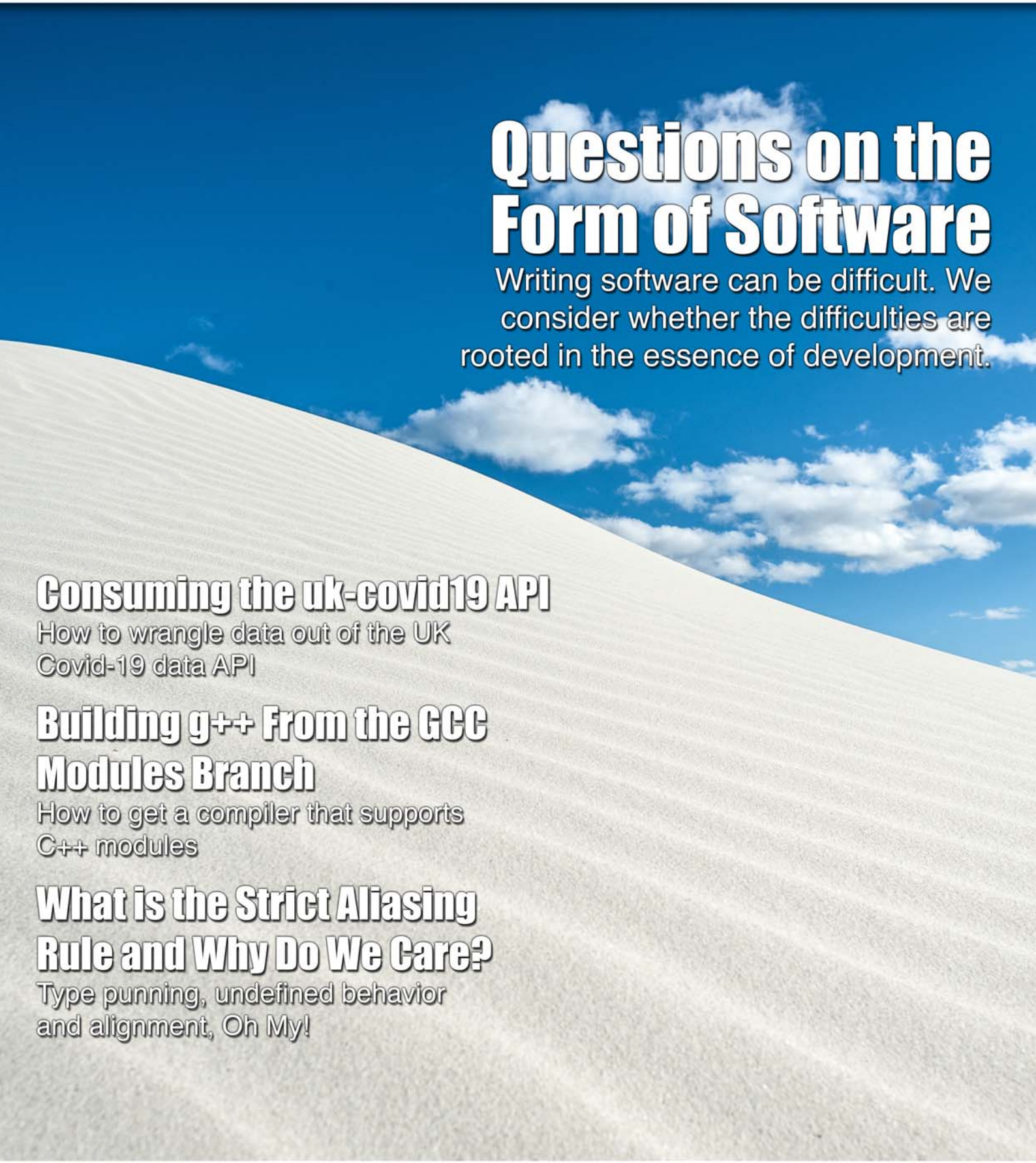
overload 160

DECEMBER 2020 £4.50



Questions on the Form of Software

Writing software can be difficult. We consider whether the difficulties are rooted in the essence of development.

Consuming the uk-covid19 API

How to wrangle data out of the UK Covid-19 data API

Building g++ From the GCC Modules Branch

How to get a compiler that supports C++ modules

What is the Strict Aliasing Rule and Why Do We Care?

Type punning, undefined behavior and alignment, Oh My!

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 160

December 2020

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

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

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 161 should be submitted by 1st January 2021 and those for Overload 162 by 1st March 2021.

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 Questions on the Form of Software
Lucian Radu Teodorescu considers whether the difficulties in writing software are rooted in the essence of development.

8 Building g++ from the GCC Modules Branch
Roger Orr demonstrates how to get a compiler that supports modules up and running.

10 Consuming the uk-covid19 API
Donald Hernik demonstrates how to wrangle data out of the UK API.

13 What is the Strict Aliasing Rule and Why Do We Care?
Strict aliasing is explained.

20 Afterwood
Chris Oldwood explains why he thinks Design Patterns are still relevant.

Copyrights and Trade Marks

Debt – My First Thirty Years

Reflecting on code often reveals gnarliness. Frances Buontempo reminds herself about all the tech debt she's ever caused.

I still owe our readers an editorial, and know I have accumulated a huge debt over the last several years. Fortunately, you don't seem to be charging me interest, so there is hope. As soon as you charge interest on a debt, it is possible for the debt to keep growing and become impossible to pay back. Charging interest, or at least excessive interest, is sometimes referred to as usury. Anyone taking advantage like this is sometimes called a loan shark. By lending money to someone who is desperate, needing food or a way to pay rent, or similar, and then threatening them with violence if they don't repay causes debt, and despair, to spiral out of control. Not a good place to be.

What's this got to do with programming, you may ask? Almost nothing, in one sense. And yet, tech debt is a term that gets banded around, so perhaps it's got everything to do with programming. Why do we describe confusing, hard to maintain code as debt? We haven't borrowed money to cover it, so can't be charged interest. We may have cut a corner or two, in order to get something out the door quickly, leaving a problem to deal with later. I say later, but it often turns out to be sooner. With untested edge cases, things may blow up regularly. Without useful logging, you can't figure out what happened. If the so-called quick win, cut corner, or tech debt, means people have to down tools and fix things on a regular basis, you have in fact slowed everything down. Cutting corners is very different to being in debt. Such short cuts are more like being in danger.

