pointer;
    }
    catch ( std::range_error const & ){
      status = value_out_of_range;
    }
    catch ( ... ){
      status = unknown_error;
    }
    return status;
  }

  status_t destroy_widget( widget * pw ){
    status_t status{ OK };
    try{
      check_pointer( pw );
      delete pw;
    }
    catch ( std::invalid_argument const & ){
      return bad_pointer;
    }
    catch ( ... ){
      status = unknown_error;
    }
    return status;
  }
}
```

Listing 10 (cont'd)

The idea is to move the work-doing part of each function, previously the code in each of the API functions' try-block, to its own lambda function

```
namespace{
  status_t handle_exception
  ( std::exception_ptr ep ){
    try{
      std::rethrow_exception( ep );
    }
    catch ( std::bad_alloc const & ){
      return no_memory;
    }
    catch ( std::range_error const & ){
      return value_out_of_range;
    }
    catch ( std::invalid_argument const & ){
      // for simplicity we assume all bad
      // arguments are bad pointers
      return bad_pointer;
    }
    catch ( ... ){
      return unknown_error;
    }
  }
}
```

Listing 11

and pass an instance of this lambda to a common function that will execute the lambda within a try block which has the common exception catch handlers as in the previous incarnation. As each lambda function in C++ is a separate, unique, type we have to use a function template, parametrised on the (lambda) function type (Listing 13).

The form of each API function implementation is now shown by the third incarnation of the **`set_widget_attribute`** implementation (Listing 14).

Using nested exceptions to inject additional context information

As I hope was apparent from the 'Nesting exceptions' section above, nested exceptions allow adding additional information to an originally thrown (inner most) exception as it progresses through stack unwinding.

Of course, doing so for every stack frame is possible but very tedious and probably overkill. On the other hand, there are times when having some additional context can really aid tracking down a problem.

One area I have found that additional context is useful is threads. You have an application, maybe a service or daemon, that throws an exception in a worker thread. You have carefully arranged for such exceptions to be captured at the thread function return boundary and set them on a promise so a monitoring thread (maybe the main thread) that holds the associated future can re-throw the exception and take appropriate action which always includes logging the problem.

You notice that an exception occurs in the logs, it is a fairly generic problem – maybe a **`std::bad_alloc`** or some such. At this point, you are wondering which thread it was that raised the exception. You go back to your thread wrapping code and beef up the last-level exception handling to wrap any exception in an outer exception that injects the thread's contextual information and hand a **`std::exception_ptr`** to the resultant nested exception to the promise object.

The contextual information could include the thread ID and maybe a task name. If the thread is doing work on behalf of some message or event then such details should probably be added to the outer exception's message as these will indicate what the thread was doing.

Of course, the thread exit/return boundary is not the only place such contextual information can be added. For example in the event case mentioned above it may be that adding the message / event information is better placed in some other function. In this case you may end up with a three-level nest exception set: the original inner most exception, the middle event context providing nested exception and the outer thread context providing nested exception.

```
  status_t set_widget_attribute( widget * pw,
  unsigned v ){
    status_t status{ OK };
    try{
      check_pointer( pw );
      pw->set_attrib( v );
    }
    catch ( ... ){
      status = handle_exception
      ( std::current_exception() );
    }
    return status;
  }
```

Listing 12

```
template <class FnT>
status_t try_it( FnT && fn ){
  try{
    fn();
  }
  catch ( std::bad_alloc const & ){
    return no_memory;
  }
  catch ( std::range_error const & ){
    return value_out_of_range;
  }
  catch (std::invalid_argument const & ){
    // for simplicity we assume all bad
    // arguments are bad pointers
    return bad_pointer;
  }
  catch ( ... ){
    return unknown_error;
  }
  return OK;
}
```

Listing 13

```
status_t set_widget_attribute
( widget * pw, unsigned v ){
  return try_it( [pw, v]() -> void{
      check_pointer( pw );
      pw->set_attrib( v );
    }
  );
}
```

Listing 14

Error codes of your very own

I saw the details of this usage example explained quite nicely by a blog post of Andrzej Krzemienski [Krzemienski17] that was mentioned on ISO Cpp [ISO].

The cases where this is relevant are those where a project has sets of error values, commonly represented as enumeration values. Large projects may have several such enumeration types for different subsystems and the enumeration values they employ may overlap. For example, we might have some error values from a game's application engine and its renderer sub-system (Listing 15 and Listing 16).

Note: The error types and values were adapted from panic value types from a simple noughts and crosses (tic tac toe) game I wrote with a friend more than a decade ago with the goal of learning a bit about Symbian mobile OS development.

