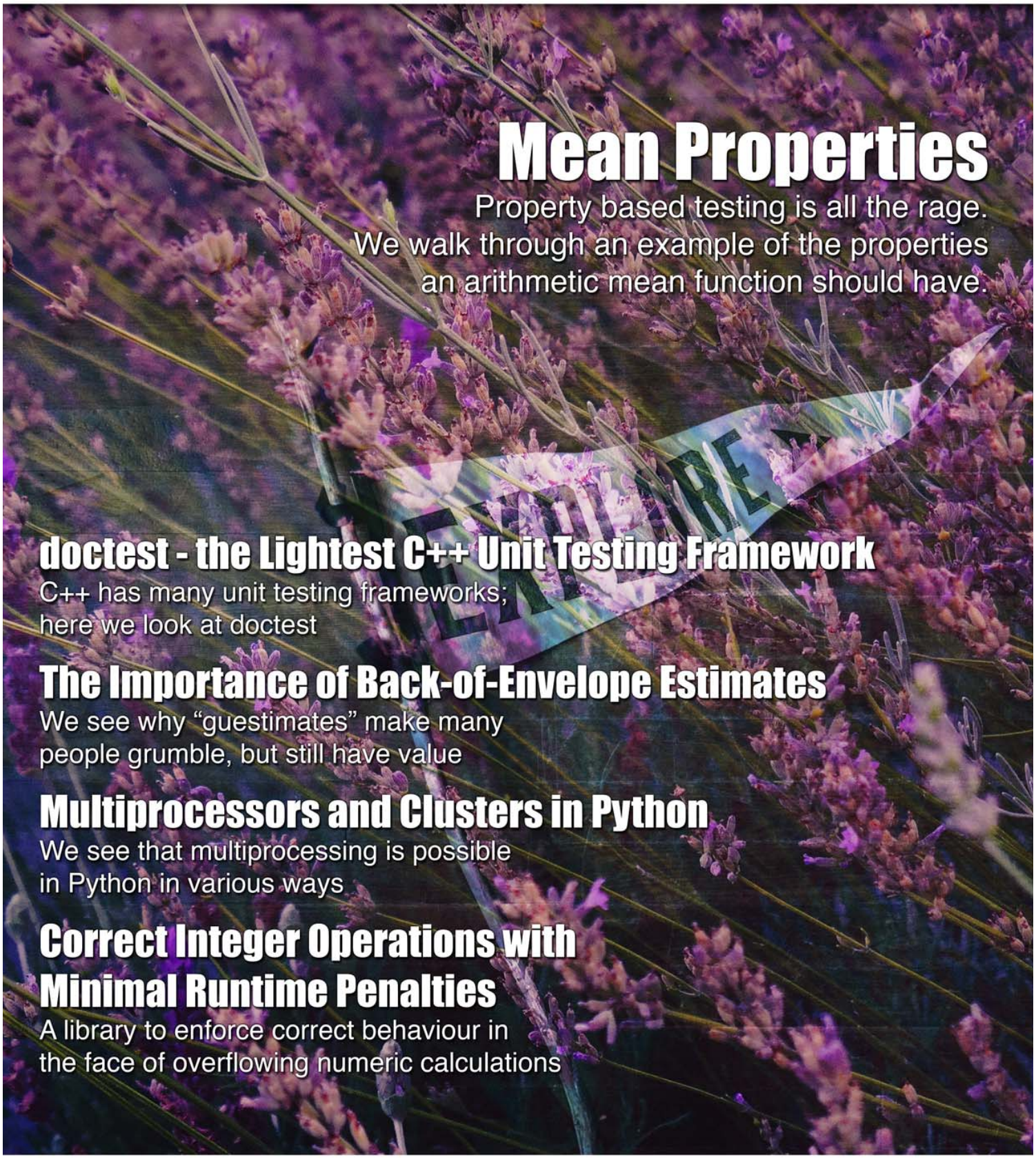
overload 137

FEBRUARY 2017 £3



Mean Properties

Property based testing is all the rage. We walk through an example of the properties an arithmetic mean function should have.

doctest - the Lightest C++ Unit Testing Framework

C++ has many unit testing frameworks; here we look at doctest

The Importance of Back-of-Envelope Estimates

We see why "guestimates" make many people grumble, but still have value

Multiprocessors and Clusters in Python

We see that multiprocessing is possible in Python in various ways

Correct Integer Operations with Minimal Runtime Penalties

A library to enforce correct behaviour in the face of overflowing numeric calculations

A magazine of ACCU

ISSN: 1354-3172

JET BRAINS

A Power Language Needs Power Tools

We at JetBrains have spent the last decade and a half helping developers code better faster, with intelligent products like IntelliJ IDEA, ReSharper and YouTrack. Finally, you too have a C++ development tool that you deserve:

- Rely on safe C++ code refactorings to have all usages updated throughout the whole code base
- Generate functions and constructors instantly
- Improve code quality with on-the-fly code analysis and quick-fixes

ReSharper C++

Visual Studio Extension for C++ developers

CLion

Cross-platform IDE for C and C++ developers

AppCode

IDE for iOS and OS X development

Find a C++ tool for you
jb.gg/cpp-accu

OVERLOAD 137

February 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 138 should be submitted by 1st March 2017 and those for Overload 139 by 1st May 2017.

The ACCU

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

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

Overload is a publication of the ACCU

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

4 Mean Properties
Russel Winder walks us through an example of properties an arithmetic mean function should have.

8 The Importance of Back-of-Envelope Estimates
Sergey Ignatchenko reminds us why back of the envelope calculations matter.

13 Multiprocessors and Clusters in Python
Silas S. Brown shows us various ways to do multiprocessing in Python.

16 doctest - the Lightest C++ Unit Testing Framework
Viktor Kirilov introduces doctest, a C++ unit testing framework.

20 Correct Integer Operations with Minimal Runtime Penalties
Robert Ramey introduces a library to enforce correct numerical calculations.

28 Afterwood
Chris Oldwood reminisces on various approaches to finding a good candidate for a job.

Copyrights and Trade Marks

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

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

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

The Uncertainty Guidelines

Uncertainty can be overwhelming. Frances Buontempo embraces the advantages of ambiguity.

Having a few days off over Christmas has given me the chance to catch up on some reading, including revisiting *Gödel, Escher, Bach* [Hofstadter]. In turn, this has absorbed rather a lot of time, so I haven't got round to writing an editorial. I was also given a Sudoku Rubik's cube, which I haven't managed to complete yet. Tips welcome. I know I will have to resort to some group theory and a sensible notation on paper rather than just randomly trying moves and hoping for the best, though that is a good way to start to explore a puzzle. As ever, I digress. Near the start of his book, Hofstadter presents a puzzle he calls 'MU'. You form valid strings from three characters, {M, I, U}. Starting with 'MI' can you form 'MU' using the following four rules?

1. *x*I => *x*IU
2. M*x* => M*xx*
3. *x*IIIy => *x*U*y*
4. *x*UU*y* => *xy*

where *x* and *y* stand for any string of characters. I haven't solved this puzzle either, though a bit of trial and error reveals starting with 'MI', or indeed any string starting with M will only form strings starting with M, since none of the rules remove an M. MU is plausible, but it is neither self-evident nor certain that it can be made from MI. Of course, it might be impossible. How, in general, can you be certain that something is impossible? If you have a finite set of possibilities you can enumerate all of them and check the claimed impossibility is not held within. In the case of MU, there are infinite possible strings, so we cannot do this. Sometimes by assuming the impossible, you can derive a contradiction, from which all reasonable people would conclude the starting point is impossible. There are other options. I will leave the reader to explore the puzzle themselves and drawn their own conclusions or cheat [Wikipedia].

The 'Sudokube' is probably possible. The internet assures me if I remind myself how to solve a Rubik's cube it will be simple enough. I have a confession – I never fully learnt how to do this. I was shown some instructions, transformed into group theory notation, when I was about twelve, yet not knowing group theory made these less than useful. Transforming the situation into another is often a successful way to solve a problem, provided you transform it into a known problem. Finding a helpful notation allows you to think more abstractly and drawn conclusions about blind alleyways and potentially fruitful paths. Abstracting and generalising can be powerful tools. We do this when we write functions. Usually, we don't have a lookup table of all the possible results, but perform a transformation or calculation to return an answer. Of course, we sometimes then resort to lookup tables to speed things up later, but that is another matter. As programmers, we frequently solve puzzles or problems. The way the problem is presented tends to influence the outcome.

If the person setting the problem frames it in terms of how they would like the problem to be solved, it is useful to ask, 'What problem are you really trying to solve?' or simply 'Why?' This type of issue is sometimes referred to as the XY problem, and frequently mentioned on Stack Overflow. It has been described as:

> a mental block which leads to enormous amounts of wasted time and energy, both on the part of people asking for help, and on the part of those providing help. It often goes something like this:
> - User wants to do X.
> - User doesn't know how to do X, but thinks they can fumble their way to a solution if they can just manage to do Y.
> - User doesn't know how to do Y either.
> - User asks for help with Y.
> - Others try to help user with Y, but are confused because Y seems like a strange problem to want to solve.
> - After much interaction and wasted time, it finally becomes clear that the user really wants help with X, and that Y wasn't even a suitable solution for X.' [XyProblem]

I am not suggesting spending all your time being a sceptic or doubter; however, it is worth trying to find the right levels of abstraction to frame a question or answer. As I said, transforming something into a problem with a known solution can help, but it must be 'isomorphic' (or at least similar enough) to the original problem. If you admit uncertainty when you are not sure, this can be fruitful. If you can find a way to express the problem you are considering, be that in interfaces, function signatures or a concise mathematical formulation you at least have a way to discuss and explore the conundrum.

I have another confession; I never managed to learn my times tables properly at school. Some were easy because they had nice patterns, for example the nine times table. I tended to count up (or down) from the numbers I did know, by some combination of lookup and algebra. If I was uncertain of my answer, I would try another combination, say 7 times 8 stumped me, then 8 times 7 was far easier being double 4 sevens. Realising the general principle that natural numbers are commutative under multiplication

$$x \times y = y \times x$$

is powerful. Proving such general principles is another matter. In fact, it is possible to develop number systems which are not commutative, specifically Hamilton's quaternions, which take the form

$$a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$$

where $a,b,c,d \in \mathbb{R}$ or $\mathbb{C}$ with $\boldsymbol{i}^2=\boldsymbol{j}^2=\boldsymbol{k}^2=\boldsymbol{ijk}=-1$. Those using complex numbers $\mathbb{C}$ are known as bi-quaternions. You cannot be certain of commutativity; it is a property of basic arithmetic, rather than something provable. On the face of it, developing new number systems might seem like a pointless mind game. What's their use? On one level this should not matter; theoretical things can be interesting in and of themselves. Nonetheless, if you insist, the quaternions provide a neat shortcut in theoretical physics to represent the Lorenz group of special relativity.

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.

They also provide quicker ways to calculate transformations in computer graphics, compared to matrices. Surprising things happen when you question things that seem certain, obvious or no-brainers.

Revisiting Hofstadter reminded me of Euclid's fifth axiom. I remember it as given a 'straight line' and a point not on the line, there is only one way to draw a line through that point which doesn't cross the line, though this appears to be Playfair's axiom [Playfair]. The fifth axiom, or parallel postulate, is more usually stated as

> If a straight line crossing two straight lines makes the interior angles on the same side less than two right angles, the two straight lines, if extended indefinitely, meet on that side on which are the angles less than the two right angles. [Cut-the-knot]

For the MU puzzle, MI was given as an axiom; you assume it is 'true' or take it as given and see what follows. We can do likewise with Euclid's axioms, and derive theorems of Euclidean geometry, such as the internal angles of a triangle sum to 180°. The thing about axioms or principles is they are more like guidelines. The inner child shouts 'But why?' then finds out what happens if you disobey the claimed 'rules'. By dropping Euclid's parallel postulate, you develop or discover so-called Non-Euclidean geometries. For a hyperbolic geometry, parallel lines get further apart, whereas for an elliptical geometry they get closer together. The fifth postulate got picked on because it seemed untidy compared to the others, which in essence defined a straight line, circles and right angles. Many people tried to prove the fifth postulate, using just the first four and all failed. In the process, counterintuitive, yet consistent geometries were discovered. Poincare accepted these counterintuitive ideas, paving the way for Einstein [Barbosa]. It is worth noting that special relativity's use of non-Euclidean geometry is not equivalent to saying it is 'true' or correct, just that the more counterintuitive mathematical setting made the scientific theory simpler. This echoes the quaternions; a different representation with fewer or different constraints can give more powerful or simpler ways to approach puzzles.

Many scientific theories are referred to as 'Laws', for example Kepler's laws of planetary motion. This suggests they are unbreakable and in some sense fixed. Newton took on board Kepler's ideas, and talks of Kepler's guesses, deductions and discoveries; he did not describe these as laws. Any ideas, be that scientific or otherwise, tended to be described as 'philosophy' at the time. It seems Voltaire first introduced the term 'Law' of Kepler's philosophy, where he describes the area rule as a 'Law inviolably observed by all the Planets' [Wilson]. Using the word law suggests either a divine decree, or a rule following from the essence or nature of the planets and indeed space. They are fundamental principles, which fit observation. Most people would just take the term on board nowadays, without giving it too much thought. We tend to regard scientific models as something which fits observations, but expect them to change and evolve over time. Our thinking about science, and indeed thinking itself, changes over time. Similarly, our approach to coding has changed over time. Some things are driven by trends or new language features. Other things are more fundamental; the move to structured code changed how we wrote and reasoned about code. Introduction of object oriented programming has an effect. Attempting to write in a functional language influences how you solve a problem. I wonder what we might discover if we looked at coding standards through the last decades? Would we see trends and changes? Perhaps they should actually be referred to as conventions, or guidelines. Some laws are more like models, and others an attempt to enforce a norm. They can still be questioned or modified over time. They may be governed by some guiding principles, like fit our current observations, communicate clearly, or be nice. Laws and conventions give ways to assess and reason about things, but should never be set in stone.

In general, our syntax and abstractions allow us to frame problems in different ways, which in turn can make analysis easier or, if we are not careful, more difficult. We see this when we try to add features to code, or even test it. Sometimes an abstraction introduced in just the right place allows us to hook something different in, be that a test or a new feature. Sometimes this was a deliberate choice from a developer, or we just got lucky. Kevlin Henney [Henney] wrote about what he coined 'The Uncertainty Principle' for *Overload* a few years ago. In particular, he said, 'in software development, a lack of certainty about something can be part of the solution rather than part of the problem.' Rather than having a long meeting and countless arguments when a choice is presented, he advocates structuring your code so it doesn't matter which is chosen. Hiding the choice between an algorithm or lookup table behind an interface helps to 'mark out the boundaries in a software system and loosen the coupling'. Using uncertainty as a positive force is a great guideline.

Uncertainty can be unnerving, but if you embrace it and remember all the times it has driven new discoveries this should give you hope. We don't know everything, and there is always room for improvement. Let's see what chaos, new discoveries and surprising, unpredicted results the New Year brings.