I recently described cutting corners as being like a Koch snowflake [Wikipedia-1]. Now, to build this 'snowflake', you start with a triangle and cut out the middle of each side replacing it with a smaller triangle, and continue ad infinitum. This, in some sense, is the opposite of cutting corners, since you add more corners each time, and tend towards something 8/5 times the original area. Cutting corners in code does often involve slapping extra bits in, copying and pasting code, wedging in a few booleans and `if`/`else` code paths and the like. Such short cuts are more like spare parts gaffer-taped on. Now, cutting corners comes from the idea of trespassing across a farmer's field, or driving on the wrong side of the road at a bend, rather than sticking to the legal route. You could suggest illegal practices aren't necessarily wrong and there are some perfectly legal cases where cutting corners makes life easier. If you smoke roll-ups, cigarette papers with the corners cut are much easier to roll – letting you tuck the paper in more easily. Or more seasonally, wrapping presents is something of an art form. I recall my Father telling me once about some way to calculate the smallest amount of wrapping paper needed, and claiming some chocolate bar manufacturers had saved a fortune using a similar idea. I was too young to pay much attention at the time and can't recall the details now, but it involved cutting corners. Computer graphics also cut corners, tending to rely on representations made from a mesh of triangles, going in straight lines rather than curves. This also means you can do the mathematics once for several vertices and speed things up [Wikipedia-2]. Cutting corners can be a good thing or a bad thing; it depends.

In contrast, cutting the mustard is always good. When mustard was grown as a main crop, it "was cut by hand with scythes, in the same way as corn. The crop could grow up to six feet high and this was very arduous work, requiring extremely sharp tools. When blunt they 'would not cut the mustard'" [Guardian]. Maybe tech debt is like blunt tools? By leaving behind software that's hard to use or difficult to understand or change you've made life difficult. Even a skilled coder won't be able to be very effective if armed with a blunt scythe.

Describing a dangerous or confusing system as debt seems odd. We decided to get some new lighting in our house and the electrician ran a safety test first. We were half expecting a can of worms, or some kind of spaghetti wiring situation. Hooray for a way to test things before touching anything live. I am pleased to report the house doesn't need re-wiring and the electrician's insulated snips worked but we need to have some 'tech debt' fixed before it is safe to get new lights installed. Without going into too many details, let's say words like 'Why would anyone do this?' and 'That's very confusing!' and 'Why would anyone connect the earth wire to live?' were banded about. Similar statements can end up as tech debt Jiras, and the engineers being told the customer's priority is new lights, so these debt tickets will have to wait until later. As a customer, I want it to be safe to change a light bulb, so please fix the dangerous stuff first. Just saying. Letting engineers talk directly to customers is often the best way. There are conventions for which coloured wire connects where for reasons: to make the wiring safe for people to change, replace and extend in the future without a huge wiring diagram and user manual. Describing cutting corners and brazenly ignoring conventions as tech debt seems to miss the point somewhat. Conventions and protocols often exist to keep us safe.

Now, not all coders regard themselves as engineers, and in fact some code isn't written for customers. Many of us have personal projects, and some of us might be regarded as hobbyist programmers. I have frequently sketched out a few lines of code in a new language I'm learning, knowing full well it's an untidy mess, or just for trying things out, like rough notes. I am a beginner so haven't discovered or understood the conventions initially. Does that count as being tech debt? When I first started learning Python, I sketched out lines of code in the repl, and became frustrated at having to type them all over again when I revisited my noodling. Frustration is like tech debt; I learnt to write my code in an actual file I could then save, keep in version control and rerun at will. Amateurs may not be professionals, but they can still cut the mustard. In fact, amateurs code for the love of it. If you code for love rather than money, write in

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.

and tell us. Try looking back over code you wrote a while ago, whether it was for work or pleasure. You will see how you have changed your style and notice better and perhaps safer ways of doing things. It's not that you left yourself a debt that you had to pay back to someone, with interest. It's more that you left your future self a puzzle to solve.

Barney Dellar recently blogged about Escape Rooms, [Dellar20]. A team of people pay money to be locked in a room, and by finding clues and solving puzzles might be able to find a key to get out of the room before their time is up. Barney points out "The way we solve the puzzles now has absolutely no effect on the difficulty of the next puzzles, or the puzzles that we'll face next time we do an Escape Room." In contrast, when we write software, we are creating potential future puzzles. "The faster we go today, the higher the difficulty level will be tomorrow. But if, instead, we go slowly and carefully today, then tomorrow's puzzles will be easy. And easy puzzles don't take long to solve. So we will move faster." Perhaps instead of talking about tech debt, we should talk about hard or easy puzzles. Imagine reporting at your daily stand up that you'd created a puzzling mess that no one could follow because it was quick, and it might not work. The tone is different to saying you've got the code into prod and raised a tech debt ticket to make it neater later.

Steve Freeman has previously used the analogy of unhedged call options to explain tech debt [Linders14]. If you don't know about investment banking, Nat Pryce summarised this as "refactoring now is an investment for the future / a hedge against the callable option I've 'sold' by writing bad code". This may not help, if you don't know what a future, hedge or callable option is. The blog explains in detail, but the high level idea is you agree, for a fee, to sell someone something in the future of a fixed price. The person buying this from you might not take up this option. Neither of you know what the items will sell for at the future date, so this is a bet: will the price got up or down? Without a 'hedge', or some way to ensure you can get hold of the items for a known amount at the date in the future, you could end up in a load of trouble. You will, of course, have the agreed fee up front, but that may be peanuts compared to the amount you could lose. The unhedged call option analogy, regards the fee or premium as a quick win now, which is all very well if you never need to go near the code again. If you do need to go back, you've left an unhedged risk. The trouble with analogies is you need to know about the parallel in order to understand the point being made. A simple way to put this (sorry for that finance pun) is to talk about tech risk rather than tech debt.

Since I've brought economics into the equation, consider John Maynard Keynes' idea of the 'animal spirit', wherein economic decision are often intuitive, emotional and irrational. Others claim the markets are 'rational', the economy flows, and that twisting the right knobs and dials will have a predictable outcome. Now, Keynes is saying confidence or lack of it can drive or hamper economic growth.

> Even apart from the instability due to speculation, there is the instability due to the characteristic of human nature that a large proportion of our positive activities depend on spontaneous optimism rather than mathematical expectations… our decisions to do something positive, the full consequences of which will be drawn out over many days to come, can only be taken as the result of animal spirits—a spontaneous urge to action rather than inaction, and not as the outcome of a weighted average of quantitative benefits multiplied by quantitative probabilities. [Wikipedia-3]

Some take the idea further, and talk about testosterone-fuelled macho nonsense. Whether you think women can be 'hero programmers' or traders, jumping in thoughtlessly and causing instability, or that oestrogen stops such idiocy, unfounded optimism and instability cause trouble. Constrain your animal spirits once in a while. Where does this leave tech debt?