```
namespace the_game{
  enum class appengine_error{
    no_object_index  = 100
  , no_renderer
  , null_draw_action = 200
  , bad_draw_context = 300
  , bad_game_object
  , null_player      = 400
  };
}
```

Listing 15

In such cases we can either deal in the enumeration types directly when such error values are passed around with the effect that the various parts of the project need access to the definitions of each enumeration type they come into contact with. Or we can use a common underlying integer type, such as `int`, for passing around such error value information and lose the ability to differentiate between errors from different subsystems or domains that share the same value.

Note: It would be possible to use different underlying types for each of the various error value sets but there are only so many and such an approach seems fragile at best given the ease with which C++ converts/promotes between fundamental types and the need to ensure each enumeration uses a different underlying type.

If only C++ had an error code type as standard that would allow us to both traffic in a single type for error values and allow us to differentiate between different sets of errors that may use the same values. If we could also assign a name for each set and text descriptions for each error value that would be icing on the cake. Oh, wait, it does: `std::error_code`. We just have to plug our own error value enumeration types into it. The only caveats are that all the underlying values be correctly convertible to `int` and that our custom error types must reserve an underlying value of 0 to mean OK, no error. Even if our error value types do not provide an OK enumeration value of 0 explicitly so long as a value of 0 is not reserved for an error value then we can always create a zero valued instance of the error enum:

```
  the_game::appengine_error ok_code_zero_value{};
```

Different error value sets or domains are called error categories by the C++ standard library and to completely define an error code we require an {error value, error category} pair.

To create our own error categories, we define a specialisation of `std::error_category` for each error value set we have. To keep `std::error_code` lightweight, it does not store a `std::error_category` object within each instance. Rather each `std::error_category` specialisation has a single, static, instance. `std::error_code` objects contain the error value and a reference (pointer) to the relevant `std::error_category` specialisation static instance. Because all references to an error category type instance refer to the same, single instance of that type, the object's address can be used to uniquely identify and differentiate each specific error category and allows `std::error_code` objects to be compared.

```
namespace the_game{
  enum class renderer_error{
    game_dimension_too_small = 100
  , game_dimension_bad_range
  , board_too_small          = 200
  , board_bad_range
  , game_dimension_bad
  , board_not_square         = 300
  , board_size_bad
  , bad_region               = 400
  , cell_coordinate_bad      = 500
  , new_state_invalid
  , prev_state_invalid
  };
}
```

Listing 16

```
struct appengine_error_category :
std::error_category{
  const char* name() const noexcept override;
  std::string message(int ev) const override;
};

const char* appengine_error_category::name()
const noexcept{
  return "app-engine";
}

std::string appengine_error_category::message
   ( int ev ) const{
  using the_game::appengine_error;

  switch( static_cast<appengine_error>(ev) ){
    case appengine_error::no_object_index:
      return "No object index";
    case appengine_error::no_renderer:
      return "No renderer currently set";
    case appengine_error::null_draw_action:
      return "Null draw action pointer";
    case appengine_error::bad_draw_context:
      return "Draw action context has null
      graphics context or renderer pointer";
    case appengine_error::bad_game_object:
      return "Draw action context has null game
      object pointer";
    case appengine_error::null_player:
      return "Current player pointer is null";
    default:
      return "?? unrecognised error ??";
  }
}
```

Listing 17

Each `std::error_category` specialisation provides overrides of the `name` and `message` pure virtual member functions. The `name` member function returns a C-string representing the name of the category. The `message` member function returns a `std::string` describing the passed in category error value (passed as an `int`). For example, an error category type for the `the_game::appengine_error` error values might look like Listing 17.

To create `std::error_code` values from a custom error (enumeration) value in addition to the `std::error_category` specialisation, we need to provide two other things. First, an overload of `std::make_error_code` that takes our error value type as a parameter and returns a `std::error_code` constructed from the passed error value and the static `std::error_category` specialisation object. This should be in the same namespace as our error value enum type.

In this use case, the `std::make_error_code` function overload is the only thing that requires access to the custom error category static instance. As such we can define the static object to be local to the `std::make_error_code` function overload, as in Listing 18.