References

[Barbosa] Pedro M. Rosario Barbosa *The Relation between Formal Science and Natural Science: Underdetermination of Science Project* http://pmrb.net/uos/?q=4_3_2

[Cut-the-knot] http://www.cut-the-knot.org/triangle/pythpar/Fifth.shtml

[Henney] 'The Uncertainty Principle', *Overload* 115, June 2013 https://accu.org/index.php/journals/1854

[Hofstadter] Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, New York: Basic Books, 1979.

[Playfair] Playfair, J. *Elements of Geometry: Containing the First Six Books of Euclid, with a Supplement on the Circle and the Geometry of Solids to which are added Elements of Plane and Spherical Trigonometry*. New York: W. E. Dean, 1861. (according to http://mathworld.wolfram.com/PlayfairsAxiom.html)

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

[Wilson] Wilson, Curtis 'Kepler's Laws, So-Called', *HAD News* (Historical astronomy division of the American Astronomical Society, May 1994. https://had.aas.org/sites/had.aas.org/files/HADN31.pdf

[XyProblem] http://mywiki.wooledge.org/XyProblem

Mean Properties

Property based testing is all the rage. Russel Winder walks us through an example of properties an arithmetic mean function should have.

In the article Testing Propositions [Winder16], I used the example of *factorial* to introduce proposition-based testing. One of the criticisms from an unnamed reviewer was that it was not clear what constituted a proposition; the test code looked very much like the implementation code, confusing the message. The reviewer had clearly missed the point, which most likely must indicate a problem with the presentation and/or the example chosen in the article. This short article is to try and present an example to address that reviewer's valid, and important, point.

A really (really) small problem

Let us consider the calculation of the mean of a set of data. Mathematically we would write:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

where $\overline{x}$ represents the mean of the dataset comprising all the values x_i. The mathematical statement leads us inexorably to an algorithm for computing the mean of a given data set: using Python[1] we implement a function **mean** as shown in Listing 1. Of course many people might have just written the code as shown in Listing 2 and whilst correct, this code is likely to be much slower than using NumPy.[2]

The question now is obviously: how can we test these implementations?

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

Listing 1

```
# python_pure.py
def mean(data):
  return sum(data) / len(data)
```

Listing 2

Do we have to?

The insightful reader will already have spotted that there is probably not a testing obligation for us with the **mean** function as implemented in Listing 1. The code uses assignment (which should work because the Python compiler and virtual machine[3] implementers have tested that it works correctly) and a reference to **numpy.mean** which should work because the NumPy implementers should have tested that *that* function works correctly.

But what about Listing 2? Is there a testing obligation given that the **sum** function, the **len** function and the **/** (division) operator are all Python features and the Python language people should have tests for all of them? Is this function not correct simply by observation, and that it reflects so easily the mathematical definition? No. We must test this function.[4]

```
from math import isclose

from hypothesis import given
from hypothesis.strategies import lists, floats

import python_numpy
import python_pure

lower_bound = -1e5
upper_bound = 1e5

effectively_zero = 1e-3

@given(lists(
  floats(min_value=lower_bound, max_value=upper_bound).
    filter(lambda x: abs(x) > effectively_zero)))
def test_the_two_implementations_give_the_same_result(data):
  assert isclose(python_numpy.mean(data), python_pure.mean(data))
```

Listing 3

Yes, we will test things

So what tests can we do in the property-based rather than example-based philosophy? Well we have two implementations of the same functionality so maybe we can test that they both produce the same answer. The property being tested here is that all correct implementations give the same answer. Using [Hypothesis] and [PyTest], I came up with the test code shown in Listing 3.

Russel Winder Ex-theoretical physicist, ex-UNIX system programmer, ex-academic. Now an independent consultant, analyst, author, expert witness and trainer. Also doing startups. Interested in all things parallel and concurrent. And build. Actively involved with Groovy, GPars, GroovyFX, SCons, and Gant. Also Gradle, Ceylon, Kotlin, D and bit of Rust. And lots of Python especially Python-CSP.

1. And the NumPy package, obviously.
2. 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.
3. In case you weren't aware, Python is a virtual machine based language; source code is compiled to bytecodes which are executed at run time.
4. Whether this is the correct answer to the question is left as an exercise for the reader.

There is clearly a lot going on in this code. There is a single test function, but we use the Hypothesis **given** decorator and **lists** and **floats** strategies to automatically generate (more or less randomly) lists of floating point values that the test is run on. Most of the work is in conditioning the **float** values that we allow in the test: the float values are constrained to the ranges [-1e5, -1e-3] and [1e-3, 1e5], via the combination of **min_value** and **max_value** constraints in the **floats** strategy combined with the **filter** strategy that applies a predicate to remove values in the range [-1e-3, 1e-3]. Values close to zero are not allowed since hardware floating point values close to zero generally cause serious problems with any and all expression evaluation and hence testing.[5] Similarly, admitting very large floating point values allows Hypothesis to easily discover values that hardware floating point expression evaluation cannot cope with – again resulting in problems of testing nothing to do with the actual properties under test.[6] So, for the purposes of testing, we stick with floating point values of about eight significant digits to try and ensure we do not get rounding errors in the hardware evaluation that falsify our properties due to the behaviour of floating point hardware rather than a failing of the property.

The test itself eschews asserting equality of two hardware floating point values, as this is clearly not a thing any right thinking programmer would ever dream would work.[7] Instead the **math.isclose** predicate is used to determine if the values are close enough to each other to be deemed equal.

```
from math import isclose

from numpy import isnan

from hypothesis import given
from hypothesis.strategies import lists, floats

from pytest import raises

import python_numpy
import python_pure

lower_bound = -1e5
upper_bound = 1e5

effectively_zero = 1e-3

def test_numpy_mean_return_nan_on_no_data():
  assert isnan(python_numpy.mean([]))

def test_our_mean_raises_exception_on_no_data():
  with raises(ZeroDivisionError):
    python_pure.mean([])

@given(lists(
  floats(min_value=lower_bound, max_value=upper_bound).
    filter(lambda x: abs(x) > effectively_zero),
  min_size=1))
def test_the_two_implementations_give_the_same_result(data):
  assert isclose(python_numpy.mean(data), python_pure.mean(data))
```

Listing 4

Anyone 'on the ball' will already have realised that there was going to be an error executing this test, even with the carefully constructed test. When pytest is run on the test code we get:

```
dataset = []
  def mean(dataset):
>   return sum(dataset) / len(dataset)
E   ZeroDivisionError: division by zero
```

One up to Hypothesis for finding that the empty dataset breaks our implementation.

If we took a 'Just fix the tests so they are green' approach[8] we might just change the tests to that as seen in Listing 4. The empty dataset case is separated out and the **@given** decorator is extended to required that only lists with at least one item are generated.

But is this the right behaviour?

The question we have to ask is whether this behaviour of the implementation of Listing 2 with empty lists is the right behaviour. Indeed this should have been the question asked instead of just patching the test in the first place.[9]

We note that **numpy.mean** returns **NaN** for empty data rather than throwing an exception. In the original submission of this article, I amended the **pure_python.mean** to return **NaN**, just to be consistent with **python_numpy.mean**. The reviewers, though, were fairly unanimous that NumPy was doing the wrong thing, that mean of an empty dataset should be undefined, i.e. raise an exception. This viewpoint is encouraged when looking at the **statistics** module in the Python standard library (from Python 3.4 onwards). Its **mean** function definitely raises an exception on no data. So, let's treat the **numpy.mean** behaviour as an aberration: let's change the implementation of our pure Python **mean** function to that shown in Listing 5 and make the appropriate change to the test for empty data, as shown in Listing 6.

But what are the properties?

None of this has though really opened up the question of properties of the computation: we have only example-based tests for the empty dataset, and a comparison test to test the equivalence of different implementations. So what are the properties of the mean calculation that we can test in the form of property-based tests for each implementation separately?

A search of Google[10] should always turn up the property that:

$$\sum_{i=1}^{n}(x_i - \bar{x}) = 0$$

i.e. the sum of the differences of individual items from the mean should be zero. Given a dataset and a putative mean calculation function, we can test that this property is not falsifiable. Further delving into the notion of 'properties of the mean' may well also turn up the following properties:

```
# python_pure_corrected.py

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

Listing 5

5. Allow values close to zero in the sample set and Hypothesis will always find a case that falsifies any property.
6. This property of hardware floating point numbers and Hypothesis (or any random test value generator with a good search algorithm) is well understood and is unavoidable, hence having to take great care conditioning the floating point values selected.
7. Any programmer claiming competence that uses equality between floating point values clearly requires re-education on this point.
8. Anyone taking this approach in real life is definitely doing it wrong, even though we all know this is what actually happens all too often. Clearly we should fight against this approach at all times.
9. Except then I wouldn't have had a chance to chide people taking the 'just fix the tests so they are green' approach.
10. Yes, Google searching can actually turn up useful facts, as well as satire, spoof, and fictional stuff. And pictures of cats, obviously.

$$\frac{1}{n}\sum_{i=1}^{n}(x_i + k) = \bar{x} + k \qquad \forall k > 0$$

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - k) = \bar{x} - k \qquad \forall k > 0$$

$$\frac{1}{n}\sum_{i=1}^{n}(x_i * k) = \bar{x} * k \qquad \forall |k| > 1$$

$$\frac{1}{n}\sum_{i=1}^{n}(x_i / k) = \bar{x} / k \qquad \forall |k| > 1$$

So here we have a number of expressions (that we can treat as predicates, i.e. properties) that can be evaluated, that our putative mean implementation should pass.[11] All of them are exactly the sort of property that Hypothesis can test using its random generation of values from a set.

And the result is...

Putting all this together I present the test as shown in Listing 7.[12,13,14]

We have the example-driven test of the empty list case, as previously.

The test of equivalence of different implementations is replaced by tests of the properties for each implementation separately: `test_sum_of_data_minus_mean_is_zero` realises the first equation, and `test_mean_of_offset_data_is_correct` handles all four 'offset testing' equations. Both of these functions are parameterised using the `pytest.mark.parametrize` decorator so as to create tests for each of the `mean` implementations being tested, three in this case. The `test_mean_offset_data_is_correct` test function is also parameterised over the arithmetic operations +, -, *, and /,[15] implementing the four 'offset testing' equations. The selection of lists of floating point values is very much as previously, but with the creation of the offset value added as a second parameter to the `given` decorator.

All the tests pass.[16]

So, hopefully, this short article has 'filled in the gaps' left by the previous article, about what a property is and what property-based testing is about.

Final thought

Is the mean of a dataset the only function that has the properties tested here? ■

```
from math import isclose

from numpy import isnan

from hypothesis import given
from hypothesis.strategies import lists, floats

from pytest import raises

import python_numpy
import python_pure_corrected

lower_bound = -1e5
upper_bound = 1e5

effectively_zero = 1e-3

def test_numpy_mean_return_nan_on_no_data():
  assert isnan(python_numpy.mean([]))

def test_our_mean_raises_exception_on_no_data():
  with raises(ValueError) as error:
    python_pure_corrected.mean([])
    assert error.value == 'Cannot take mean of no data.'

@given(lists(
  floats(min_value=lower_bound, max_value=upper_bound).
    filter(lambda x: abs(x) > effectively_zero),
  min_size=1))
def test_the_two_implementations_give_the_same_result(data):
    assert isclose(python_numpy.mean(data), python_pure_corrected.mean(data))
```

Listing 6

References

[Hypothesis] http://hypothesis.works/

[PyTest] http://pytest.org/

[Winder16] Testing Propositions, *Overload* 134, pp.17–27, 2016-08.
PDF file of the entire issue: https://accu.org/var/uploads/journals/Overload134.pdf .
Web page of this article: https://accu.org/index.php/journals/2272

11. In the early draft of this article, I failed to note the constraints on *k* in the equations, although I had implemented the constraints in the code. This raised a debate amongst the reviewers that there were only two, not four, equations. Both sides were right because the presentation was, at that time, inconsistent. The question was which consistency to go with. For a small number of minor 'hidden agenda' points, I have gone with four equations, but going with two equations and not restricting the domain would be an equally valid way forward.
12. I have added `statistics.mean` in here as well, just for good measure. Formally, fairly useless, or it should be, but fun nonetheless.
13. Actually, we really should note that testing `numpy.mean` and `statistics.mean` here is completely superfluous and totally redundant in terms of testing obligation. It is done here to show that all the implementations do, in fact, pass the tests.
14. Thanks to D R MacIver, the author of Hypothesis, for being around on Twitter on 2017-01-01 to fix a serious architectural failing with the draft of this code. In summary: each test should have one and only one given decorator.
15. Python provides functional forms of the +, -, and * operators, but not /, hence the lambda function for division.
16. At the time of submission of this article, running on Debian Sid with Python 3.5.2, PyTest 3.0.5, and Hypothesis 3.6.0, anyway.