Debt can be paid back at some point. The word covers up for some very challenging financial situations many people find themselves in. Risk, on the other hand, sounds more, well, risky or downright dangerous. Debt has the idea of having borrowed something from someone for a bit, like an 'I owe you' (IOU). The word comes from *debere* 'to owe' or 'keep something away from someone', from de- 'away' (see de-) + habere 'to have' [Wikipedia-4]. What has been taken from whom in tech debt? Sharp tools? Easy to solve puzzles in the future? Maybe. David Graeber's book *Debt, the first 5000 years* regards money as an IOU giving a way to formalize debtors and creditors, and calls into question the idea that debts have to be paid. 'Says who?', basically. Religious texts, well certainly the Old Testament, decries usury and also instigates a Jubilee year "a trumpet-blast of liberty" [Wikipedia-5]. Imagine a clean slate, with all your debts paid off. Michael Feathers recently shared a metaphor for tech debt as running a commercial kitchen, but only cooking, never cleaning anything. A health inspector would shut you down. Software doesn't have health inspectors, but does still need cleaning up for (mental) health reasons. Go one, you owe it to yourself. Tidy your house, fix your wiring, clean up once in a while. Start afresh. Bring on a happy, healthy New Year!

References

[Dellar20] 'Creating our own puzzles', 30 October 2020: https://barneydellar.blogspot.com/2020/10/creating-our-own-puzzles.html

[Guardian] Semantic enigmas: 'What is the origin of the phrase "doesn't cut the mustard"?': https://www.theguardian.com/notesandqueries/query/0,5753,-2242,00.html

[Linders14] Ben Linders (2014) 'Is Unhedged Call Options a Better Metaphor for Bad Code?', posted 24 December 2014 on InfoQ: https://www.infoq.com/news/2014/12/call-options-bad-code/

[Wikipedia-1] Koch snowflakes: https://en.wikipedia.org/wiki/Koch_snowflake

[Wikipedia-2] Triangle mesh: https://en.wikipedia.org/wiki/Triangle_mesh

[Wikipedia-3] Animal spirits (Keynes): https://en.wikipedia.org/wiki/Animal_spirits_(Keynes)

[Wikipedia-4] Debt: https://en.wikipedia.org/wiki/Debt

[Wikipedia-5] Jubilee (biblical): https://en.wikipedia.org/wiki/Jubilee_(biblical)

Questions on the Form of Software

Writing software can be difficult. Lucian Teodorescu considers whether these difficulties are rooted in the essence of development.

2020 is ending. Australian bushfires, Covid-19 pandemic outbreak, Black Lives Matter protests, Beirut explosion, US West Coast wildfires, and a lot of other catastrophic events. Quite a year! At the end of the year, and the beginning of the new year, it's often the time for more reflection, and less action. In this spirit, I will put on hold my intent of writing another article on threading; with all these disasters, the least thing we need is yet another article showing how disastrous are the usual approaches on threading.

Instead, I'll try a reflective, more philosophical article. We will start from Brook's 'No Silver Bullet' article [Brooks86, Brooks95], discuss the essentialist and metaphysical views expressed by the article and consider some questions that may arise from them. I do not have an answer for any of these questions; providing such an answer would probably be one of the biggest advances in Software Engineering. As a software engineer passionate about understanding the essence of things, it is natural for me to ponder over these questions, even if the answers seem far away.

Starting point: essential and accidental software properties

I would argue that 'No Silver Bullet' [Brooks86, Brooks95] is one of the most fundamental articles written in software engineering. It defines the main problems in software engineering, and simultaneously it defines the limits of the field. To prove its point, Brooks makes a metaphysical inquiry in software engineering.

The main conclusion of the article is:

> There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

This statement was first made in 1986, and 9 years later Brooks confirms that it was a true prediction[1]. Although there were some claims that the prediction is not true[2], there is no generally accepted position that there is a *silver bullet* in software engineering from the time that Brooks predicted this.

For the present article, the method through which Brooks arrived at this conclusion is far more important than the conclusion itself. Let's outline his reasoning.

Brooks starts from looking at the *difficulties* of software engineering. Following Aristotle, he divides them into *essential* and *accidental* difficulties[3]. We have *complexity*, *conformity*, *changeability* and *invisibility* as essential difficulties; all the other difficulties (tooling, use of high-level languages, processes, etc.) are accidental.

Here is, for example, a famous passage from the article [Brooks86, Brooks96]:

> Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one, a subroutine, open or closed.

After having a discussion on these four essential difficulties of software, Brooks argues that, due to their nature, one cannot find any single method to considerably improve on these difficulties. He goes over great lengths and show that all the major promises of software engineering attack accidental difficulties and not essential ones[4].

The point that Brooks makes is that if most of the promises are related to accidental difficulties, while the essential difficulties are predominant[5], then there is no way for a promise to deliver more than 10× in terms of productivity, reliability or simplicity. Even if we were to completely eliminate accidental difficulties, we will not be able to achieve one order-of-magnitude improvement.

Like Sisyphus, software engineers are cursed forever to have to struggle with complexity, conformity, changeability. There is no silver bullet and no spell to set them free.

Background

Essentialism in Brooks, essentialism in a software

Merriam-Webster [MW] defines *essentialism* as:

> : a philosophical theory ascribing ultimate reality to essence embodied in a thing perceptible to the senses
>
> : the practice of regarding something (such as a presumed human trait) as having innate existence or universal validity rather than as being a social, ideological, or intellectual construct

Reading *No Silver Bullet* one cannot only but remark the strong *essentialism* present in the article. That is, Brooks believes that there is an *idea* or a *form* of software engineering activities, that is more than just a generalisation of the practices seen so far. The laws of the Universe are made in a way that software engineering is *essentially* difficult.

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. In his spare time, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

1. See the '"No Silver Bullet" Refired' chapter in *The Mythcal Man-Month. Essays on Software Engineering*, Anniverary Edition [Brooks95].
2. For a more recent claim, please see *Yes silver bullet* by Mark Seemann [Seemann19].
3. The distinction between *essential* and *accidental* plays an important role in *Topics* [Aristotle84a] to make the distinction between the definition of a thing and a property of a thing, and in *Metaphysics* [Aristotle84b] to analyse the *essence* of being.
4. Interestingly enough, Object-Orientation is among the promises that Brooks analyses and finds that it cannot achieve much; this happens before OOP became mainstream.
5. Brooks actually clarifies this better in the follow-up '"No Silver Bullet" Refired' chapter of the 1995 edition of *The Mythical Man-Month* book [Brooks95]; he believes that until that date most pressing accidental difficulties are solved, and that the ratio between accidental and essential difficulties cannot be greater than 9 to 1.

for any particular software problem ... we have essential difficulties of the software ... and then we have accidental difficulties

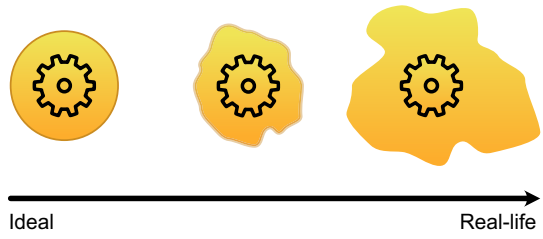


Figure 1

I'm going to take this idea further and apply it to actual software. That is, for any particular software problem that needs to be solved, we have essential difficulties of the software (e.g., there is an essential complexity of the software), and then we have accidental difficulties involved in making that software (e.g., the software took more time to develop, the design is not ideal, we have technical debt, etc.)