```
namespace the_game{
  std::error_code make_error_code
     (appengine_error e){
    static const appengine_error_category
      theappengine_error_categoryObj;
    return {static_cast<int>(e),
      theappengine_error_categoryObj};
  }
}
```

Listing 18

```
namespace std{
  using the_game::appengine_error;
  template <> struct
    is_error_code_enum<appengine_error>
    : true_type {};
}
```

Listing 19

As the **std::make_error_code** function overload definition is the only thing that requires the definition of the **std::error_category** specialisation it is probably best if they are both placed in the same implementation file. The declaration can be placed in the same header as the custom error value enumeration type definition as it will be used when converting such values to **std::error_code** instances – the appengine_error.h header for the **appengine_error** example case.

Second, we need to provide a full specialisation of the **std::is_error_code_enum** struct template, specifying our error code type as the template parameter. The easiest implementation is to derive from **std::true_type** and have an empty definition. This should be in the **std** namespace, one of the few things application code can add to **std**. Listing 19 shows the **std::is_error_code_enum** specialisation for **the_game::appengine_error**.

It is also probably best placed in the same header as the custom error values enumeration type definition.

Subsystem API (member) functions can then pass around **std::error_code** instances rather than specific enumeration types or simple integer values that loose the category information. Producers of such error codes need to include both **system_error** for **std::error_code** and the header containing the error value enum definition, along with the **std::make_error_code** overload *declaration* (only) and the **std::is_error_code_enum** struct template specialisation definition. So to produce **std::error_code** objects from **the_game::appengine_error** values, the previously mentioned appengine_error.h header would need to be included.

Consumers need only include **system_error** for **std::error_code** and will still be able to access the error value, category name and error value description string.

For example some spoof game **appengine** implementation code for updating the game board might complain if it does not have an associated renderer object to pass on the request to by returning a **the_game::appengine_error::no_renderer** error converted to a **std::error_code** (Listing 20).

It thus needs to include the appengine_error.h header and well as **system_error**. However, the caller of this member function only sees the returned **std::error_code**, and so only needs to include **system_error**, as well as any **appengine** API headers of course. This is demonstrated by the simple spoof usage program in Listing 21, which shows spoof usage for both converted **the_game::renderer_error** values and the **the_game::appengine_error** values I have shown examples of. When built and run the output should be:

```
  renderer:101 Reported max. supported game grid
  less than the min.
  app-engine:101 No renderer currently set
```

```
std::error_code appengine::update_game_board(){
  // good case demonstrates zero-initialising
  // enum class instance
  return rp_ ? appengine_error{} :
  appengine_error::no_renderer;
}
```

Listing 20

```
#include "custom_error_code_bits/the_game_api.h"
#include <system_error>
#include <iostream>
#include <string>

void log_bad_status_codes( std::error_code ec ){
  if ( ec )
    std::clog << ec << " " << ec.message()
              << "\n";
}

int main(){
  auto & engine{ the_game::get_appengine() };

  // Should fail as setting renderer supporting
  // invalid dimension range
  std::unique_ptr<the_game::renderer> rend{
    new the_game::oops_renderer};
  log_bad_status_codes( engine.take_renderer(
    std::move(rend) ) );

  // Should fail as no renderer successfully set to
  // draw board
  log_bad_status_codes
    ( engine.update_game_board() );

  // OK - nothing to report, this renderer is fine
  // and dandy
  rend.reset( new the_game::fine_renderer );
  log_bad_status_codes( engine.take_renderer
    ( std::move(rend)) );