```
from math import fabs, isclose
import operator
import statistics

from numpy import isnan
from hypothesis import given, settings
from hypothesis.strategies import lists, floats
from pytest import mark, raises
import python_numpy
import python_pure_corrected

implementations = (python_numpy.mean, python_pure_corrected.mean, statistics.mean)
operators = (operator.add, operator.sub, operator.mul, lambda a, b: a / b)

lower_bound = -1e5
upper_bound = 1e5
effectively_zero = 1e-3

def test_numpy_mean_return_nan_on_no_data():
  assert isnan(python_numpy.mean([]))
def test_our_mean_raises_exception_on_no_data():
  with raises(ValueError) as error:
    python_pure_corrected.mean([])
    assert error.value == 'Cannot take mean of no data.'
def test_statistics_mean_raises_exception_on_no_data():
  with raises(statistics.StatisticsError):
    statistics.mean([])

@mark.parametrize('implementation', implementations)
@given(lists(
  floats(min_value=lower_bound, max_value=upper_bound)
    .filter(lambda x: fabs(x) > effectively_zero),
  min_size=1, max_size=100))
def test_sum_of_data_minus_mean_is_zero(implementation, data):
  x_bar = implementation(data)
  assert sum(x - x_bar for x in data) < effectively_zero

@mark.parametrize('implementation', implementations)
@mark.parametrize('op', operators)
@given(lists(
  floats(min_value=lower_bound, max_value=upper_bound).
    filter(lambda x: fabs(x) > effectively_zero).
    map(lambda x: round(x, 3)),
  min_size=1),
  floats(min_value=2.0, max_value=upper_bound),
)
def test_mean_of_changed_data_obeys_property(implementation, op, data, offset):
  x_bar = implementation(data)
  x_bar_offset = implementation([op(x, offset) for x in data])
  assert isclose(x_bar_offset, op(x_bar, offset))
```

Listing 7

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

The Importance of Back-of-Envelope Estimates

Guestimate questions make many people grumble. Sergey Ignatchenko reminds us why they matter.

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.

With all the techniques we use during development (ranging from 'just keyboard and vim' to 'RAD IDE which claims to make writing code unnecessary'), there is one thing which is unfortunately meticulously neglected across the software development industry (and moreover, there are arguments pushed forward that it is a Bad Thing even to *try* using it). I'm speaking about order-of-magnitude estimates made on the back of an envelope. While it is impossible to provide any strict proof that such estimates are useful, we'll see some anecdotal kinda-evidence that such estimates do help us to avoid spending months (and even years) on development which is apparently unnecessary or outright infeasible.

BTW, the subject of back-of-the-envelope estimates in IT is not something new (it was discussed at least in [McConnell] and [Atwood], and recently mentioned in [Acton]); indeed, I myself am guilty of using it in [NoBugs12] and [NoBugs13]☺. However, with such estimates being so important and so frequently neglected, I am sure it is necessary to emphasize their importance once again (and again, and again ;-)).

Definition and examples from physics

First of all, let's define what we're talking about. Without going into lengthy discussion, let's just quote Wikipedia:

> In the hard sciences, *back-of-the-envelope calculation* is often associated with physicist Enrico Fermi, who was well known for emphasizing ways that complex scientific equations could be approximated within an order of magnitude using simple calculations.

Besides noting that with Enrico Fermi being one of the brightest minds of the XX century, we're certainly in a good company ;-), we should emphasize this 'within an order of magnitude' qualifier. In other words, we're not even trying to get results which can be seen as a replacement for benchmarking. On the other hand, if our simplified calculations can give us an order of magnitude that is appropriate for approximating the exact value we need, we don't have to spend time performing an actual experiment.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko and Dmytro Ivanchykhin using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

Two most famous examples of back-of-the-envelope calculations in physics are the following (with lots and lots of others not recorded as they're not seen as anything special):

- Fermi estimating the yield of an atomic bomb by dropping bits of paper and measuring the distance they were blown by the blast wave (he estimated 10K tons of TNT, the actual result was 18.6K tons). See also his famous 'Fermi problem' of 'how many piano tuners are there in Chicago?' ☺.
- Arnold Wilkins spending a few hours proving that radio-based 'death rays' (claimed to be invented by several influential people, including Guglielmo Marconi and Nicola Tesla) are outright impossible, but using radio waves to detect moving objects is perfectly feasible. This, in turn, led to the development of the radar.

Estimating CPU and RAM usage

Well, as most of us are not physicists, and are more like software developers and/or admins, let's take a look at some real-world examples from IT. While (of course), any such examples at best count as anecdotal evidence, they still illustrate the tremendous practical value which can come out of them.

My first family of examples are about estimating CPU and RAM usage. Actually, examples of such calculations are so routine that they come and go without being noticed. Still, I managed to remember four recent ones which I encountered within the last two months.

The first example was about CPU. In an otherwise very good and insightful presentation (which I won't name exactly since it was very good and insightful), an example was provided where a certain suboptimality had led to memory reads of 1Megabyte per 10,000 frames of the video game – instead of the 1 Kbyte which was really necessary. As it was noted (and perfectly correctly too), this corresponds to 99.9% of CPU memory bandwidth being wasted. What wasn't mentioned, however, is that:

- 1Megabyte of waste per 10,000 frames corresponds to a mere 100 bytes/frame
- at a typical 60 frames/second, this corresponds to a mere 6K bytes/second of memory bandwidth being wasted
- with modern x64 CPU bandwidth being of the order of 20+ GByte/second, we're talking about the waste of about 3e-7 of the total memory bandwidth.

BTW, let's note that I am not arguing with the main point of the presentation – that you can easily waste a lot of memory bandwidth; it is just in this specific example, trying to optimize out a 3e-7 performance hit is very rarely worth the trouble.

The second example was also about CPU. In a high-performance event-driven machine handling TCP/UDP packets coming over the Ethernet, the question has arisen whether it is ok to use virtual functions to implement event dispatch. A very quick calculation on the back of the envelope has shown that:

if our simplified calculations can give us an order of magnitude that is appropriate for approximating the exact value we need, we don't have to spend time performing an actual experiment

- The minimal size of an Ethernet packet which such a system can possibly receive is Ethernet_Frame_Gap + Ethernet_Preamble + Minimal_Ethernet_Packet_Size, or 12+8+64 bytes = 84 bytes.
- For a 100MBit/s link (which is the limit the system has for other reasons), in the very, very best case, it can get 100Mbit/8 ~= 1.2e7 bytes/second, or (given the 84 bytes/packet) 150K packets/second.
- As it is noted in [NoBugs16], the cost of a virtual function call is usually within 50 CPU cycles
- On the other hand, at 3GHz and with 150K packets/second, we're talking about 20,000 CPU cycles available for handling each packet.
- The waste of at most 50 CPU cycles – compared to 20K CPU cycles – is about 0.2%. Compared to the trouble of avoiding virtual dispatch, the saving seems too small to bother with.

On the other hand, the picture would be *very* different if we were talking about virtualizing some operation which is done 100 times per packet (or about handling the 10GBit/s link, but the latter is quite difficult regardless of the virtual function calls). The whole point is about the numbers in specific scenarios – and certainly *not* about blanket statements such as 'virtual functions costs are {negligible|very high} regardless of the context'.

The third example of back-of-the-envelope estimates which I encountered in recent months was about the cost of exceptions vs the cost of the return-and-check of the error code. The cost of modern implementations of exception handling in C++ are 'zero-cost when no exception happens' – but they cost thousands of CPU cycles when they do happen. On the other hand, an old-fashioned return of the error code from the function being checked by the caller will cost some single-digit CPU cycles on each function call even if nothing wrong happens. When we combine these two numbers together, we realize that performance-wise exceptions will beat return-and-checks as soon as there is 10K function calls per one exception; on the other hand, having one exception per 100 function calls is probably detrimental to performance. Anything between 100 and 10K function calls is too close to call – but on the other hand, the performance difference probably won't be too drastic regardless of the approach chosen. NB: I am intentionally ignoring the other advantages of exceptions at the moment; for the purposes of this article, the whole discussion is about performance, and only about performance.

The fourth example (and final among those which I've run into during the last month or two) was about RAM. In the context of app-level caching of USERS from a database, I was asked "Hey, if we try to cache a million users, will they fit in memory?" My very simple analysis went along the following lines:

- If we are talking about a 'user' which is just a current state of the user (and not the history of those things which user has ever done), we're usually talking about user id, name, hashed password, e-mail address, maybe physical address, and some attributes such as current credit. If we add all these items together, we'll be talking about some hundreds of bytes. Accounting for all the inefficiencies and overheads, let's say the upper bound is around 1K bytes/user.
- With this in mind, a million users will take around 1 GByte.
- As this whole discussion was in the context of servers, and as modern 'workhorse' 2S/1U servers are able to host up to 512G RAM easily (and modern 4S/4U servers able to host 4+ terabytes[1]), it means that even 100M users aren't likely to cause too much trouble. In addition, noting that this is not really the number of all users in the DB, but just those users which need to be cached (which roughly corresponds to 'users currently on site') – well, it means that there aren't that many systems out there (if any) which would have any trouble caching users in the app-level cache.

Overall, as we can see, there are quite a few situations when back-of-the-envelope (a.k.a. order of magnitude) analysis can provide very valuable information even before we start implementing our system. Sure, such calculations are not a substitute for benchmarking, but they can save us lots and lots of trouble either (a) trying to implement things which won't be able to cope with the load, or (b) trying to do premature optimizations which aim to eliminate problems which will never really bite us.

Estimating MTBF

While estimating CPU and RAM usage is very important, they're certainly *not* the only cases when calculations on the back of the envelope come in handy.

Quite recently, I was involved in quite an interesting analysis related to making systems redundant. Lots of people out there tend to postulate that 'Hey, to be reliable your system needs to be redundant, period!'; however, under a more thorough analysis this claim fails pretty badly in quite a significant number of real-world scenarios.

First of all, let's note that when talking about reliability, it is only MTBF (= Mean Time Between Failures) which matters; in other words,

> *if comparing two systems – one being redundant but failing every month, and another being non-redundant but failing every year – I will clearly pick the second one every day of the week.*[2]

Ok, sure we should note that there are other factors which may need to be considered within the context of reliability (in particular, the way in which the system fails; there is quite a significant difference between losing all the business data, having a 'split-brain', or just needing a restart). On the other hand, all these things we're really interested in are inherently observable ones; as redundancy is not an observable property, it is merely an implementation detail to implement those observable MTBFs etc.

Now, let's consider an OLTP database running on a larger 4S/4U box; as noted in [NoBugs16a]. Such a box (with proper optimizations) can be able to run up to dozens of billions transactions/year, and this is quite a substantial number for quite a few systems out there.

With 4S/4U boxes having typical MTBFs of 3–5 years, the next question we should ask ourselves, is "Hey, will we really be able to write software

1. 4S = "4-Socket"; 4U = "4 rack Units". Popular examples include HP DL580 (my personal favourite for years ;-)), and Dell R930.
2. And this happens in real-world too, see also below

spending time making the system redundant is not efficient, and that spending the same time on things such as code reviews can lead to better overall reliability

which crashes much more rarely than that?" If not (and for any business software, the chance of failures being much more frequent than once every 3–5 years is overwhelmingly high), then the whole point of being redundant pretty much goes away; in other words, it can easily be the case that spending time making the system redundant is not efficient, and that spending the same time on things such as code reviews can lead to better overall reliability.

On MTBFs of redundant systems

As a side note – actually, when calculating MTBFs of a redundant system with 2 nodes, the numbers are not as good as they might seem on first glance. Let's consider a redundant system Z consisting of 2 redundant components X and Y. Now we need to introduce MDT (= Mean Down Time), which is the mean time between the node going down; MDT is usually measured in hours and usually ranges from 8 hours to several days. (Note that MTTR (= Mean Time To Repair), while closely related to MDT, doesn't normally include such things as 'How to get the replacement part to your datacenter' – and so is not directly usable for our MTBF analysis.)

Let's note that the maths below, while perfectly common and accepted (see, for example, Chapter 8 in [Smith]) is using quite a few implicit assumptions, which are beyond the scope of our exercise. In particular, it assumes that (a) MDTs are negligible compared to MTBFs, and (b) that failure probabilities (inverse of MTBFs) can be added together (i.e. that failure probabilities are small enough to say that non-linear effects when adding probabilities, are negligible).

Note that all these assumptions, while potentially confusing, do stand in most real-world situations. What we'll be concentrating on is a different implicit assumption – the one which doesn't usually stand ☹.

```
//WARNING: INVALID IMPLICIT ASSUMPTION AHEAD
```

At this point, it is common to say (erroneously! See below) that redundant system Z will fail if and only if one of the following scenarios happen: (a) after component X fails, component Y will fail within MDT*x* (i.e. while component X is still being repaired); or (b) after component Y fails, component X will fail within MDT*y* (i.e. while component Y is still being repaired). The probability of such a failure of component Y within the MDT*x*, assuming that MTBFs are large, and MDTs are relatively small compared to MTBFs, is

$$P_{yx} = 1/MTBF_y * MDT_x$$

NB: relying on assumption (a) above

It means that $MTBF_z$ can be calculated as

$$\begin{aligned} MTBF_z^{incorrect} &= 1 / (1 / MTBF_x * P_{yx} + 1 / MTBF_y * P_{yx}) \\ &= 1 / (1 / MTBF_x * 1/MTBF_y * MDT_x \\ &\quad + 1 / MTBF_y * 1/MTBF_x * MDT_y) \\ &= MTBF_x * MTBF_y / (MDT_x + MDT_y) \end{aligned}$$

NB: relying on assumption (b) above