Even though Brooks never said this, based on the essentialism emanated from his article, I can perfectly picture him saying:

> Within every software there is a simpler software striving to be free.

In this paradigm, every software has an *essence*, and that would be the *ideal software*, or *the Form of the software*. Now, software is typically more complex that it needs to be, it has conformity problems (i.e., bugs), but also changeability and invisibility issues; that is, essentially is less ideal. We call such a software a *less-ideal software*. In addition to that, software typically has many accidental issues (e.g., tooling-related issues) – this is the *real-life* software. Figure 1 tries to graphically show these three types of software, by adding deformity to an ideal form (circle).

Philosophical views of essentialism

Before going forward with our questions on the *Form* of software, we need to take a very short trip through some philosophical views of essentialism. We'll briefly present the metaphysical views of Plato, Aristotle and Ockham[6]. Although Brooks explicitly mentions Aristotle, we will start from Plato, Aristotle's teacher.

Plato (429?-347 B.C.) is one of the greatest philosophers in the western world, often named the founder of western philosophy [Kenny10][7]. Due to the large variety of topics approached by Plato, Alfred North Whitehead noted [Whitehead78]:

> the safest general characterization of the European philosophical tradition is that it consists of a series of footnotes to Plato

Plato's most important contribution to philosophy is the *theory of Forms*, which is exactly what we are interested in for our article. According to Plato, the reality consists of *Forms* (or *Ideas*) which are outside our material world. The material world consists in shadows (or copies) of these abstract *Forms*. Our senses cannot interact with the world of *Forms* but only with these shadows. The *Forms* are eternal and unchanging, while the things in our material world are always changing. For example, all the humans in the world are just shadows of a *Human Form*; similarly, all the good things in the world are just shadows of the *Goodness Form*; there are *Forms* for shortness, justice, redness, humanness, etc. This view about the existence of *Forms* in an ideal, abstract world is called *Platonic realism*.

So far, we have used the term *Forms* in the context in which we meant abstract ideas; this was done on purpose, to refer to these Platonic *Forms*, the most extreme form of idealism.

Aristotle (384–322 B.C.) was a student of Plato at the Academy. If Plato is credited for founding western philosophy, Aristotle is often credited as being the first scientist, with the systematic way that he approached all the branches of knowledge. He is widely accepted as the inventor of logic and of the deductive reasoning. [Kenny10]

Aristotle was Plato's first true critic. He exposed a very detailed critique of the theory of *Forms*, arguing its inability for expressing change and its inability for explaining how the *Forms* are related with the real-world. Aristotle replaced Plato's extreme idealism, with a *moderate realism*; universal properties exist, but only in as much as they exist in our world.

Being able to distinguish between accidental and essential properties of objects is paramount to being able to identify universal properties in the world. Having a human-like body is an essential property that makes Socrates a human, but the colour of his skin is just an accidental property of Socrates. Also, Aristotle bases a lot of his metaphysics on the notions of potentiality and actuality; with time the set of *humans* change, and we can also think of the *potential humans* and apply the same reasoning to them.

Both Plato and Aristotle argued that there are universal concepts like *Human* (either in an ideal world or in the material world). There are, however, philosophical views that deny the existence of these universals. Probably the best known theory is the Nominalism theory of William of Ockham (1287–1347)[8].

Ockham argues that no universal exists outside our mind; everything in the world is singular. That is, there is no concept of *Human* in this world, or any other world that somehow influences our world. The world contains only human instances, and nothing more; the *Human* concept is just present in our minds.

Let's transpose these three theories in software engineering:

1. Plato would argue that there is a *Form* of Software Engineering in a parallel universe, and all instances of software engineering in our world are imperfect copy of that *Form*.
2. Aristotle would argue that there is a *universal* called *Software Engineering* that can be analysed by properly looking at its essential properties (and ignore accidental properties).

6. In Latin, Ockham is spelled Occam.
7. Plato also created the Academy, which is considered the first higher learning institution in the western world.
8. The famous Occam's razor is most of the time attributed to Ockham; the principle says that "*entities should not be multiplied without necessity*", and sometimes it's paraphrased as "*the simplest explanation is most likely the right one*".

All the improvements we can hope for in software engineering will then come from our ability to improve how our mind looks at these problems

3. Ockham would argue that there is no such thing as software engineering; there are just many activities that have nothing truly in common, and it is only in our minds that we call them software engineering.

Brooks seems to share the same views as Aristotle.

With these three views covered, we are now finally ready to reach the questions, the essentials of this article.

Q1: Does a Form of a software exist?

Let us take as an example a text editor software (with a known set of requirements). We call this *TextEditorSw*. The question then becomes whether a *Form* of *TextEditorSw* exists, and how real is this.

According to the three philosophical views, we have three potential answers:

1. Yes, there is a *Form* of *TextEditorSw* in an ideal world, and all our real instances of text editors are just imperfect copies of it.
2. Yes, there is a universal for *TextEditorSw*, which is present (at least in potentiality) in this world; we just have to enumerate the essential characteristics of this universal to better understand it.
3. No, there is no such universal; everything is in our head.

If the answer is one of the first two, then there is an 'essence' of *TextEditorSw*. Furthermore, it probably makes sense to assume that this 'essence' will be the least complex, the one that is the most possible conformant, with no changeability and no visibility issues. That is, the software would be perfect.

If we can identify the perfect software (at least for a particular type of software), then that would be the *Holy Grail* of software engineering. Trying to understand this perfect software, and the means that we need to take to reach it would be the most important activities of software engineering.

If we would have a clear proof (or at least a strong enough argument) that this perfect software exists, I can imagine a lot of research being founded to get us closer to this perfect software. We can probably start developing a method/program to dramatically improve software engineering, similar to Hilbert's program in mathematics[9]. This ultimately would lead us to the *silver bullet* of software engineering.

If the answer is *yes* with an extreme idealist form (Plato's theory of *Forms*) it means that the perfect software is somewhere inaccessible to us, but we can get near it by pure reasoning. One can, theoretically, just by thinking, arrive close enough to the *Form* of the perfect software. In the process it might uncover other *Forms* of what a good software means, and thus, we can improve our software engineering practices.