  // OK - now have renderer to render board updates
  log_bad_status_codes
    ( engine.update_game_board() );
}
```

Listing 21

Of course this is all about error values, codes and categories, nothing about exceptions (other than returning **std::error_code** values could allow functions to be marked **noexcept**). Remember however that we can always construct a **std::system_error** exception object from a **std::error_code** object. ■

References

[cppreference] http://en.cppreference.com

[ISO] Cpp: https://isocpp.org/

[Josuttis12] Josuttis, Nicolai M. (2012) *The C++ Standard Library*, second edition, Addison Wesley Longman

[Krzemienski17] Your own error code, Andrzej Krzemienski: https://akrzemi1.wordpress.com/2017/07/12/your-own-error-code/

[Love] Steve Love: https://uk.linkedin.com/in/steve-love-1198994

[N3337] Post C++11 Working Draft, Standard for Programming Language C++

[N4152] Herb Sutter, uncaught_exceptions: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4152.pdf

[Vijayakumar16] Design Patterns – Execute Around Method Pattern: http://www.karthikscorner.com/sharepoint/design-patterns-execute-around-method-pattern/

Afterwood

Too soon! Chris Oldwood reviews optimisation in the development process.

The most famous quote about optimisation, at least in programming circles, is almost certainly "premature optimisation is the root of all evil". When I was growing up, this was attributable to Donald Knuth, but he 'fessed up saying that he was just quoting Sir Tony Hoare, although Sir Tony seemed reluctant to claim ownership. According to the fount of all knowledge, Wikipedia, things appear to have been straightened out now and Sir Tony has graciously accepted attribution. That quote was originally about the performance of code and should ideally be presented in its greater detail so as not to lose the context in which it was said. The surrounding lines "We should forget about small efficiencies, say about 97% of the time: […]. Yet we should not pass up our opportunities in that critical 3%." remind us that there is a time and a place for optimisation.

Of course, we're all agile these days and do not pamper to speculative requirements – we only consider performance when there is a clear business need to. Poppycock! Herb Sutter and Andrei Alexandrescu certainly didn't believe such nonsense and popularised the antonym 'pessimization' in the process. Item 9 in their excellent book *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* tells us not to prematurely pessimize either; i.e. we shouldn't go out of our way to avoid appearing to prematurely optimise by simply writing stupid code – choosing a sensible design or algorithm is not premature optimisation. For most of us, the sorting algorithm was a problem solved years ago and the out-of-the box Quicksort variation (something else attributable to Sir Tony Hoare) that we get in our language runtime is almost certainly an excellent starting point.

Another favourite quote on the subject of optimisation comes from Rob Pike where he tells us "fancy algorithms are slow when N is small, and N is usually small". While there are many new products which aim to scale to the heady heights of Twitter and Facebook, most are destined for a user base many orders of magnitude lower than them. Whilst it's all very interesting to read up on how a company like Facebook has to design its own hardware to deal with its scaling issues, that's definitely something for the back pocket in case you really do end up on The Next Big Thing rather than an architectural stance which you should adopt by default.

On Twitter, John Carmack once extrapolated from Sir Tony and observed that performance is not the only thing which we can be accused of prematurely optimising: "you can prematurely optimize maintainability, flexibility, security, and robustness [too]". Although I didn't realise it at first, I eventually discovered that my own C++ unit testing framework, which I thought I was being super clever with by eliding all those really difficult bits, like naming, was actually a big mistake. By focusing so heavily on the short term goals I had written a framework that was optimised for writing tests, but not reading them. As such, many of my earliest unit tests were shockingly incoherent mere days later and not worth the (virtual) paper they were written on. Every time I revisited them I probably spent orders of magnitude more time trying to understand them than what it would have taken in the first place to slow down and write them more lucidly.

Outside the codebase, the development process is another area where it's all too easy to end up optimising for the wrong audience. The primary measure of progress should be working software, and yet far too much effort can be put into finding proxies for this metric. Teams that choose internally to track their work using tools and metrics to help them improve their rate of delivery are laudable, whereas imposing complex work tracking processes on a team to make it easier to measure progress from afar is unproductive. For example The Gemba Walk – the practice of management directly observing the work – allows those doing the work to dedicate more of their time to generating value rather than finding arbitrary ways to represent their progress.

Tooling is a common area where a rift between those who do and those who manage arises. For example, it's not uncommon to find effort duplicated between the source code or version control system and the work tracking tool because the management wants it in a form they can easily consume, even if that comes at the expense of more developer time. For example I've seen code diffs and commit comments pasted into change request Word documents for review, and acceptance criteria in Gherkin synchronised between JIRA stories and test code because those in control get to call the shots on the format of any handover so they can minimise their own workload.

Engineering is all about trade-offs, both within the product itself and the process used to deliver it. As software engineers, we might find it easier to trade off the dimensions of time and space in the guise of CPU cycles and memory and find it harder to weigh-up, or more likely, control, trading off time between our present and future selves. The increasing recognition of metaphors like Technical Debt and management styles such as Servant Leadership will continue to help raise the profile of some of the more common sources of tension, but we still need to be on the lookout for those moments where our apparent cleverness may really be a rod for our own backs. ■

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

JOIN tHE ACCU!

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

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

With full ACCU membership you get:

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

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

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

How to join

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

Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

GET MOORE
intel
Software
intel
PARALLEL STUDIO XE
£634.99
TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER
Take your results to the next level with screaming-fast code.

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

To find out more about Intel products please contact us:

020 8733 7101 | enquiries@qbssoftware.com
www.qbssoftware.com/parallelstudio

qbs SOFTWARE