```
//END OF INVALID IMPLICIT ASSUMPTION
```

It looks all grand and dandy (and with typical MTBFs of 3–5 years and MDTs being maximum 3–5 days, we'd have an $MTBF_z^{incorrect}$ of thousands of years – wow!) – until we notice that there is one thing which is utterly unaccounted for in the formula above: it is the MTBF of the redundancy system itself. Let's name it $MTBF_r$.

Practically, $MTBF_r$ needs to cover all the components which form the redundancy system itself. Just one example: if our system uses a 'heartbeat' Ethernet cable between two nodes to detect failure of the other node, then failure of this cable is likely to lead to all kinds of trouble (including extremely disastrous 'split-brain' failures), and so it needs to be accounted for in $MTBF_r$. In a similar manner, network cards (and their respective drivers(!)) serving this 'heartbeat' cable, also need to be included into $MTBF_r$. Moreover, if this cable and NICs are made redundant (which would be quite unusual, but is certainly doable), they will still have their respective $MTBF_r$, and moreover there will be some kind of software (or, Linus forbid, drivers) handling this redundancy, which will also have its own $MTBF_r$. And so on, and so forth.

With $MTBF_r$ in mind (and realizing that whenever redundancy system itself fails – the whole thing will fail too) – $MTBF_z^{correct}$ can be written as

$$MTBF_z^{correct} = 1 / (1/ MTBF_z^{incorrect} + 1/ MTBF_r). (*)$$

How large your $MTBF_r$ is depends, but I can assure you that for the vast majority of real-world cases, it will be *much* smaller than those hyper-optimistic 'thousands of years'.

> *Moreover, according to the formula (*) above, if $MTBF_r$ is smaller than $MTBF_x$ and $MTBF_y$, adding redundancy makes things worse than it was without any redundancy.*

And in practice (and especially whenever redundancy is implemented in software), $MTBF_r$ can be easily much smaller than $MTBF_x$. For example, if $MTBF_r$ is 1 month (BTW, it is not a joke, I've seen quite a few redundancy systems which exhibited less-than-a-week MTBFs under serious loads) while having $MTBF_x$ at 3–5 years – the formula (*) will show that $MTBF_z^{correct}$ is about 0.9999 of $MTBF_r$ (i.e. *much* smaller than original non-redundant $MTBF_x$).

Redundancy estimates – sanity checks

As a nice side effect, our formula of $MTBF_z^{correct}$ also explains an apparent discrepancy of theoretically predicted MTBFs (those calculated in a manner similar to the calculation of $MTBF_z^{incorrect}$), and realistic numbers – especially if redundancy is implemented in software. Just two real-world stories in this regard.

Once upon a time (around 1997 or so), Stratus (one of the leaders of really serious redundant systems aimed at military, nuclear stations etc. – and generally performing as promised) decided to produce a software-based RADIO cluster. Essentially, the RADIO cluster was just two Windows boxes with a fault-tolerant logic implemented in software; using $MTBF_z^{incorrect}$-like calculations, its MTBF was in hundreds of years. It was a rather impressive system (well, *all* the Stratuses are quite impressive) – that is, until RADIO crashed with a dreaded Blue Screen Of Death (coming from redundancy driver) just half an hour into testing ☹;

and it wasn't a fluke: under any kind of decent load, RADIO kept crashing several times a day. So much for very-nicely-looking MTBFs calculated along the lines of $MTBF_z^{incorrect}$. RADIO was discontinued really soon, and to the best of my knowledge, Stratus has given up on software-based redundancy at least for a long while. BTW, for hardware-based Stratuses MTBF is indeed in hundreds of years.

In a completely separate story, at some point a rather seriously loaded system was undergoing pre-IPO technical audit. Out of the whole audit, two points are of interest to our discussion:

- At some point, the auditor asked about system downtime, and after hearing the answer, he was really surprised; the wording went along the lines of 'how do you guys manage to achieve unplanned downtimes which are 5x–10x lower than others in the industry?!' Ok, the full log files were provided, essentially proving that the downtimes-being-*much*-lower-than-industry-average were real.
- At another point (coming much later in the discussion), the auditor noticed that main DB server of the system runs without redundancy. From this point on, the dialog went along the following lines:
  - Auditor: Why don't you use clusters?
  - Team: Why should we?
  - Auditor: Because it isn't reliable without redundancy.
  - Team: Actually, the reason why we have unplanned downtimes which are that much lower than industry average is because we do NOT use clusters – and the rest of the industry does.
  - The curtain falls.

At that point, the team wasn't able to articulate the formulae and back-of-the-envelope estimates discussed above to explain the whole thing; however, as of now, feel free to use it as an argument with all kinds of the auditors (and management) insisting on redundancy of all the mission-critical boxes. Note, though, that this logic stands only if *both* of the following two observations stand for your system:

- The cost of failure can be calculated; in other words, it is not about loss of life, and even not about loss of the whole business.
- the number of boxes involved is very low (like in 'just one, maybe two'). As the formulae above will show, the cost of redundancy (even if it has an $MTBF_r$ as poor as 1 failure per month) is low enough compared to the chance of any one of a thousand of servers failing.

Estimating network traffic and RTT

Last but not least, back-of-the-envelope estimates often come handy in the context of network programming.

One rather typical case for traffic calculations occurs when we want to push small pieces of information to the clients all the time; this happens all the time in the context of a game (and IMNSHO, games include stock exchanges, etc.).

Let's consider an example where we need to calculate traffic for an RPG game server which handles 10K players, with each of the players receiving updates about 10 other players and 50 objects (those which are nearby, so the player can see them), with each of players and objects described with three coordinates and a quaternion (the latter to describe rotation), and the updates are sent 60 times per second (at the frame rate of the game client). In this case, if we're encoding each coordinate and each angle as an 8-byte double, we'll find that:

- Each update will take ((10+50)*7*8) bytes of payload, plus at least 14+28 bytes of Ethernet + IP + UDP headers, totalling to ~3K per update
- Each player will need to receive 3Kbytes / update * 60 updates/second = 180 kBytes/second
- With 10K players/server, it means that per-server traffic will be like 1,8 GByte/second, or 14.5 GBit/second
- Even with typical pricing these days being of the order of $3K / month for 10 Gbit/s, it is going to cost quite a lot.

On the other hand, if we:

- reduce the update frequency (for most RPG out there, 20 updates / second will be perfectly unnoticeable, and 10 updates / second will still probably be fine)
- switch to rounded fixed-point representation for coordinates (which can be as little as 10–12 bits per coordinate; 10 bits is enough to represent 10 cm precision in a 100m radius around our character). It will mean that each coordinate will use 10 bits instead of former 64(!).
- switch to Euler-angle-based fixed-point representation for rotations (with precision of each angle being 10 bits, or 0.3 degree)

we can easily reduce our traffic to:

- ((10+50)*6*1.25)+42 ~= 500 bytes / update
- 500 bytes / update * 10 updates / second = 5 kBytes / second per player
- 5kBytes/second/player * 10K players/server = 50 Mbytes / second, or 400 Mbit/second

While it is still quite a substantial number, it is roughly 36x smaller than before, so at least our traffic costs went down quite a bit ☺. Moreover, while this is very difficult to account for in advance, rounded fixed-point numbers are usually compressed significantly better than full-scaled doubles, which will usually allow further reduction of the traffic.

Another case for back-of-the-envelope estimates in the context of network programming is related to round-trip times (routinely abbreviated as RTT). An example which I want to describe in this regard is a rather long real-world story.

Once upon a time, there was a Really Big Company. And developers there were tasked with developing a system so that companies around the globe (in particular, from the US and Europe) can collaborate. And the guys were tasked with doing it over CORBA as a Big Fat Business Requirement (please don't tell me that CORBA is actually an implementation detail so it doesn't belong in Business Requirements; of course it is, but this kind of stuff does happen, *especially* in Really Big Companies).

And CORBA of that time (which didn't support the concept of passing objects by value rather than by reference) had an ideology that you create a remote object, and then you call a remote method to add an item to a certain list in the remote object, and then you call a remote method to set a field in the item of the remote object you just created, and so on. And for several of the forms within the program – well, the number of fields combined over various items has reached thousands(!); with CORBA, this meant that when a user presses the 'submit' button, there will be thousands of those remote calls.

The guys from the Really Big Company have built the system along the lines above, and they even tested it in LAN. However, when they tried to deploy it over the Atlantic (which was their primary use case), submitting certain forms started to take up to 20 minutes(!), which was obviously unacceptable. Sure, the system was fixed (by removing those roundtrips, which required rewriting like a half of the system and took half a year or so).

What matters, from our current perspective, is that if the guys had performed a back-of-the-envelope estimate in advance, they would see that:

- 5,000 fields for all the items in a form, combined with CORBA's '1 remote call per field' approach, means 5,000 remote calls
- Each remote call incurs at least one round-trip
- Over the trans-Atlantic cable, each remote call takes at the very least 80 ms (the absolute theoretical limit imposed by the speed of light in fibre is 56ms for NY-to-London); taking into account that the program was intended to be used worldwide, we should expect a round-trip time of 200ms at the very least.

- Hence, in the worst-case, 5,000 remote calls would lead to 5,000 round-trips of 200ms each, which translates into a 1000-second delay, or over 15 minutes.

Sure, in practice it was even worse than that – but even 15 minutes would be bad enough ☹ to see that the whole model is not really feasible. Realizing this in advance would save those guys (and that Really Big Company) quite a bit of wasted effort – and quite a bit of embarrassment too.

The importance of sanity checks

I am terribly sorry, but here your calculations are wrong by EIGHT orders of magnitude, so I cannot give you a credit for this lab.

~ overheard during my uni physics courses

One all-important precaution which needs to be done whenever you're trying to use back-of-the-envelope estimates (or actually, any kind of calculations, but this is a subject for a separate discussion) is making sanity checks. When you have your estimate, you at least need to try to think whether it makes sense given your experience; however, if you're going to use the estimate to make any serious decisions, you *should* try to get some other way to double-check your estimate. BTW, in quite a few real-world scenarios, a back-of-the-envelope estimate can be double-checked by another – *independent(!)* – back-of-the-envelope estimate.

Benchmarks vs back-of-the-envelope estimates

When trying to convince various people to do some back-of-the-envelope estimates, I've often run into arguments such as 'Hey, don't even try this, the only way to get anything meaningful is to try it and benchmark it properly in the context of your specific task'.

I'm not going to argue with this statement, it is indeed a good advice – that is, *if* trying and benchmarking is feasible. However, especially at the earlier stages in development, trying certain things can be extremely expensive; this is when back-of-the-envelope estimates come in really handy.

> *Back-of-the-Envelope Estimates are not a replacement for benchmarks; instead, they're prerequisites to implementing systems which can be used for benchmarking.*

Conclusions

Summarizing all the anecdotal kinda-evidence above, I am comfortable to state the following.

On the one hand, back-of-the-envelope calculations are all-important at least at architectural stages of the project. In particular, they often enable avoiding implementing things which cannot possibly fly for various reasons – and on the other hand, allow avoidance of premature optimizations which will never be important enough for the task in hand.

On the other hand, back-of-the-envelope calculations are just one of the instruments available, and, unfortunately, they're not the most reliable one ☹.

Make sure to double-check back-of-the-envelope estimates, and to test them, and to benchmark things – as long as it is feasible, that is. ■

References

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

[Atwood] Jeff Atwood, How Good an Estimator Are You?, https://blog.codinghorror.com/how-good-an-estimator-are-you/

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

[McConnell] Steve McConnell. *Software Estimation: Demystifying the Black Art (Developer Best Practices)*. https://www.amazon.com/exec/obidos/ASIN/0735605351/codihorr-20

[NoBugs12] 'No Bugs' Hare, 640K 2^{256} Bytes of Memory is More than Anyone Would Ever Need Get, *Overload* #112, 2012

[NoBugs13] 'No Bugs' Hare, Hard Upper Limit on Memory Latency, *Overload* #116, 2013

[NoBugs16] 'No Bugs' Hare, Infographics: Operation Costs in CPU Clock Cycles, http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

[NoBugs16a] 'No Bugs' Hare, Gradual OLTP DB Development - from Zero to 10 Billion Transactions per Year and Beyond, http://ithare.com/gradual-oltp-db-development-from-zero-to-10-billion-transactions-per-year-and-beyond/

[Smith] *Reliability, Maintainability and Risk*. Dr David J. Smith. ISBN 978-0080969022

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Multiprocessing and Clusters in Python

Multiprocessing is possible in Python. Silas S. Brown shows us various ways.

It's surprisingly easy to use more than one CPU core in Python. You can't do it with straightforward threads, since the C implementation of Python has a Global Interpreter Lock (GIL) which means there can only ever be one thread performing active calculations at any one time, so threads in Python are generally useful only for waiting on I/O, handling GUIs and servers and such, not actually processing in parallel when you have multiple CPU cores. (The Java implementation has no GIL and really can run on multiple cores in parallel, but I'm assuming you have an existing Python project and want to stick with the C implementation.) But there are now ways of multiprocessing in standard C Python, and they're not too difficult, even to add in to legacy Python code.

Python 3.2 introduced the concurrent.futures module as standard [Python], and there's a backport for Python 2.7 which can usually be installed on Unix or GNU/Linux via `sudo pip install futures` (in Debian or Ubuntu you might need `sudo apt-get install python-pip` first; on a Mac try `sudo easy_install pip`). One nice thing about this module is it's quite straightforward to roll your own 'dummy version' for when parallelism is not available: see Listing 1.

This gives you an object called `executor` which supports `submit(function, arguments)` returning an object that will, when asked for its `result()` later, give you either the result of the calculation or the exception raised by it, as appropriate. (Java programmers should recognise these semantics.) The `executor` object also has a `map(function, iterables)` which works like the built-in `map()`. If you're on a multi-core machine and the real `concurrent.futures` is available in its Python installation, then some of the work will be done asynchronously on other CPU cores in between the calls to `submit()` and `result()`, so you can parallelise programs simply by looking for situations where independent calculations can be started ahead of when their results are needed, or even just by parallelising a few calls to `map()` as long as your map functions are not so trivial that the overhead of parallelising them would outweigh any benefit. But if your script is run on an older machine with no `concurrent.futures` available, it will fall back to the 'dummy' code which simply runs the function sequentially when its result is called for. (And if that result turns out not to be required after all and is not asked for, then the function won't run. So if parallelism is not available then at least you can benefit from lazy evaluation. But this applies only if your algorithm involves speculative computations i.e. ones you start before knowing if you'll really need them.)

```
try:
  import concurrent.futures
  executor = \
    concurrent.futures.ProcessPoolExecutor()
except:
  class DummyExecutor:
    def submit(self, fn, *args, **kwargs):
      class Future:
        def result(self,*_):
          return fn(*args,**kwargs)
      return Future()
    def map(self, func, *iterables, **kwargs):
      for n in map(func,*iterables): yield n
  executor = DummyExecutor()
```

Listing 1

I like the idea of 'if it's there, use it; if not, do without': it means users of my scripts don't *have* to make sure `concurrent.futures` is available in their Python installation. If they don't have whatever it takes to install it, they'll simply get the sequential version of my script rather than an `ImportError` (`ImportError`s in your scripts can be bad PR). Note I'm not specifically catching `ImportError` around the `concurrent.futures` import, because it's also possible for this import to succeed but still fail to make `ProcessPoolExecutor` available. This can be seen by reading `__init__.py` in the source code of `concurrent.futures`: if `ProcessPoolExecutor` cannot be loaded, then the module will just give you `ThreadPoolExecutor`. But there's no point using `ThreadPoolExecutor` for multiprocessing, because `ThreadPoolExecutor` is subject to the GIL, so we want to verify that `ProcessPoolExecutor` is available before going ahead.

The interface to the 'dummy' object is actually quite a far cry from that of the real thing. With the real concurrent.futures, you can't pass lambda or locally-defined functions to `submit()` or `map()`, but the dummy object lets you get away with doing this. Also, the real `concurrent.futures` has extra functionality, such as `add_done_callback` and polling for completion status, and does not run a function twice if you call its `result()` twice. All of this can be worked around by writing a more complex dummy object, but if all you're going to do anyway is call `submit()` and `result()` then there's not a lot of point making the fallback that complicated: if a few lines of script are supposed to be a 'poor man's' fallback for a large library, then we don't want to make the substitute so big and complicated that we almost might as well bundle the library itself into our script. Just make sure to test your code at least once with the real `concurrent.futures` to make sure you haven't accidentally tried to give it a lambda function or something (the dummy object won't pick up on this error). You can of course insert print statements into the code to tell you which branch it's using, to make sure you're testing the right one; you may even want to leave something in there for the production version (i.e. 'this script should run faster if you install futures').

Oversized data

From this point on, I'll assume the real `concurrent.futures` *is present on the system and you are doing real multiprocessing.*

You don't have to worry about causing too many context switches if too many tasks are launched at once, since `ProcessPoolExecutor`

Silas S. Brown is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

defaults to queuing up tasks when all CPU cores are already occupied with them. But you might sometimes be worried about what kind of data you are passing in to each task, since serialisation overheads could be a serious slow-down if it has to be large.

If you're on Unix, Python's underlying 'multiprocessing' module will start the new processes with `fork()`, which means they each get a copy of the parent process's memory (with copy-on-write semantics if supported by the kernel, so that no copying occurs until a memory-page is actually changed). That means your functions can read module-level global variables that have been set up at runtime before the parallel work started (just don't try to change these during the parallel work, unless you want to cope with such changes affecting some future calculations but not others depending on which CPU or process ID happens to run what). `fork()` does, however, mean you'd better be careful if you're also using threads in the same program, such as for a GUI; there are ways of working around this, but I'd suggest concentrating on making a command-line tool and let somebody else wrap it in a GUI in a different process if they must.

But you can't rely on having `fork()` if your script might be run on Windows, nor if you might eventually use multiple machines in a cluster using `mpi4py.futures` (more on this below), SCOOP [SCOOP], or a similar tool that gives you the same API as `concurrent.futures`. In these cases, it's likely that your script will be separately imported on each core, so it had better not run unless `__name__ == "__main__"`. You can set up a few module-level variables when that happens; the subprocesses should still have the same `sys.argv` and `os.environ` if that's any help. However, you probably won't want to repeat a long precalculation when doing this.

Since most multiprocessing environments, even across multiple machines in a cluster, assume a shared filesystem, one fairly portable way of sharing such large precalculated data is to do it via the filesystem, as in Listing 2. To avoid the obvious race condition, this must be done before initialising the parallelism.

Listing 2 can detect the case where `fork()` has been used and the data does not need to be read back from the filesystem, although without further low-level inspection it won't be able to detect when it can avoid writing it to the filesystem at all (but that might not be an issue if you want to write it anyway). There are other ways of passing data to non-`fork()`ed subprocesses without using the filesystem, but they involve going at a lower level than concurrent.futures (you can't get away with simply passing the data into a special 'initialiser' function to be run on each core, since the `concurrent.futures` API by itself offers no guarantee that all cores in use will be reached with it).

MPI

Message Passing Interface (MPI) is a standard traditionally used on high-performance computing (HPC) clusters, and you can access it from Python using a number of libraries for interacting with one of the underlying C implementations of MPI (typically MPICH or OpenMPI). Now that we have `concurrent.futures`, it's a good idea to look for libraries supporting that API so we won't have to write anything MPI-specific (if it's there, we can use it; if not, we can use something else). mpi4py [MPI] plans to add an `mpi4py.futures` module in its version 2.1, but, at the time this article was written, version 2.1 was not yet a stable release (and standard `pip` commands were fetching version 2.0), so if you want to experiment with `mpi4py.futures`, you'll have to download the in-development version of mpi4py.