If we take an Aristotelian view, then the perfect software might exist in the real world (albeit it can be hard to define); if it does not actually exist, then it is certainly possible for it to exist. We can certainly use empirical inquiries to find out more about the properties of the software, and thus to continuously improve our software engineering practices. This time, the attack on software engineering difficulties must be targeting only essential difficulties.

On the other hand, if we believe that a nominalism view would be more appropriate, and we conclude that the third answer is the right one, then we would be hopeless in trying to identify what good software engineering means. All our systematic approaches in improving software engineering will inevitably fail. We can never know what makes a software a good software; we just have different instances of software, and all the generalisations are just in our head. All the improvements we can hope for in software engineering will then come from our ability to improve how our mind looks at these problems.

Note that there are multiple other nuances between these three possible answers. I'm just trying to lay out the main answers according to the main theories related to the presence of universals.

Q2: How close can we get to the Form of a software?

Provided that we exclude the nominalist view, and we have enough confidence that there is a (almost) perfect *Form* of the software (either in this world or outside it), can we go near it? Or, lowering the bar a bit, can we go to something like 10% close to it?

Now, for the sake of argument, let's assume that if we get close enough to that software, we get 90% of the benefits of having a perfect software, and we just have something like 10% of difficulties. If that would be the case, then, this again would probably count as the silver bullet. We eliminated most of the complaints that Brooks had on why software engineering is difficult.

More than that, we have a model to study on what the almost-perfect software would look like. If we know how an almost-perfect *TextEditorSw* would look like, then we can probably generalise this knowledge to other types of text editors. And, then, even further, to other types of software products.

Continuing this line of thought we would probably end up with a systematic way of ensuring that we 'properly' build software, that ensures that we get very close to ideal software.

It is almost like we gain the same level of trust that bridge building has from structural engineering. Software engineering would not be such a costly activity anymore, as we would know how to get around most of the difficulties. We can probably also automate it in a large proportion (i.e., robots writing code).

This question is far too important to be left unanswered, or at least to be left without any attempts to be answered.

Q3: Do we have a way to measure the distance to the Form of a software?

This questions is mainly a continuation of the previous question. We assume that there is a *Form* of software, and we may or may not know how to get to it.

9. Hilbert proposed a program to ensure the existence of a method to prove all propositions in mathematics [Wikipedia]; Göedel showed later that such a program is unattainable.

In the best case we can have a test that would give us a numeric distance between a particular software and its ideal form. But, if that is not possible, a test that would tell them whether the two are close enough is also a significant addition to our toolset. This would be similar to the inability of finding a numeric measurement to indicate the 'distance' between an object and a chair, but we can easily test if the given object is a chair or not.

If we would have a numeric measurement, then arriving to (or close to) the ideal form of a software is relatively easy. If we don't have a systematic way, then we can apply some form of learning (similar to how we do it in Machine Learning) to arrive close to the ideal *Form*.

If we would only have a binary test, then we can approach this problem the brute-force way. Doing this enough times, for different types of problems, would probably lead us to some conclusions, and thus we can learn by repeated experience how to design the perfect software (I'm assuming that there are some universal laws that can be approximated, and the perfect software is not just subjected to randomness).

Thus, if we have such a measurement, we can empirically derive an algorithm to get our software to the perfect state, and, moreover, we can hope that we can learn the best way to do this for all software problems.

Q4: Can we distinguish between accidental and essential complexity?

One thing that puzzles me in Brooks's description is that it is not apparent what essential complexity is. On one hand, Brooks puts complexity as a whole as an essential difficulty to software engineering, but then he goes on and argues that the use of higher-level programming languages are just covering for accidental difficulties.

Let us consider an algorithm over a collection of data that can be written in 10 lines of a high-level programming language (either OO or functional). Let's assume that this algorithm does some sorting, some mapping and some filtering, etc. If we were to translate this into assembly it would probably far too big for a person to understand it in a day, not to mention writing it. So, there are at least a couple of orders of magnitude between the time spent in understanding the high-level code and understanding the assembly one (not to mention that assembly can be considered high-level compared to the processes that happen at the hardware level). Do we really think that understanding assembly is just accidental difficulty? Probably not.

On the other hand, one can write the same problem in multiple ways using a high-level language. And most of the variants are having roughly the same complexity. For example, breaking a private 10 lines of code function into two separate function is most probably not changing the complexity of the solution. This surely cannot be considered as being essential traits for the software.

So then, what is essential, and what is accidental? Although we do have the tools to properly make this distinction, I don't think we know how to do it.

If we would answer this question, then probably we could easily derive reasoning that would tell us what a good software is, and what is not.

Q5: Is technical debt an essential difficulty?

Most often when software developers discuss difficulties in their day to day work, the phrase technical debt pops up. Symptoms of technical debt include: large number of bugs in a particular area, ad hoc design, difficulty of changing the software, lack of or inappropriate documentation, lack of testing, etc.

There are cases in which technical debt can be a good thing for the project lifetime (i.e., for proof-of-concepts, for features that need to make a deadline, for delaying some effort that are not sure that are really needed, etc.), but most of the time technical debt is considered a bad thing; thus a difficulty in the sense we are discussing here.

The main question that arises then is whether technical debt is an essential difficulty or not.

On one hand, one can argue that it is one of the major sources of complexity in a project, and as complexity is an essential difficulty of software engineering, so technical debt is an essential difficulty. On the other hand, if some technical debt can be good for the project, then it cannot be an essential difficulty.

But maybe, technical debt is somewhere in the middle. Sometimes, over certain limits it is an essential difficulty, and sometimes, when kept under control, it is not an essential difficulty (either an accidental difficulty or a good thing).

If we can find out the limits after which technical debt is an essential difficulty, then maybe we can put processes in place to prevent it from crossing those limits.

It's essential to understand essential difficulties (pun intended).

Final words

In our post-modern world, people are always looking for solutions to problems. Sometimes we even invent solutions to problems that we never had. But, if it is said that asking questions is more important than answering them, perhaps the best solutions can be found only after trying hard to ask the right set of questions.

This is my attempt to ask some questions that I feel are important for the field of software engineering. Maybe they cannot be properly answered, maybe they are not the most important questions that we have to ask, but it's clear that they gravitate around the essence of software engineering (if there is an essence to it).

I argue that answering any of these questions will be a significant step forward in software engineering. Probably it would be the mythical silver bullet. But, if it's not, it would most likely lead to alleviating some difficulties in our field.

If these cannot be properly answered, a good answer approximation would probably still advance the fundamental research in software engineering.