```
from cPickle import Pickler, Unpickler
if __name__ == "__main__":
  data = our_precalculation()
  Pickler(open('precalc','wb'),-1).dump(data)
else:
  try: data
  except NameError:
    data = Unpickler(open('precalc','rb')).load()
```

Listing 2

Addendum for OpenMPI

In OpenMPI, Listing 2 won't work because

```
__name__ == "__main__"
```

in all processes. The OpenMPI equivalent is

```
os.environ['OMPI_COMM_WORLD_RANK'] == '0'
```

Additionally, in OpenMPI all processes will start running even before the `MPIPoolExecutor` is instantiated, so you can't rely on delaying that until after the results of long initial calculations have been written to a file: the subprocesses will either have to poll the file for being ready, or else load it on-demand when they get the first task and cache it from there.

On a typical GNU/Linux box, you can do this as follows: become root (`sudo su`), make the `mpicc` command available (on RedHat-based systems that requires typing something like `module add mpi/mpich-x86_64` after installing MPICH, or equivalent after installing OpenMPI; Debian/Ubuntu systems make it available by default when one of these packages is installed), make sure the python-dev or python-devel package is installed (apt-get install python-dev or yum install python-devel), and then try:

```
pip install https://bitbucket.org/mpi4py/mpi4py/
get/master.tar.gz
```

At this point Listing 1 can be changed (after adding extra indentation to each line) by putting Listing 3 before the beginning. Here, we check if we are being run under MPI, and, if so, we use it; otherwise we drop back to the previous Listing 1 behaviour (use `concurrent.futures` if available, otherwise our 'dummy' object). A subtlety is that `mpi4py.futures` will work only if it is run in a command like this:

```
mpiexec -n 4 python -m mpi4py.futures script.py
args...
```

and that in an MPI environment too (i.e. the above `module add` command will need to have been run in the same shell, if appropriate). Some versions of mpiexec also have options for forwarding standard input and environment variables to processes, but not all do, so you'll probably have to arrange for the script to run without these. Also, any script that uses `sys.stdout.isatty()` to determine whether or not output is being redirected will need to be updated for running under MPI, because MPI always redirects the output from the program's point of view even when it's still being sent to the terminal.

If you want MPI to use other machines in a cluster, then how to do this depends on your MPI version: it may involve extra setup steps before starting your program, as is the case with mpd in older versions of MPICH2 such as version 1.2. But in MPICH2 version 1.5 (the current mpich2 package in Debian Jessie), and in MPICH 3.1 (Jessie's current mpich package), the default process manager is `hydra` and you simply create a text file listing the host names (or IP addresses) of the cluster machines, ensure they can all ssh into each other without password and share the filesystem, and pass this text file to mpiexec using the `-f` parameter or the `HYDRA_HOST_FILE` environment variable. (In OpenMPI you use the `--hostfile` parameter.) Modern MPI implementations are also able to checkpoint and restart processes in the event of failure of one or more machines in the cluster; refer to each implementation's documentation for how to set this up.

If our script is run outside of MPI, then our detecting and handling of 'no MPI' is a little subtle because `mpi4py.futures` (if installed) will still successfully import, and it will even let you instantiate an `MPIPoolExecutor()`, but then will likely crash after you submit a job, and catching that crash from your Python script is very awkward (normal `try`/`except` won't cut it). So we need to look at the command line to check we're being run in the right way for MPI first. But we can't just inspect `sys.argv`, because that will have been rewritten before control is passed to our script, so we have to get the original command line from the `ps` command. The `ps` parameters in Listing 3 were tested on both GNU/Linux and Mac OS X, and if any system does not support them then we should just fall back to the safety of not using MPI.

```
try:
  import os, commands
  commands.getoutput(
    "ps -p " + str(os.getpid()) + " -o args") \
    .index("-m mpi4py.futures") # ValueError
                                # if not found
  import mpi4py.futures
  executor = mpi4py.futures.MPIPoolExecutor()
except:
  # etc (as Listing 1, extra indent)
```

Listing 3

A pattern for moving long-running functions to other CPUs

If you have a function that normally runs quite quickly but can take a long time on certain inputs, it might not pay to have every call run on a different CPU, since in fast cases the overheads of doing so would outweigh the savings. But it might be useful if the program could determine for itself whether or not to run this particular call on a different CPU.

Since, in Python, any function can be turned into a generator by replacing **`return x`** with **`yield x ; return`** (giving a generator that yields a single item), the pattern shown in Listing 4 seems natural as a way to refactor existing sequential code into multiprocessing. The part marked 'first part of function goes here' will be repeated on the other CPU, which seems wasteful but could be faster than passing variables across if they are large; it is assumed that this part of the function does what is necessary for us to be able to figure out if the function is likely to take a long time, e.g. if the first part of the function shows that we are now generating a LOT of intermediate data (which is why we probably don't want to pass it all across once we've decided we're better off running in the background). The **`my_function_wrapped`** part is necessary because **`submit()`** takes only functions not generators.

I'm not suggesting writing new programs like Listing 4, but it might be a useful pattern for refactoring legacy sequential code.

Avoiding CPU overload

The above pattern for moving long-running functions to other CPUs should work as-is on MPI, but with **`concurrent.futures`** it will result in one too many processes, because **`ProcessPoolExecutor`** defaults to running as many parallel processes as there are CPU cores, on the assumption that the control program won't need much CPU itself, an assumption that is likely to break down when using this pattern. The Linux and BSD kernels are of course perfectly capable of multiplexing a load that's greater than the number of available CPU cores, but it might

```
def my_function(param, can_background = True):
  # first part of function goes here
  if (can_background and
   likely_to_take_a_long_time()):
    job = executor.submit(my_function_wrapped,
      param)
    yield "backgrounded"
    yield job.result() ; return
  # rest of function goes here
  # change all 'return x' to 'yield x ; return'

def my_function_wrapped(param):
  return my_function(param, False).next()

def caller():
  gen = my_function(param)
  result = gen.next()
  if result == "backgrounded":
    # Do something else for a while...
    result = gen.next() # get actual result
```

Listing 4

```
import multiprocessing
num_cpus = multiprocessing.cpu_count()
if num_cpus < 2:
  raise Exception("Not enough CPUs")
from concurrent.futures import \
  ProcessPoolExecutor
executor = ProcessPoolExecutor(num_cpus - 1)
```

Listing 5

be more efficient to reduce the number of 'slave' processes by 1 to allow the master to have a CPU to itself. This can be accomplished using code like that in Listing 5.

Evaluation

The above methods were used to partially parallelise Annotator Generator [Brown12] resulting in a 15% overall speed increase when using **`concurrent.futures`** as compared to the unmodified code. This could almost certainly be improved with more parallelisation (recall Amdahl's Law: the speedup is limited by the fraction of the program that must be sequential). Only a fraction of a percent was saved by subtracting 1 from the number of CPUs to achieve a more even load.

Results using MPI were not so satisfactory. When running with 4 processes on a single quad-core machine using MPI, the program was actually slowed down by 8% compared with running single-core, which in turn was 6% slower than the unmodified code. I believe that 6% represents the overhead of converting functions into generators, and could be eliminated by duplicating and modifying the code for the single-core case, but that would introduce a maintenance issue unless it could somehow be automated. Given Annotator Generator's desktop usage scenario, the prevalence of multi-core CPUs on desktops, and the speedup using **`concurrent.futures`**, it doesn't seem very high-priority to invest code complexity in saving that 6% in the single-core case. MPI's poor performance is more worrisome, but I later discovered it was due to the system running low on RAM (and therefore being slowed down by more page faults) while running four separate MPI processes: **`concurrent.futures`** was able to share the data structures, but MPI wasn't (even though it could use shared memory for some message passing). Once I reduced the size of the input, MPI was 14% faster than the single-core case and **`concurrent.futures`** was 18% faster than the single-core case. Perhaps MPI would perform better on a real cluster, which I have not yet had an opportunity to test. A cluster of virtual machines with OpenMPI ran 5% faster than the single-core case, but because these machines were virtual and all running on the same actual machine, I do not believe that result to be meaningful other than as a demonstration that the underlying protocols were working. Still, I suspect a greater deal of parallelisation is required to outweigh the overheads of MPI beyond those of concurrent.futures. But as it can now use the same API as **`concurrent.futures`**, not to mention SCOOP, it is now possible to write for a single concurrency API and experiment to see which framework gives the best speed improvement to your particular application. ■

References

[Brown12] Silas S. Brown. Web Annotation with Modified-Yarowsky and Other Algorithms. *Overload* issue 112 (December 2012) page 4. The modified code is now at http://people.ds.cam.ac.uk/ssb22/adjuster/annogen.html

[MPI] MPI for Python http://mpi4py.scipy.org/

[Python] Python library documentation https://docs.python.org/3/library/concurrent.futures.html

[SCOOP] SCOOP (Scalable COncurrent Operations in Python) http://scoop.readthedocs.io/

doctest – the Lightest C++ Unit Testing Framework

C++ has many unit testing frameworks. Viktor Kirilov introduces doctest.

doctest is a fully open source light and feature-rich C++98 / C++11 single-header testing framework for unit tests and TDD. A complete example with a self-registering test that compiles to an executable looks like Listing 1.

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

int fact(int n) {
  return n <= 1 ? n : fact(n - 1) * n;
}

TEST_CASE("testing the factorial function") {
    CHECK(fact(0) == 1); // will fail
    CHECK(fact(1) == 1);
    CHECK(fact(2) == 2);
    CHECK(fact(10) == 3628800);
}
```

Listing 1

And the output from that program is in Listing 2.

Note how a standard C++ operator for equality comparison is used – doctest has one core assertion macro (it also has macros for less than, equals, greater than...) – yet the full expression is decomposed and the left and right values are logged. This is done with expression templates and C++ trickery. Also the test case is automatically registered – you don't need to manually insert it to a list.

Doctest is modeled after Catch [Catch], which is currently the most popular alternative for testing in C++ (along with googletest [GoogleTest]) – check out the differences in the FAQ [Doctest-1]. Currently a few things which Catch has are missing but doctest aims to eventually become a superset of Catch.

Motivation behind the framework: how it is different

doctest is inspired by the unittest {} functionality of the D programming language and Python's docstrings – tests can be considered a form of documentation and should be able to reside near the production code which they test (for example in the same source file a class is implemented).

A few reasons you might want to do that:

- Testing internals that are not exposed through the public API and headers of a module becomes easier.

Viktor Kirilov With 4+ years of professional experience with C++ in the game and VFX industries, Viktor currently spends his time writing open source software. His interests are the making of games and game engines and also good practices in software development – his profession is his hobby. Contact him at vik.kirilov@gmail.com

```
[doctest] doctest version is "1.1.3"
[doctest] run with "--help" for options
===============================================
main.cpp(6)
testing the factorial function

main.cpp(7) FAILED!
  CHECK( fact(0) == 1 )
with expansion:
  CHECK( 0 == 1 )

===============================================
[doctest] test cases:  1 |  0 passed |  1 failed |
[doctest] assertions:  4 |  3 passed |  1 failed |
```

Listing 2

- Lower barrier for writing tests. You don't have to:
  - make a separate source file
  - include a bunch of stuff in it
  - add it to the build system
  - add it to source control

You can just write the tests for a class or a piece of functionality at the bottom of its source file – or even header file!

- Faster iteration times – TDD becomes a lot easier.
- Tests in the production code stay in sync and can be thought of as active documentation or up-to-date comments, showing how an API is used.

The framework can still be used like any other even if the idea of writing tests in the production code doesn't appeal to you – but this is the biggest power of the framework, and nothing else comes close to being so practical in achieving this.

This isn't possible (or at least practical) with any other testing framework for C++: Catch [Catch], Boost.Test [Boost], UnitTest++ [UnitTest], cpputest [CppUTest], googletest [GoogleTest] and many others [Wikipedia]. Further details are provided below.

There are many other features [Doctest-2] and a lot more are planned in the roadmap [Doctest-3].

What makes doctest different is that it is ultra light on compile times (by orders of magnitude – further details are in the 'Compile time benchmarks' section) and is unobtrusive.

About doctest

Web Site: https://github.com/onqtam/doctest
Version tested: 1.1.3
System requirements: C++98 or newer
License & Pricing: MIT, free
Support: through the GitHub project page

if doctest is included in 1000 source files ... the overall build slowdown will be only ~10 seconds

The key differences between it and the others are:

- Ultra light – below 10ms of compile time overhead for including the header in a source file (compared to ~430ms for Catch); see the 'Compile time benchmarks' section
- The fastest possible assertion macros – 50 000 asserts can compile for under 30 seconds (even under 10 sec)
- Offers a way to remove everything testing-related from the binary with the `DOCTEST_CONFIG_DISABLE` identifier
- Doesn't pollute the global namespace (everything is in the doctest namespace) and doesn't drag any headers with it
- Doesn't produce any warnings even on the most aggressive warning levels for MSVC / GCC / Clang
  - `-Weverything` for Clang
  - `/W4` for MSVC
  - `-Wall -Wextra -pedantic` and over 35 other flags not included in these!
- Very portable and well tested C++98 – per commit tested on CI with over 220 different builds with different compilers and configurations (gcc 4.4-6.1/clang 3.4-3.9/MSVC 2008-2015, debug/release, x86/x64, linux/windows/osx, valgrind, sanitizers...)
- Just one header and no external dependencies apart from the C / C++ standard library (which are used only in the test runner)

So if doctest is included in 1000 source files (globally in a big project) the overall build slowdown will be only ~10 seconds. If Catch is used – this would mean over 350 seconds just for including the header everywhere.

If you have 50 000 asserts spread across your project (which is quite a lot) you should expect to see roughly 60–100 seconds of increased build time if using the normal expression-decomposing asserts or 10–40 seconds if you have used the fast form [Doctest-5] of the asserts.

These numbers pale in comparison to the build times of a 1000 source file project. Further details are in the 'Compile time benchmarks' section.

You also won't see any warnings or unnecessarily imported symbols from doctest, nor will you see a valgrind or a sanitizer error caused by the framework. It is truly transparent.

The main() entry point

As we saw in the example above, a `main()` entry point for the program can be provided by the framework. If, however, you are writing the tests in your production code you probably already have a `main()` function. Listing 3 shows how doctest is used from a user `main()`.

With this setup the following 3 scenarios are possible:

- running only the tests (with the `--exit` option)
- running only the user code (with the `--no-run` option)
- running both the tests and the user code

This must be possible if you are going to write the tests directly in the production code.

```
#define DOCTEST_CONFIG_IMPLEMENT
#include "doctest.h"

int main(int argc, char** argv) {
  doctest::Context ctx;
  // default - stop after 5 failed asserts
  ctx.setOption("abort-after", 5);
  // apply command line - argc / argv
  ctx.applyCommandLine(argc, argv);
  // override - don't break in the debugger
  ctx.setOption("no-breaks", true);
  // run test cases unless with --no-run
  int res = ctx.run();
  // query flags (and --exit) rely on this
  if(ctx.shouldExit())
    // propagate the result of the tests
    return res;
  // your code goes here
  return res; // + your_program_res
}
```

Listing 3

Also this example shows how defaults and overrides can be set for command line options.

Note that the `DOCTEST_CONFIG_IMPLEMENT` or `DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN` identifiers should be defined before including the framework header – but only in one source file – where the test runner will get implemented. Everywhere else just include the header and write some tests. This is a common practice for single-header libraries that need a part of them to be compiled in one source file (in this case the test runner).

Removing everything testing-related from the binary

You might want to remove the tests from your production code when building the release build that will be shipped to customers. The way this is done using doctest is by defining the `DOCTEST_CONFIG_DISABLE` preprocessor identifier in your whole project.

The effect that identifier has on the `TEST_CASE` macro for example is the following – it gets turned into an anonymous template that never gets instantiated:

```
#define TEST_CASE(name)                          \
  template <typename T>                          \
  static inline void ANONYMOUS(ANON_FUNC_)()
```

This means that all test cases are trimmed out of the resulting binary – even in Debug mode! The linker doesn't ever see the anonymous test case functions because they are never instantiated.

The `ANONYMOUS()` macro is used to get unique identifiers each time it's called – it uses the `__COUNTER__` preprocessor macro which returns an

The execution model resembles a DFS traversal – each time starting from the start of the test case and traversing the 'tree' until a leaf node is reached (one that hasn't been traversed yet)

integer with 1 greater than the last time each time it gets used. For example:

```
int ANONYMOUS(ANON_VAR_); // int ANON_VAR_5;
int ANONYMOUS(ANON_VAR_); // int ANON_VAR_6;
```

Subcases: the easiest way to share setup / teardown code between test cases

Suppose you want to open a file in a few test cases and read from it. If you don't want to copy/paste the same setup code a few times you might use the Subcases mechanism of doctest (see Listing 4).

The following text will be printed:

```
opening the file
seeking
closing... (by the destructor)
opening the file
reading
closing... (by the destructor)
```

As you can see the test case was entered twice – and each time a different subcase was entered. Subcases can also be infinitely nested. The execution model resembles a DFS traversal – each time starting from the start of the test case and traversing the 'tree' until a leaf node is reached (one that hasn't been traversed yet) – then the test case is exited by popping the stack of entered nested subcases.

Examples of how to embed tests in production code

If shipping libraries with tests, it is a good idea to add a tag in your test case names (like this: `TEST_CASE("[the_lib] testing foo")`) so the user can easily filter them out with

```
--test-case-exclude=*[the_lib]*
```

if he wishes to.

```
TEST_CASE("testing file stuff") {
  printf("opening the file\n");
  std::ifstream is ("test.txt",
                    std::ifstream::binary);

  SUBCASE("seeking in file") {
    printf("seeking\n");
    // is.seekg()
  }
  SUBCASE("reading from file") {
    printf("reading\n");
    // is.read()
  }
  printf("closing... (by the destructor)\n");
}
```

Listing 4

```
// fact.h
#pragma once

inline int fact(int n) {
  return n <= 1 ? n : fact(n - 1) * n;
}

#ifdef DOCTEST_LIBRARY_INCLUDED
TEST_CASE("[fact] testing the factorial function")
{
  CHECK(fact(0) == 1); // will fail
  CHECK(fact(1) == 1);
  CHECK(fact(2) == 2);
  CHECK(fact(10) == 3628800);
}
#endif // DOCTEST_LIBRARY_INCLUDED
```

Listing 5

- If you are shipping a header-only library there are mainly 2 options:
  1. You could surround your tests with an `ifdef` to check if doctest is included before your headers like Listing 5.
  2. You could use a preprocessor identifier (like `FACT_WITH_TESTS`) to conditionally use the tests like Listing 6.

  In both of these cases the user of the header-only library will have to implement the test runner of the framework somewhere in his executable/shared object.
- If you are developing an end product and not a library for developers, then you can just mix code and tests and implement the

```
// fact.h
#pragma once

inline int fact(int n) {
  return n <= 1 ? n : fact(n - 1) * n;
}

#ifdef FACT_WITH_TESTS

#include "doctest.h"

TEST_CASE("[fact] testing the factorial function")
{
  CHECK(fact(0) == 1); // will fail
  CHECK(fact(1) == 1);
  CHECK(fact(2) == 2);
  CHECK(fact(10) == 3628800);
}
#endif // FACT_WITH_TESTS
```

Listing 6

test runner like described in the section 'The main() entry point'. You could define the `DOCTEST_CONFIG_DISABLE` preprocessor identifier in the Release config so no tests are shipped to the customer.

- If you are developing a library which is not header-only, you could again write tests in your headers like shown above, and you could also make use of the `DOCTEST_CONFIG_DISABLE` identifier to optionally remove the tests from the source files when shipping it – or figure out a custom scheme like the use of a preprocessor identifier to optionally ship the tests - `MY_LIB_WITH_TESTS`.

Compile time benchmarks

So there are 3 types of compile time benchmarks that are relevant for doctest:

- cost of including the header
- cost of assertion macros
- how much the build times drop when all tests are removed with the `DOCTEST_CONFIG_DISABLE` identifier

In summary:

- Including the doctest header costs around 10ms compared to 250–460ms of Catch – so doctest is 25–50 times lighter
- 50 000 asserts compile for roughly 60 seconds, which is around 25% faster than Catch
- 50 000 asserts can compile for as low as 30 seconds (or even 10) if alternative assert macros [Doctest-5] are used (for power users)
- 50 000 asserts spread in 500 test cases just vanish when disabled with `DOCTEST_CONFIG_DISABLE` – all of it takes less than 2 seconds!

The lightness of the header was achieved by forward declaring everything and not including anything in the main part of the header. There are includes in the test runner implementation part of the header but that resides in only one translation unit – where the library gets implemented (by defining the `DOCTEST_CONFIG_IMPLEMENT` preprocessor identifier before including it).

Regarding the cost of asserts – note that this is for trivial asserts comparing 2 integers – if you need to construct more complex objects and have more setup code for your test cases then there will be an additional amount of time spent compiling. This depends very much on what is being tested. A user of doctest provides a real world example of this in his article [Wicht].

In the benchmarks page [Doctest-4] of the project documentation you can see the setup and more details for the benchmarks.

Conclusion

The doctest framework is really easy to get started with and is fully transparent and unintrusive. Including it and writing tests will be unnoticeable both in terms of compile times and integration (warnings, build system, etc). Using it will speed up your development process as much as possible – no other framework is so easy to use!

Note that Catch 2 is on its way (not public yet), and when it is released there will be a new set of benchmarks.

The development of doctest is supported with donations. ■

References

[Boost] http://www.boost.org/doc/libs/1_60_0/libs/test/doc/html/index.html

[Catch] https://github.com/philsquared/Catch

[CppUTest] https://github.com/cpputest/cpputest

[Doctest-1] https://github.com/onqtam/doctest/blob/master/doc/markdown/faq.md#how-is-doctest-different-from-catch

[Doctest-2] https://github.com/onqtam/doctest/blob/master/doc/markdown/features.md

[Doctest-3] https://github.com/onqtam/doctest/blob/master/doc/markdown/roadmap.md

[Doctest-4] https://github.com/onqtam/doctest/blob/master/doc/markdown/benchmarks.md

[Doctest-5] https://github.com/onqtam/doctest/blob/master/doc/markdown/assertions.md#fast-asserts

[GoogleTest] https://github.com/google/googletest

[UnitTest] https://github.com/unittest-cpp/unittest-cpp

[Wikipedia] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B

[Wicht] http://baptiste-wicht.com/posts/2016/09/blazing-fast-unit-test-compilation-with-doctest-11.html

And the winners are...

In the last *Overload* we invited our readers to vote for their favourite articles of 2016 in *CVu*, which is our sibling magazine for members, and in *Overload*.

For *Overload*, in joint first place we have:

Jonathan Wakely for C++ Antipatterns

Steve Love for A Lifetime in Python

For *CVu*, we have one clear winner:

Silas S. Brown for Why Floats are Never Equal

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

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

Correct Integer Operations with Minimal Runtime Penalties

Results of C++ integer operations are not guaranteed to be arithmetically correct. Robert Ramey introduces a library to enforce correct behaviour.

This library is intended as a drop-in replacement for all built-in integer types in any program which must:

- be demonstrably and verifiably correct.
- detect every user error such as input, assignment, etc.
- be efficient as possible subject to the constraints above.

Problem

Arithmetic operations in C/C++ are NOT guaranteed to yield a correct mathematical result. This feature is inherited from the early days of C. The behavior of **`int`**, **`unsigned int`** and others were designed to map closely to the underlying hardware. Computer hardware implements these types as a fixed number of bits. When the result of arithmetic operations exceeds this number of bits, the result will not be arithmetically correct. The following is just one example of where this causes problems:

```
int f(int x, int y){
  // this returns an invalid result for some
  // legal values of x and y !
  return x + y;
}
```

It is incumbent on the C/C++ programmer to guarantee that this behavior does not result in incorrect or unexpected operation of the program. There are no language facilities which implement such a guarantee. A programmer needs to examine each expression individually to know that his program will not return an invalid result.There are a number of ways to do this. In the above instance, INT32-C seems to recommend the following approach:

```
int f(int x, int y){
  if (((y > 0) && (x > (INT_MAX - y)))
  || ((y < 0) && (x < (INT_MIN - y)))) {
    /* Handle error */
  }
  return x + y;
}
```

This will indeed trap the error. However, it would be tedious and laborious for a programmer to alter his code to do so. Altering code in this way for all arithmetic operations would likely render the code unreadable and add another source of potential programming errors. This approach is clearly not functional when the expression is even a little more complex as is shown in the following example.

```
int f(int x, int y, int z){
  // this returns an invalid result for some
  // legal values of x and y !
  return x + y * z;
}
```

Robert Ramey Robert is a freelance C++ developer living in Santa Barbara, California. He is author of the Boost Serialization Library and an active member of the Boost community. He is also frequent speaker at C++ conferences. ramey@rrsd.com

This example addresses only the problem of undefined/erroneous behavior related to overflow of the addition operation as applied to the type **`int`**. Similar problems occur with other built-in integer types such as **`unsigned`**, **`long`**, etc. And it also applies to other operations such as subtraction, multiplication etc. C/C++ often automatically and silently converts some integer types to others in the course of implementing binary operations and similar problems occur in this case as well. Since the problems and their solution are similar, we'll confine the current discussion to just this example.

Solution

This library implements special versions of **`int`**, **`unsigned`**, etc. which behave exactly like the original ones **except** that the results of these operations are guaranteed to be either arithmetically correct or invoke an error. Using this library, the above example would be rendered as:

```
#include <boost/safe_numeric/safe_integer.hpp>
using namespace boost::numeric;
safe<int> f(safe<int> x, safe<int> y){
  return x + y; // throw exception if correct
                // result cannot be returned
}
```

Library code in this document resides in the name space **`boost::numeric`**. This name space has generally been eliminated from text, code and examples in order to improve readability of the text.

The addition expression is checked at runtime or (if possible) at compile time to trap any possible errors resulting in incorrect arithmetic behavior. This will permit one to write arithmetic expressions that cannot produce an erroneous result. Instead, one and only one of the following is guaranteed to occur.

- the expression will yield the correct mathematical result
- the expression will emit a compilation error.
- the expression will invoke a runtime exception.

In other words, the **library absolutely guarantees that no arithmetic expression will yield incorrect results**.

How it works

The library implements special versions of **`int`**, **`unsigned`**, etc. named **`safe<int>`**, **`safe<unsigned int>`**, etc. These behave exactly like the underlying types **except** that expressions using these types fulfill the above guarantee. These types are meant to be 'drop-in' replacements for the built-in types they are meant to replace. So things which are legal – such as assignment of a **`signed`** to an **`unsigned`** value – are not trapped at compile time, as they are legal C/C++ code. Instead, they are checked at runtime to trap the case where this (legal) operation would lead to an arithmetically incorrect result.

Note that the library addresses arithmetical errors generated by straightforward C/C++ expressions. Some of these arithmetic errors are defined as conforming to the C/C++ standards while others are not. So

Facilities particular to C++14 are employed to minimize any runtime overhead. In many cases there is no runtime overhead at all.

characterizing this library as addressing undefined behavior of C/C++ numeric expressions would be misleading.

Facilities particular to C++14 are employed to minimize any runtime overhead. In many cases there is no runtime overhead at all. In other cases, a program using the library can be slightly altered to achieve the above guarantee without any runtime overhead.

Additional features

Operation of safe types is determined by template parameters which specify a pair of policy classes which specify the behavior for type promotion and error handling. In addition to the usage serving as a drop-in replacement for standard integer types, users of the library can:

- Select or define an exception policy class to specify handling of exceptions.
  - throw exception on runtime, trap at compile time.
  - trap at compiler time all operations which might fail at runtime.
  - specify custom functions which should be called at runtime
- Select or define a promotion policy class to alter the C/C++ type promotion rules. This can be used to
  - use C/C++ native type promotion rules so that, except for throwing/trapping of exceptions on operations resulting in incorrect arithmetic behavior, programs will operate identically when using/not using safe types.
  - replace C/C++ native promotion rules with ones which are arithmetically equivalent but minimize the need for runtime checking of arithmetic results.
  - replace C/C++ native promotion rules with ones which emulate other machine architectures. This is designed to permit the testing of C/C++ code destined to be run on another machine on one's development platform. Such a situation often occurs while developing code for embedded systems.
- Enforce of other program requirements using ranged integer types. The library includes the types `safe_range<Min, Max>` and `safe_literal<N>`. These types can be used to improve program correctness and performance.

Requirements

This library is composed entirely of C++ Headers. It requires a compiler compatible with the C++14 standard.