If that doesn't happen either, then I hope at least I was able to detract the reader from this cruel reality, and make them enjoy this short detour in philosophising about software engineering. ■

References

[Aristotle84a] Aristotle, 'Topics' in *The Complete Works of Aristotle, Volume 1: The Revised Oxford Translation* (vol 1), edited by Jonathan Barnes, Princeton University Press, 1984

[Aristotle84b] Aristotle, 'Metaphysics' in *The Complete Works of Aristotle, Volume 2: The Revised Oxford Translation*, edited by Jonathan Barnes, Princeton University Press, 1984

[Brooks86] Frederick P. Brooks Jr., 'No Silver Bullet – Essence and Accidents of Software Engineering', *Proceedings of the IFIP Tenth World Computing Conference*, edited by H.-J. Kugler, 1986

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

[Kenny10] Anthony Kenny, *A New History of Western Philosophy In Four Parts*, Clarendon Press, Oxford, 2010

[MW] Merriam-Webster, 'essentialism', accessed Oct 2020, https://www.merriam-webster.com/dictionary/essentialism

[Seemann19] Mark Seemann, 'Yes silver bullet', 2019, https://blog.ploeh.dk/2019/07/01/yes-silver-bullet/

[Wikipedia] *Wikipedia The Free Encyclopedia*, Hilbert's program, accessed Oct 2020, https://en.wikipedia.org/wiki/Hilbert%27s_program

[Whitehead78] Alfred North Whitehead, *Process and Reality*, New York: The Free Press, 1978.

Building g++ From the GCC Modules Branch

Using the WSL to build the g++ modules branch. Roger Orr demonstrates how to get a compiler that supports modules up and running.

The last issue of *Overload* contained Nathan Sidwell's article 'C++ Modules: A Brief Tour' where he provided some short examples of C++20 modules in action. The 'Implementation' box in the article showed the status of four compilers, including Nathan's own branch of gcc. In the conclusion of the article he wrote: "Unfortunately, for GCC one must use Godbolt, which is awkward for the more advanced use, or build one's own compiler, which is a steep cliff to climb for most users."

I thought a worked example of building g++ from the modules branch from scratch might be helpful for people who are keen to experiment further with gcc's implementation of C++ modules but are intimidated by the thought of building the compiler for the first time.

Getting started

The Gnu Compiler Collection (gcc) can be built on a very wide range of systems. The overall process is much the same, but there will be various differences depending on the exact target. One main difference between systems is the mechanism you need to use to obtain the various pre-requisites that building gcc requires.

I have no statistics on which operating systems the readership of *Overload* use and so I have chosen to build the compiler on **Windows 10** using Ubuntu running in the 'Windows Subsystem for Linux'. This should be useful both to those with Windows machines and to those with Ubuntu running natively.

Other alternatives on Windows are possible; you can for example build the compiler using Cygwin.

On other Linux distributions the process will be similar, but the actual commands used to download the other tools will depend on the package management system they use. On Ubuntu the downloads can be performed using APT (which in this context is the acronym for the 'Advanced Package Tool' rather than for an 'Advanced Persistent Threat'…!)

One of the things that makes building gcc quite painful, in my experience, is that analysing any build errors is complicated by the number of lines of output the build produces. In particular, I spent quite a bit of time when I first built gcc tracking down missing dependencies since, especially as a newcomer, the symptoms do not necessarily directly indicate the root cause. For example, it is worth checking specifically for the string 'missing' early in the logs if you get a build failure to see if it may be caused by a dependency you are lacking.

Installing WSL

If you have not previously installed this feature, it is quite straightforward to get started with it, at least on moderately recent versions of Windows 10. Open the control panel, click on *Programs and Features* and in the resulting dialog box, click on *Turn Windows features on or off*. Enable *Windows Subsystem for Linux* and click *Ok*.

The computer needs to reboot to install the additional feature, and when this has completed you should visit the Microsoft Store and select an appropriate Linux installation: I simply selected *Ubuntu* which, at the time of writing, installed version 20.04 LTS (Long Term Support). The length of time this takes will depend on your download speed – it's a touch under 500 Mb. After this has completed you should now have an 'Ubuntu' icon in your start menu.

The first time you run this you will need to enter the username and password for the primary account (which will be granted `sudo` permissions, which you will need to install the prerequisites). I suggest the first two commands you run are `sudo apt update` and `sudo apt upgrade` which ensures your base operating system is up-to-date.

Note: the installation of WSL on earlier versions of Windows 10 required enabling developer mode, which is not the case on the current release. Additionally, there is now support for both 'version 1' and 'version 2' of WSL. Interesting as this might be, it is orthogonal to the primary purpose of *this* article, which is focussed on building the g++ compiler.

Getting dependencies

As mentioned above, the build of gcc makes use of a number of other tools, some of which will be installed in the base Linux installation but some of which may need to be installed manually.

Since in this case I am looking to build and use a branch *of* gcc, rather than being a developer *of* gcc, I can save some time by avoiding the 'bootstrap' part of building gcc and use a mainstream version of gcc to compile the modules branch.

For building gcc I needed to ensure the following components were present:

bison, **flex**, **git**, **g++**, and **make**

On Ubuntu this can be achieved with one command:

```
sudo apt install bison flex git g++ make
```

On some installations you also need to install **m4** and **perl**, but they're part of the base install of Ubuntu. The build also uses **makeinfo** to create **info** files for the compiler. I was not particularly interested in the info files, so didn't bother to install **makeinfo**, but if you do want those files then you also need to install the **texinfo** package which provides **makeinfo**.

Checking a base build of g++

Once these dependencies are installed, you can test your setup by building the main trunk of g++. Having had various issues building g++ on Windows machines caused by the Windows default line ending (carriage return and line feed), I err on the side of caution by specifying explicit options to git to ensure that a single line feed is used.

The base build splits into two parts, firstly **downloading** the source files and some other dependencies:

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

One of the things that makes building gcc quite painful is that analysing any build errors is complicated by the number of lines of output the build produces

```
mkdir ~/projects
cd ~/projects
git clone -c core.eoln=lf -c core.autocrlf=false\
git://gcc.gnu.org/git/gcc.git
cd gcc
./contrib/download_prerequisites
```

and then **building** the compiler:

```
mkdir ../build
cd ../build
../gcc-trunk/configure --disable-bootstrap
--disable-multilib\
--enable-languages=c++ --enable-threads=posix
make -j 4
```

If all goes well this will (eventually!) build a version of the latest trunk code for the g++ compiler.

A few notes on the build commands.

- It's best to build in a directory outside the source tree: here I use a sibling directory
- `disable-bootstrap` as this avoids 'bootstrapping' the build process (by using a reasonably up-to-date g++ compiler to kickstart the build) which makes the build significantly quicker
- `disable-multilib` as I only want the 64-bit compiler, without this option I'll get the 32-bit compiler too (and will also need some additional prerequisites)
- `enable-languages=c++` as I just want to try out the c++ modules support
- `enable-threads=posix` so I can use threads in my C++ programs

Building the modules branch

Once you have got this far, building the modules branch itself should be relatively straightforward. When I originally built the modules branch after the article was published, there was an additional prerequisite (zsh) and you also needed to download and build the libcody library separately; but this was recently included as a subproject in the modules branch and so the build process is now simpler.