The following Boost Libraries must be installed in order to use this library

- MPL
- Integer
- Config
- Concept Checking
- Tribool
- Enable_if

The Safe Numerics library is delivered with an exhaustive suite of test programs. Users who choose to run this test suite will also need to install the Boost.Preprocessor library.

Scope

This library currently applies only to built-in integer types. Analogous issues arise for floating point types but they are not currently addressed by this version of the library. User or library defined types such as arbitrary precision integers can also have this problem. Extension of this library to these other types is not currently under development but may be addressed in the future. This is one reason why the library name is 'safe numeric' rather than 'safe integer' library.

Eliminating runtime penalty

Up until now, we've focused on detecting when incorrect results are produced and handling these occurrences either by throwing an exception or invoking some designated function. We've achieved our goal of detecting and handling arithmetically incorrect behavior – but at what cost. It is a fact that many C++ programmers will find this trade-off unacceptable. So the question arises as to how we might minimize or eliminate this runtime penalty.

The first step is to determine what parts of a program might invoke exceptions. Listing 1 is similar to previous examples but uses a special exception policy: `trap_exception`.

Now, any expression which ***might*** fail at runtime is flagged with a compile time error. There is no longer any need for `try`/`catch` blocks. Since this program does not compile, the **library absolutely guarantees that no arithmetic expression will yield incorrect results**. This is our

```
#include <iostream>
#include "../include/safe_integer.hpp"
#include "../include/exception.hpp"
  // include exception policies

using safe_t = boost::numeric::safe<
int,
boost::numeric::native,
boost::numeric::trap_exception
  // note use of "trap_exception" policy!
>;

int main(int argc, const char * argv[]){
std::cout << "example 81:\n";
safe_t x(INT_MAX);
safe_t y(2);
safe_t z = x + y; // will fail to compile !
return 0;
}
```

Listing 1

In short, given a binary operation, we silently promote the types of the operands to a wider result type so the result cannot overflow

original goal. Now all we need to do is make the program work. There are a couple of ways to do this.

Using automatic type promotion

The C++ standard describes how binary operations on different integer types are handled. Here is a simplified version of the rules:

- promote any operand smaller than **`int`** to an **`int`** or **`unsigned int`**.
- if the signed operand is larger than the signed one, the result will be signed, otherwise the result will be unsigned.
- expand the smaller operand to the size of the larger one

So the result of the sum of two integer types may result in another integer type. If the values are large, the result can exceed the size that the resulting integer type can hold. This is what we call 'overflow'. The C/C++ standard characterises this as undefined behavior and leaves to compiler implementors the decision as to how such a situation will be handled. Usually, this means just truncating the result to fit into the result type – which sometimes will make the result arithmetically incorrect. However, depending on the compiler, compile time switch settings, such a case may result in some sort of run time exception.

The complete signature for a safe integer type is:

```
template <
  class T,           // underlying integer type
  class P = native, // type promotion policy class
  class E = throw_exception
                     // error handling policy class
>
safe;
```

The promotion rules implemented in the default **`native`** type promotion policy are consistent with those of standard C++. Up until now, we've focused on detecting when this happens and invoking an interrupt or other kind of error handler.

But now we look at another option. Using the **`automatic`** type promotion policy, we can change the rules of C++ arithmetic for safe types to something like the following:

- For any C++ numeric type, we know from **`std::numeric_limits`** what the maximum and minimum values that a variable can be – this defines a closed interval.
- For any binary operation on these types, we can calculate the interval of the result at compile time.
- From this interval we can select a new type which can be guaranteed to hold the result and use this for the calculation. This is more or less equivalent to the following code:

```
int x, y;
int z = x + y // which could overflow
int x, y;
long z = (long)x + (long)y;
             // which can never overflow
```

  One could do this by editing this code manually, but such a task would be tedious, error prone, and leave the resulting code hard to read and verify. Using the **`automatic`** type promotion policy will achieve the equivalent result without these problems.
- Since the result type is guaranteed to hold the result, there is no need to check for errors – they can't happen!!! The usage of the **`trap_exception`** exception policy enforces this guarantee.
- Since there can be no errors, there is no need for **`try`**/**`catch`** blocks.
- The only runtime error checking we need to do is when safe values are initialized or assigned from values which are 'too large'. These are infrequent occurrences which generally have little or no impact on program running time. And many times, one can make small adjustments in selecting the types in order to eliminate all runtime penalties.

In short, given a binary operation, we silently promote the types of the operands to a wider result type so the result cannot overflow. This is a fundamental departure from the C++ Standard behavior.

If the interval of the result cannot be guaranteed to fit in the largest type that the machine can handle (usually 64 bits these days), the largest available integer type with the correct result sign is used. So even with our 'automatic' type promotion scheme, it's still possible to overflow. In this case, and only this case, is runtime error checking code generated. Depending on the application, it should be rare to generate error checking code, and even more rare to actually invoke it. Any such instances are detected at compile time by the **`trap_exception`** exception policy.

Listing 2 illustrates how to use automatic type promotion to eliminate all runtime penalty. It produces the following output:

```
example 82:
x = <int>[-2147483648,2147483647] = 2147483647
y = <int>[-2147483648,2147483647] = 2
z = <long>[-4294967296,4294967294] = 2147483649
```

The output uses a custom output manipulator for safe types to display the underlying type and its range as well as current value. Note that:

- the **`automatic`** type promotion policy has rendered the result of the some of two **`integers`** as a **`long`** type.
- our program compiles without error – even when using the **`trap_exception`** exception policy
- We do not need to use **`try`**/**`catch`** idiom to handle arithmetic errors – we will have none.
- We only needed to change two lines of code to achieve our goal.

Using safe_range

Instead of relying on automatic type promotion, we can just create our own types in such a way that we know they won't overflow. In Listing 3, we presume we know that the values we want to work with fall in the range [-24,82]. So we 'know' the program will always result in a correct result. But since we trust no one, and since the program could change and

Instead of relying on automatic type promotion, we can just create our own types in such a way that we know they won't overflow

```
#include <iostream>
#include "../include/safe_integer.hpp"
#include "../include/exception.hpp"
#include "../include/automatic.hpp"
#include "safe_format.hpp"
  // prints out range and value of any type
using safe_t = boost::numeric::safe<
  int,
  boost::numeric::automatic,
    // note use of "automatic" policy!!!
  boost::numeric::trap_exception
>;

int main(int argc, const char * argv[]){
  std::cout << "example 82:\n";
  safe_t x(INT_MAX);
  safe_t y = 2;
  std::cout << "x = " << safe_format(x)
    << std::endl;
  std::cout << "y = " << safe_format(y)
    << std::endl;
  std::cout << "z = " << safe_format(x + y)
    << std::endl;
  return 0;
}
```

Listing 2

the expressions be replaced with other ones, we'll still use the **trap_exception** exception policy to verify at compile time that what we 'know' to be true is in fact true:

- **safe_signed_range** defines a type which is limited to the indicated range. Out of range assignments will be detected at compile time if possible (as in this case) or at run time if necessary.
- **safe_signed_literal** defines a constant with a specific value. Defining constants in this way enables the library to correctly anticipate the range of the results of arithmetic expressions.
- The usage of **trap_exception** will mean that any assignment to z which could be outside the legal range will result in a compile time error.
- So if this program compiles, it's guaranteed to return a valid result.

This program produces the following run time output.

```
example 83:
x = <signed char>[10,10] = 10
y = <signed char>[67,67] = 67
z = <signed char>[-24,82] = 77
```

```
#include <iostream>
#include "../include/safe_range.hpp"
#include "../include/safe_literal.hpp"
#include "../include/exception.hpp"
#include "../include/native.hpp"
#include "safe_format.hpp"
  // prints out range and value of any type
using namespace boost::numeric;
  // for safe_literal

// create a type for holding small integers. We
// "know" that C++ type promotion rules will work
// such that addition will never overflow. If we
// change the program to break this, the usage
// of the trap_exception promotion policy will
// prevent compilation.
using safe_t = safe_signed_range<
  -24,
  82,
  native, // C++ type promotion rules work
          // OK for this example
  trap_exception // catch problems at compile time
>;

int main(int argc, const char * argv[]){
  std::cout << "example 83:\n";
  // the following would result in a compile time
  // error since the sum of x and y wouldn't be in
  // the legal range for z.
  // const safe_signed_literal<20> x;
  const safe_signed_literal<10> x; // no problem
  const safe_signed_literal<67> y;

  const safe_t z = x + y;
  std::cout << "x = " << safe_format(x)
    << std::endl;
  std::cout << "y = " << safe_format(y)
    << std::endl;
  std::cout << "z = " << safe_format(z)
    << std::endl;
  return 0;
}
```

Listing 3

Mixing approaches

For purposes of exposition, we've divided the discussion of how to eliminate runtime penalties by the different approaches available. A realistic program would likely include all techniques mentioned above. Consider Listing 4:

- As before, we define a **safe_t** to reflect our view of legal values for this program. This uses **automatic** type promotion policy as

these types are guaranteed to contain legal values and will throw an exception when this guarantee is violated

well as **trap_exception** exception policy to enforce elimination of runtime penalties.

- The function **f** accepts only arguments of type **safe_t** so there is no need to check the input values. This performs the functionality of ***programming by contract*** with no runtime cost.

```
#include <stdexcept>
#include <iostream>
#include "../include/safe_range.hpp"
#include "../include/automatic.hpp"
#include "../include/exception.hpp"

#include "safe_format.hpp"
  // prints out range and value of any type

using namespace boost::numeric;
using safe_t = safe_signed_range<
  -24,
  82,
  automatic,
  trap_exception
>;
// define variables use for input
using input_safe_t = safe_signed_range<
  -24,
  82,
  automatic,
          // we don't need automatic in this case
  throw_exception // these variables need to
>;

// function arguments can never be outside of
// limits
auto f(const safe_t & x, const safe_t & y){
  auto z = x + y; // we know that this cannot fail
  std::cout << "z = " << safe_format(z)
    << std::endl;
  std::cout << "(x + y) = " << safe_format(x + y)
    << std::endl;
  std::cout << "(x - y) = " << safe_format(x - y)
    << std::endl;
  return z;
}

int main(int argc, const char * argv[]){
  std::cout << "example 84:\n";
  input_safe_t x, y;
  try{
    std::cin >> x >> y; // read varibles,
                        // maybe throw exception
  }
```

Listing 4

```
  catch(const std::exception & e){
    // none of the above should trap.
    // Mark failure if they do
    std::cout << e.what() << std::endl;
    return 1;
```

Listing 4 (cont'd)

- In addition, we define **input_safe_t** to be used when reading variables from the program console. Clearly, these can only be checked at runtime so they use the **throw_exception** policy. When variables are read from the console they are checked for legal values. We need no ad hoc code to do this, as these types are guaranteed to contain legal values and will throw an exception when this guarantee is violated. In other words, we automatically get checking of input variables with no additional programming.
- On calling of the function **f**, arguments of type **input_safe_t** are converted to values of type **safe_t**.

  In this particular example, it can be determined at compile time that construction of an instance of a **safe_t** from an **input_safe_t** can never fail. Hence, no **try**/**catch** block is necessary. The usage of the **trap_exception** policy for **safe_t** types guarantees this to be true at compile time.

Here is the output from the program when values 12 and 32 are input from the console:

```
  example 84:
  12 32
  x<signed char>[-24,82] = 12
  y<signed char>[-24,82] = 32
  z = <short>[-48,164] = 44
  (x + y) = <short>[-48,164] = 44
  (x - y) = <short>[-106,106] = -20
<short>[-48,164] = 44
```

Background

This library started out as a re-implementation of the facilities provided by David LeBlanc's SafeInt Library [http://safeint.codeplex.com]. I found this library very well done in every way. My main usage was to run unit tests for my embedded systems projects on my PC. Still, I had a few issues.

- It was a lot of code in one header – 6400 lines. Very unwieldy to understand, modify and maintain.
- I couldn't find separate documentation other than that in the header file.
- It didn't use Boost conventions for naming.
- It required porting to different compilers.
- It had a very long license associated with it.
- I could find no test suite for the library.

If the range of the result type includes the range of the result of the operation, no run time checking of the result is necessary

This version addresses these issues. It exploits many facilities of C++14 and the Boost libraries to reduce the number of lines of source code to approximately 4700.

Library internals

This library should compile and run correctly on any conforming C++14 compiler.

The Safe Numerics library is implemented in terms of some more fundamental software components described here. It is not necessary to know about these components to use the library. This information has been included to help those who want to understand how the library works so they can extend it, correct bugs in it, or understand its limitations. These components are also interesting in their own right. For all these reasons, they are described here. In general terms, the library works in the following manner:

- All unary/binary expressions where one of the operands is a 'safe' type are overloaded. These overloads are declared and defined in the header file `safe_integer.hpp`. SFINAE – 'Substitution Failure Is Not An Error' – and **`std::enable_if`** are key features of C++ used to define these overloads in a correct manner.
- Each overloaded operation implements the following procedure at compile time:
  - Retrieve range of values for each operand of type T from both: **`std::numeric_limits<T>::min()`** **`std::numeric_limits<T>::max()`**.
  - Given the ranges of the operands, determine the range of the result of the operation using interval arithmetic. This is implemented in the `interval.hpp` header file using **`constexpr`** facility of C++14.
  - If the range of the result type includes the range of the result of the operation, no run time checking of the result is necessary, so the operation reduces to the original built-in C/C++ operation.
  - Otherwise, the operation is implemented as a 'checked integer operation' at run time. This operation returns a variant which will contain either a correct result or an **`enum`** indicating why a correct result could not be obtained. The variant object is implemented in the header file `checked_result.hpp` and the checked operations are implemented in `checked.hpp`.
  - If a valid result has been obtained, it is passed to the caller.
  - Otherwise, an exception is invoked.