Simply check out the right branch:

```
cd ../gcc
git checkout devel/c++-modules
```

and then build and install this version

```
mkdir ../build-modules
cd ../build-modules
../gcc-modules/configure --disable-bootstrap
 --disable-multilib --enable-languages=c++
 --enable-threads=posix
 --prefix=/usr/share/gcc-modules
make -j 4
sudo make install
```

Note that I am providing a specific target directory for the installation with `--prefix` as I don't want the modules branch build to other builds of gcc that I have installed. This does mean I will need to select this version explicitly; for example by giving the full path to g++ or by pre-pending the directory containing g++ to the `PATH` environment variable.

Kicking the tyres

Now we should be able to build the first example from Nathan's article:

```
cd ~/projects/example1
export PATH=/usr/share/gcc-modules;%PATH%

g++ -fmodules-ts -std=c++20 -c hello.cc
g++ -fmodules-ts -std=c++20 -c main.cc
g++ -o main main.o hello.o
./main
Hello World
```

Success!

Updating the build

If you want to rebuild the compiler to pick up later changes to the modules branch there are a couple of things to bear in mind.

Firstly, the `./contrib/download_prerequisites` command adds some directories and symlinks to the source tree. You don't usually need to run this again; but if the versions of the prerequisites **change** (as they sometimes do) it is important to remove the old versions before running the command. (My own scripts simply delete any existing artefacts and unconditionally download each time I do a build of gcc.)

Secondly, I recommend deleting the contents of the build directory before re-compiling. While in theory the timestamp-based dependency algorithm used by `make` should handle changes smoothly, this has not always been my actual experience and the resultant build issues took me longer to resolve than any time saved by performing an incremental build.

So my instructions for a full refresh are:

```
cd gcc-modules
git pull --ff-only
rm -f gmp* mpfr* mpc* isl*
./contrib/download_prerequisites
rm rf ../build-modules
```

and then build and install as before.

Conclusion

Building gcc can seem difficult, but I hope this worked example encourages some of you to try it for yourself and thereby be able to further explore the gcc modules implementation that Nathan's article made reference to. ■

Consuming the uk-covid19 API

Covid-19 data is available in many places. Donald Hernik demonstrates how to wrangle data out of the UK API.

WARNING: This article is written in an unnecessarily cheerful tone ("Ah! So you're a waffle man!" [Red Dwarf] as an antidote to the subject matter and the current state of the world. Stay safe, everybody.

Please note: This article was written in October 2020 and the *Developers' Guide* document referenced below has been updated many times since.

Introduction

I don't think I've seen so many charts in the press since the happy days of the Brexit referendum or, perhaps, the Credit Crunch. Say what you like about Coronavirus but if you like charts then this is a fantastic time to be alive...

I am not a data scientist but I wondered – could I get the underlying data and plot my own charts?

Good news, yes! But there were some problems along the way.

Public Health England (PHE) Data

Public Health England publish the UK Covid data and sites exist to view the various charts [GOV.UK-1].

The data are also published via an endpoint:
https://api.coronavirus.data.gov.uk/v1/data

- There is a Developers' Guide [GOV.UK-2] (henceforth referred to as DG) for consuming this. The DG tells you how to structure requests, what metrics are supported, error codes, etc.
- The list of metrics that can be requested is (as documented in the DG) regularly updated so there may be more metrics to request next week than this.
- Separately there is a wrapper SDK (uk-covid19) which simplifies using the endpoint. There is separate documentation for this [PHE] but reading the DG is still very useful.

The uk-covid19 SDK API

In summary:

- The SDK is provided for Python, JavaScript, and R.
- Requests are input as JSON.
- Response data can be extracted as JSON or XML.
- Without the SDK, requests can be made directly to the endpoint above via e.g. the Python HTTP *requests*. The SDK libraries wrapper useful behaviour such as processing multiple 'pages' of data in the response. It also swallows some error cases – see below.

Donald Hernik has a BSc in Information Systems and has been a software developer for over twenty years, predominantly using C++, and most recently in Financial Services. He is currently looking for an interesting, fully remote, job. He can be contacted at donaldhernik@hx1technology.co.uk

The Python implementation

I am not a Python developer (see also 'data scientist', above) having only really used it for build scripts and log scrapers but this was an interesting opportunity to learn something new, and Python has a well-earned reputation for developing things quickly and simply.

The Python SDK requires Python 3.7+ so I installed Anaconda 3.8. The SDK module is installed via PIP.

```
pip install uk-covid19
```

Making requests

Please note that (through nobody's fault) the formatting of the listings has suffered slightly for publication. You'll just have to trust me that it's valid Python.

WITHOUT using the API

Making a request *without* using the API is simple enough – see Listing 1 – however:

> **NOTE1:** Quiz – does the `get` method get all of the pages of the response? The API requests multiple pages in a loop until the response is `HTTPStatus.NO_CONTENT`...
>
> **NOTE2:** We can handle all the HTTP status codes, especially 204 (Success – no data).

WITH the uk-covid19 API

Making a request using the API is simple enough – see Listing 2 – however:

> **NOTE3:** Can we detect that a 204 (Success – no data) response happened? No. The API throws an exception only for HTTP error codes >= 400.

API Pitfalls

Some problems that I encountered along the way.

The 204 response

As documented in the DG, HTTP response 204 is 'Success – no data' and the response JSON looks like this.

```
{'data': [], 'lastUpdate': '2020-10-30T15:31:25.0
00000Z', 'length': 0, 'totalPages': 0}
```

Unfortunately, via the API, you can't tell what the HTTP status code was (unless it's >= 400, in which case an exception is thrown).

Where is my data (part 1)?

Surely there is data for 'Englund'? Why is my response empty?

If you e.g. misspell an `areaName` then the server responds with a "204 OK" response. The API swallows the status code so we can't tell if there is genuinely no data or a typo in our request.