Rationale and FAQ

1. Is this really necessary? If I'm writing the program with the requisite care and competence, problems noted in the introduction will never arise. Should they arise, they should be fixed 'at the source' and not with a 'band aid' to cover up bad practice.
   This surprised me when it was first raised. But some of the feedback I've received makes me thing that it's a widely held view. The best answer is to consider the cases in the Tutorials and Motivating Examples section of the library documentation.
2. Can safe types be used as drop-in replacement for built-in types?
   Almost. Replacing all built-in types with their safe counterparts should result in a program that will compile and run as expected. In some cases compile time errors will occur and adjustments to the source code will be required. Typically these will result in code which is more correct.
3. Why is Boost.Convert not used?
   I couldn't figure out how to use it from the documentation.
4. Why is the library named 'safe ...' rather than something like 'checked ...'?
   I used 'safe' in large part as this is what has been used by other similar libraries. Maybe a better word might have been 'correct' but that would raise similar concerns. I'm not inclined to change this. I've tried to make it clear in the documentation what the problem that the library addressed is.
5. Given that the library is called 'numerics', why is floating point arithmetic not addressed?
   Actually, I believe that this can/should be applied to any type **`T`** which satisfies the type requirement 'Numeric' type as defined in the documentation. So there should be specializations **`safe<float>`** and related types as well as new types like **`safe<fixed_decimal>`** etc. But the current version of the library only addresses integer types. Hopefully the library will evolve to match the promise implied by its name.
6. Isn't putting a defensive check just before any potential undefined behavior often considered a bad practice?
   By whom? Is leaving code which can produce incorrect results better? Note that the documentation contains references to various sources which recommend exactly this approach to mitigate the problems created by this C/C++ behavior. See [Seacord].
7. It looks like the implementation presumes two's complement arithmetic at the hardware level. So this library is not portable, correct? What about other hardware architectures?
   As far as is known as of this writing, the library does not presume that the underlying hardware is two's complement. However, this has yet to be verified in a rigorous way.
8. Why do you specialize **`numeric_limits`** for 'safe' types? Do you need it?
   `safe<T>` behaves like a 'number' just as **`int`** does. It has **`max`**, **`min`**, etc Any code which uses numeric limits to test a type **`T`** should work with **`safe<T>`**. **`safe<T>`** is a drop-in replacement for **`T`** so it has to implement all the operations.
9. According to C/C++ standards, unsigned integers cannot overflow – they are modular integers which 'wrap around'. Yet the Safe Numerics library detects and traps this behavior as errors. Why is that?
   The guiding purpose of the library is to trap incorrect arithmetic behavior – not just undefined behavior. Although a savvy user may understand and keep present in his mind that an unsigned integer is

By doing range arithmetic at compiler-time, I could skip runtime checking on many/most integer operations

really a modular type, the plain reading of an arithmetic expression conveys the idea that all operands are integers. Also in many cases, unsigned integers are used in cases where modular arithmetic is not intended, such as array indexes. Finally, the modulus for such an integer would vary depending upon the machine architecture. For these reasons, in the context of this library, an unsigned integer is considered to a representation of a subset of integers. Note that this decision is consistent with INT30-C, "Ensure that unsigned integer operations do not wrap" in the CERT C Secure Coding Standard [Seacord].

10. Why does the library require C++14?

    The original version of the library used C++11. Feedback from CPPCon, Boost Library Incubator [www.blincubator.com] and Boost developer's mailing list convinced me that I had to address the issue of run-time penalty much more seriously. I resolved to eliminate or minimize it. This led to more elaborate meta-programming. But this wasn't enough. It became apparent that the only way to really minimize run-time penalty was to implement compile-time integer range arithmetic – a pretty elaborate sub library. By doing range arithmetic at compiler-time, I could skip runtime checking on many/most integer operations. C++11 constexpr wasn't quite enough to do the job. C++14 constexpr can do the job. The library currently relies very heavily on C++14 constexpr. I think that those who delve into the library will be very surprised at the extent that minor changes in user code can produce guaranteed correct integer code with zero run-time penalty.

11. This is a C++ library, yet you refer to C/C++. Which is it?

    C++ has evolved way beyond the original C language. But C++ is still (mostly) compatible with C. So most C programs can be compiled with a C++ compiler. The problems of incorrect arithmetic afflict both C and C++. Suppose we have a legacy C program designed for some embedded system.

    - Replace all **`int`** declarations with **`int16_t`** and all **`long`** declarations with **`int32_t`**.
    - Create a file containing something like Listing 5 and include it at the beginning of every source file.
    - Compile tests on the desktop with a C++14 compiler and with the macro **`TEST`** defined.
    - Run the tests and change code to address any thrown exceptions.

    This example illustrates how this library, implemented with C++14 can be useful in the development of correct code for programs written in C.

Current status

The library is currently in the Boost Review Queue [http://www.boost.org/community/review_schedule.html]. The proposal submission can be found in the Boost Library Incubator [http://blincubator.com/bi_library/safe-numerics/?gform_post_id=426]

- The library is currently limited to integers.
- Although care has been taken to make the library portable, it's likely that at least some parts of the implementation – particularly **`checked`** arithmetic – depend upon two's complement representation of integers. Hence the library is probably not currently portable to other architectures.

```
#ifdef TEST
// using C++ on test platform
#include <cstdint>
#include <safe_integer.hpp>
#include <cpp.hpp>
using pic16_promotion = boost::numeric::cpp<
  8,  // char
  8,  // short
  8,  // int
  16, // long
  32  // long long
>;
// define safe types used desktop version of
// the program.
template <typename T>
  // T is char, int, etc data type
using safe_t = boost::numeric::safe<
  T,
  pic16_promotion,
  boost::numeric::throw_exception
    // use for compiling and running tests
>;
typedef safe_t<std::int16_t> int16_t;
typedef safe_t<std::int32_t> int32_t;
#else
/* using C on embedded platform */
typedef int int16_t;
typedef long int32_t;
#endif
```

Listing 5

- Currently the library permits a **`safe<int>`** value to be uninitialized. This supports the goal of 'drop-in' replacement of C++/C built-in types with safe counter parts. On the other hand, this breaks the 'always valid' guarantee.
- The library is not quite a 'drop-in' replacement for all built-in integer types. In particular, C/C++ implements implicit conversions and promotions between certain integer types which are not captured by the operation overloads used to implement the library. In practice these case are few and can be addressed with minor changes to the user program to avoid these silent implicit conversions. ■

Acknowledgements

This library would never have been created without inspiration, collaboration and constructive criticism from multiple sources.

David LeBlanc

This library is inspired by David LeBlanc's SafeInt Library [http://safeint.codeplex.com]. I found this library very well done in every way and useful in my embedded systems work. This motivated me to take to the 'next level'.

Andrzej Krzemieński [https://akrzemi1.wordpress.com]

Andrzej Commented and reviewed the library as it was originally posted on the Boost Library Incubator [www.blincubator.com]. The the consequent back and forth motivated me to invest more effort in developing documentation and examples to justify the utility, indeed the necessity, for this library. He also noted many errors in code, documentation, and tests. Without his interest and effort, I do not believe the library would have progressed beyond its initial stages.

Boost [www.boost.org]

As always, the Boost Developer's mailing list has been the source of many useful observations from potential users and constructive criticism from very knowledgeable developers.

Bibliography

[coker] Zack Coker. Samir Hasan. Jeffrey Overbey. Munawar Hafiz. Christian Kästner. *Integers In C: An Open Invitation To Security Attacks?* [https://www.cs.cmu.edu/~ckaestne/pdf/csse14-01.pdf] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]. JTC1/SC22/WG21 – The C++ Standards Committee – ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/]. January 15, 2012. Coker

[crowl1] Lawrence Crowl. *C++ Binary Fixed-Point Arithmetic* [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3352.html] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]. JTC1/SC22/WG21 – The C++ Standards Committee – ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/]. January15, 2012. Crowl

[crowl2] Lawrence Crowl. Thorsten Ottosen. *Proposal to add Contract Programming to C++* [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.html] [http://www.cert.org/secure-coding/publications/books/securecoding-c-c-second-edition.cfm?]. WG21/N1962 and J16/06-0032 – The C++ Standards Committee – ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/]. February 25, 2006. Crowl & Ottosen

[dietz] Will Dietz. Peng Li. John Regehr. Vikram Adve. *Understanding Integer Overflow in C/C++* [http://www.cs.utah.edu/~regehr/papers/overflow12.pdf]. *Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland* [http://dl.acm.org/citation.cfm?id=2337223&picked=prox]. June 2012. Dietz

[garcia] J. Daniel Garcia. *C++ language support for contract programming* [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4293.pdf] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-secondedition.cfm?]. WG21/N4293 – The C++ Standards Committee – ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/]. December 23, 2014. Garcia

[katz] Omer Katz. *SafeInt code proposal* [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-secondedition.cfm?]. *Boost Developer's List* [https://groups.google.com/a/isocpp.org/forum/?fromgroups#!forum/stdproposals]. Katz

[keaton] David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, and Timothy Wilson. *As-if Infinitely Ranged Integer Model* [http://resources.sei.cmu.edu/asset_files/TechnicalNote/2009_004_001_15074.pdf] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?] Software Engineering Institute [http://www.sei.cmu.edu]. CMU/SEI-2009-TN-023.

[leblanc1] David LeBlanc. *Integer Handling with the C++ SafeInt Class* [https://msdn.microsoft.com/en-us/library/ms972705.aspx]. Microsoft Developer Network [https://www.cert.org]. January 7, 2004. LeBlanc

[leblanc2] David LeBlanc. *SafeInt* [https://safeint.codeplex.com]. CodePlex [https://www.cert.org]. Dec 3, 2014. LeBlanc

[lions] Jacques-Louis Lions. *Ariane 501 Inquiry Board report* [https://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report]. Wikisource [https://en.wikisource.org/wiki/Main_Page]. July 19, 1996. Lions

[matthews] Hubert Matthews. *CheckedInt: A Policy-Based Range-Checked Integer* [https://accu.org/index.php/journals/324] . *Overload* Journal #58 [https://accu.org/index.php]. December 2003. Matthews

[mouawad] Jad Mouawad. *F.A.A Orders Fix for Possible Power Loss in Boeing 787* [http://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html?_r=0] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]. New York Times. April 30, 2015.

[plakosh] Daniel Plakosh. *Safe Integer Operations* [https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding/312-BSI.html]. U.S. Department of Homeland Security [https://buildsecurityin.us-cert.gov]. May 10, 2013. Plakosh

[seacord1] Robert C. Seacord. *Secure Coding in C and C++* [http://www.cert.org/secure-coding/publications/books/securecoding-c-c-second-edition.cfm?]. 2nd Edition. Addison-Wesley Professional. April 12, 2013. 978-0321822130. Seacord

[seacord2] Robert C. Seacord. *INT30-C. Ensure that operations on unsigned integers do not wrap* [https://www.securecoding.cert.org/confluence/display/seccode/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow?showComments=false]. Software Engineering Institute, Carnegie Mellon University [https://www.cert.org]. August 17, 2014. INT30-C

[seacord3] Robert C. Seacord. *INT32-C. Ensure that operations on signed integers do not result in overflow* [https://www.securecoding.cert.org/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap]. Software Engineering Institute, Carnegie Mellon University [https://www.cert.org]. August 17, 2014. INT32-C

[stroustrup] Bjarn Stroustrup. *The C++ Programming Language Fourth Edition*. Addison-Wesley [http://www.open-std.org/jtc1/sc22/wg21/] . Copyright © 2014 by Pearson Education, Inc. January 15, 2012. Stroustrup

[forum] Forum Posts. *C++ Binary Fixed-Point Arithmetic* [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3352.html] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]. ISO C++ Standard Future Proposals [https://groups.google.com/a/isocpp.org/forum/?fromgroups#!forum/std-proposals]. Forum

Afterwood

Trying to find a good candidate for a role is hard. Chris Oldwood reminisces on various factors that influence interviewers.

After more than two decades as a programmer, you would think I'd have got this whole screening and interviewing process licked, but just when I think I've found the optimal solution, I once again find another 'edge case' that blows my theory out of the water. The problem, of course, with hiring people is that it involves human beings, and humans are notoriously difficult to deal with because, well, they're all different.

In my early years as a professional programmer, I suffered from an arrogance which suggested that all the decent programmers in the world probably had backgrounds in engineering rather than computer science. Having studied electronic systems engineering at university myself (though sucking badly at everything apart from the assembly programming side due to my bedroom coder upbringing) I entered professional programming with a good appreciation for the hardware which was useful in the industry where I started out – PC graphics applications.

Correlation, of course, does not imply causation and with such a small data set to work from it's not surprising I came to such a conclusion (I suck at statistics too, it seems). I subconsciously fostered this view for some years, which naturally biased my opinion of people whose CVs crossed my path. I don't remember discarding anyone solely based on that criterion but I'm sure any candidate who reached the interview stage will unknowingly have had yet another prejudice to overcome. Eventually I got to work in a big team at a large company and mixed with all kinds of people from different backgrounds, which helped dissipate that particular notion.

However, there is just not enough time to interview every candidate face-to-face that crosses your path and so you have to develop multiple heuristics if you are ever to separate the proverbial 'wheat from the chaff'. Every time I think I've come up with a fairly solid heuristic, I invariably end up working with special people that buck the trend and I once again begin to feel discomfort in how many others of a similar calibre I may have discarded because not enough of what I felt were the right boxes were ticked.

For example, when hiring for senior roles, which is what I've mostly been involved in, I would historically hope for them to show some kind of interest in programming outside of their 9 to 5 job, especially when they describe themselves as 'passionate' about the career. This does not have to be anything as grandiose as leading a major open source project or writing a column, but just something simple like regularly attending a meetup, or reading a journal or blog. It turns out some people are really good at keeping a sensible work/life balance. This doesn't mean they don't help out when things go awry, but that they don't unnecessarily subscribe to a hero mentality.

Another heuristic I've tried on pre-screening phone calls is to get the candidate to talk about their programming 'war stories'. Any meetups or conferences I've attended that result in a trip to the pub usually end up with various people describing some of the bizarre code or production incidents they've had to deal with in their past. There is almost a case of one-upmanship going on as the night progresses and alcohol is consumed. However, unless they are of our own making and we feel entirely comfortable sharing our mishaps, these stories usually come at the expense of someone else and therefore we may be behaving no better than the archetypal builder who denigrates the work of others. Being asked to do that with a prospective employer naturally makes some people uncomfortable. Another idea scrapped.

It's now been a decade since the classic Fizz Buzz programming exercise entered the mainstream consciousness and yet I'm still amazed at the number of overly complex solutions I've had to review from experienced developers. Paradoxically, I've also rejected candidates based on this simple programming problem only to discover they've been hired because of other forces at play. I consider this a lucky escape and rejoice that a sane hiring process has won out, but once again I feel uncomfortable with the outcome of a screening process designed solely to weed out highly undesirable candidates. I've found myself questioning over and over again what signs I'd missed this time around. If the purpose is to hire someone who can write simple code, what does it say when asking them to do just that with a simple problem in their own time ends up going awry? It seems some people really struggle to believe you when you say you want a simple solution to a simple problem; they want to impress you, to show you what they can really do.

My early experiences in the interviewing process (as hirer, not seeker) were definitely predicated on a desire to find someone like myself. I don't believe this was in a narcissistic kind of way, but more borne out of the need to find someone suitable using the minimal time out from normal duties. Is it any wonder all the heuristics I came up with were really just how I see myself? And then there is the need to feel that we must choose the best candidate from those we have shortlisted as if we were out shopping for a new car and wanted to be sure we had gotten the best deal possible. Too many times a good candidate has gotten away because we hoped that someone later would be even better. Hiring, just like programming it seems, is also subject to the problems of 'gold plating'.

So, when the next recruitment agent or HR person asks the inevitable question 'what should I be looking out for to identify the best candidates for you?' the best I can do is shrug my shoulders. Yes, I can provide some useful heuristics that may help me prioritise candidates through the initial screening process but ultimately it seems I still have to fall back on the most subjective heuristic of all – gut instinct. ■

Chris Oldwood Chris 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

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