This is why we, as good programmers, always validate our input.

```
import requests

def main():
  """Get the Covid data via the endpoint"""
  try:
    area_name = 'suffolk'
    area_type = 'utla'
    url = 'https://api.coronavirus.data.gov.uk/v1/
data?'
    filters =
f'filters=areaType={area_type};areaName={area_name
}&'
    struc = 'structure={"date":"date",
    "newAdmissions":"newAdmissions",
    "cumAdmissions":"cumAdmissions",
    "newCasesByPublishDate":
      "newCasesByPublishDate:}'
    endpoint = url + filters + struc
    # NOTE 1: Does this get all of the data?
    # Or just the first page?
    response = requests.get(endpoint, 30)
    if response.status_code == 200:
      # OK
      data = response.json()
      print(data)
    else:
      if 204 == response.status_code:
        # NOTE 2: This explicitly warns if no
        # data is returned.
        print(f'WARNING: url [{url}], status_code
        [{response.status_code}], response
        [Success - no data]')
      else:
        print(f'ERROR: url [{url}], status_code
        [{response.status_code}], response
        [{response.text}]')
    except Exception as ex: # pylint:
    disable=broad-except
      print(f'Exception [{ex}]')
if __name__ == "__main__":
  main()
```

Listing 1

```
from uk_covid19 import Cov19API
def main():
  """Get the Covid data via the API"""
  try:
    area_name = 'suffolk'
    area_type = 'utla'

    # The location for which we want data.
    location_filter = [f'areaType={area_type}',
    f'areaName={area_name}']

    # The metric(s) to request. NOTE: More than in
    # the previous example, for variety.
    req_structure = {
    "date": "date",
    "areaCode": "areaCode",
    "newCasesByPublishDate":
"newCasesByPublishDate",
    "newCasesBySpecimenDate":
"newCasesBySpecimenDate",
    "newDeaths28DaysByDeathDate":
"newDeaths28DaysByDeathDate",
    "newDeaths28DaysByPublishDate":
"newDeaths28DaysByPublishDate"
    }

    # Request the data.
   # This gets all pages and we don't need to care
how.
    api = Cov19API(filters=location_filter,
structure=req_structure)
    # Get the data.
    # NOTE3: If a 204 (Success - no data) occurs
can we tell?
    data = api.get_json()
    print(data)
  except Exception as ex: # pylint: disable=broad-
except
    print(f'Exception [{ex}]')
if __name__ == "__main__":
  main()
```

Listing 2

Where is my data (part 2)?

There are multiple **areaType** values (briefly documented in the DG). I've never worked in healthcare or the public sector (see also 'Python developer' and 'data scientist', above) so some of these are new to me. The non-obvious **areaType** values are:

- **nhsRegion** – how and why is this different to **region** (e.g. 'Yorkshire and the Humber')?

  What are the valid values? I haven't had time to find out as I stuck to obvious **areaType**s – **nation** etc.

- **utla** v **ltla** – Upper Tier v Lower Tier Local Authorities.

  Some values e.g. 'Leeds' are both a UTLA and an LTLA, and some are not.

  Suffolk (UTLA) for example is composed of 'Babergh', 'Ipswich', 'South Suffolk', 'Mid Suffolk', and 'West Suffolk' (each an LTLA).

If you *mismatch* a *valid* **areaName** and a *valid* **areaType** in your request then you can get a 204. For example: e.g.

areaName	areaType	HTTP response status
Leeds	ltla	200 – OK
Leeds	utla	200 – OK
Suffolk	ltla	204 – OK // No data
Suffolk	utla	200 – OK

This makes sense, but more input validation required.

Where is my data (part 3)?

Occasionally, especially while coding on Saturdays, I encountered error code 500 'An internal error occurred whilst processing your request, please try again' responses even for my perfectly crafted requests.

I tried again later – there was data.

Where is my data (part 4)?

As documented in the *About the data* guide [GOV.UK-3] there are sensible caveats about data correctness and availability.

- Sometimes data is simply not available for all areas for a given date. It is common (and by design) that for some requested metrics the response value is **None** (data missing) which is different to a response value of zero (data present, and zero).
- Sometimes data is retrospectively corrected/added so be careful if you're going to e.g. cache it by date. Data that is not there today for day *T-n* might one day be added (or might not).
- The broader the **areaType** (e.g. **nation**) the more metrics are populated.

  For example, **hospitalCases**, **covidOccupiedMVBeds**, **maleCases**, and **femaleCases** are populated for England (on dates that values are available) but are **never** (to date) populated at the LTLA or UTLA level.
- The only data consistently populated to date for UTLA and LTLA **areaType**s are various cases and death metrics (**newCases**…, **newDeaths**…, **cumDeaths**…, etc). This may change in the future.

```
{
  "date": "2020-10-29",
  "hospitalCases": 8681,
  "newAdmissions": null,
  "cumAdmissions": null,
  "covidOccupiedMVBeds": 803,
  "newCasesByPublishDate": 19740,
  "newCasesBySpecimenDate": 726,
  "cumDeaths28DaysByDeathDate": 40854,
  "newDeaths28DaysByDeathDate": 61,
  "cumDeaths28DaysByPublishDate": 40628,
  "newDeaths28DaysByPublishDate": 214
}
```

Listing 3

- For cumulative metrics (e.g. *cumAdmissions*) the value is only populated on dates it changes e.g. on date *T* `cumAdmissions` may be 9999 and on date *T*+1 it may be `None`.

If you inspect the response JSON as you develop, you will spot this and anticipate `None` values.

Processing the data

Data

Once your request is perfected, you'll get some nice, shiny, data. This example is from `areaType=nation`, `areaName=England`. Only one date is shown here but there are multiple dates in the JSON and data back to 2020-01-03. See Listing 3.

NOTE: The `null` values are a side effect of saving the data to file. In the Python app they are `None`.

Plotting a chart

This article would be too long ("So you're a waffle man!") if I delved into plotting charts. Suffice to say that I had a poke around on Stackoverflow [Stackoverflow] and discovered matplotlib [Matplotlib]. One tutorial later (I don't remember which – sorry) and I churned out a chart of my own. There was much rejoicing. Sadly, the chart showed that hospital admissions and mechanical ventilated bed occupancy were increasing, so the rejoicing was reined in somewhat.

Conclusion

- The uk-covid19 SDK is easy to use and the data can be used to plot your own charts – mission accomplished!
- The data comes with documented caveats to which you should pay close attention.
- Not all metrics are available for all `areaType`s.
- Watch out for HTTP code 204 and other pitfalls. ■

References

[GOV.UK-1] Daily Summary: https://coronavirus-staging.data.gov.uk/

[GOV.UK-2] Developers' Guide: https://coronavirus.data.gov.uk/developers-guide

[GOV.UK-3] About the Data: https://coronavirus.data.gov.uk/about-data

[Matplotlib] https://matplotlib.org/3.1.1/index.html

[Red Dwarf] Talkie Toaster: https://reddwarf.fandom.com/wiki/Talkie_Toaster

[PHE] Python SDK Guide: https://publichealthengland.github.io/coronavirus-dashboardapi-python-sdk/pages/getting_started.html#

[Stackoverflow] Stackoverflow: https://stackoverflow.com/

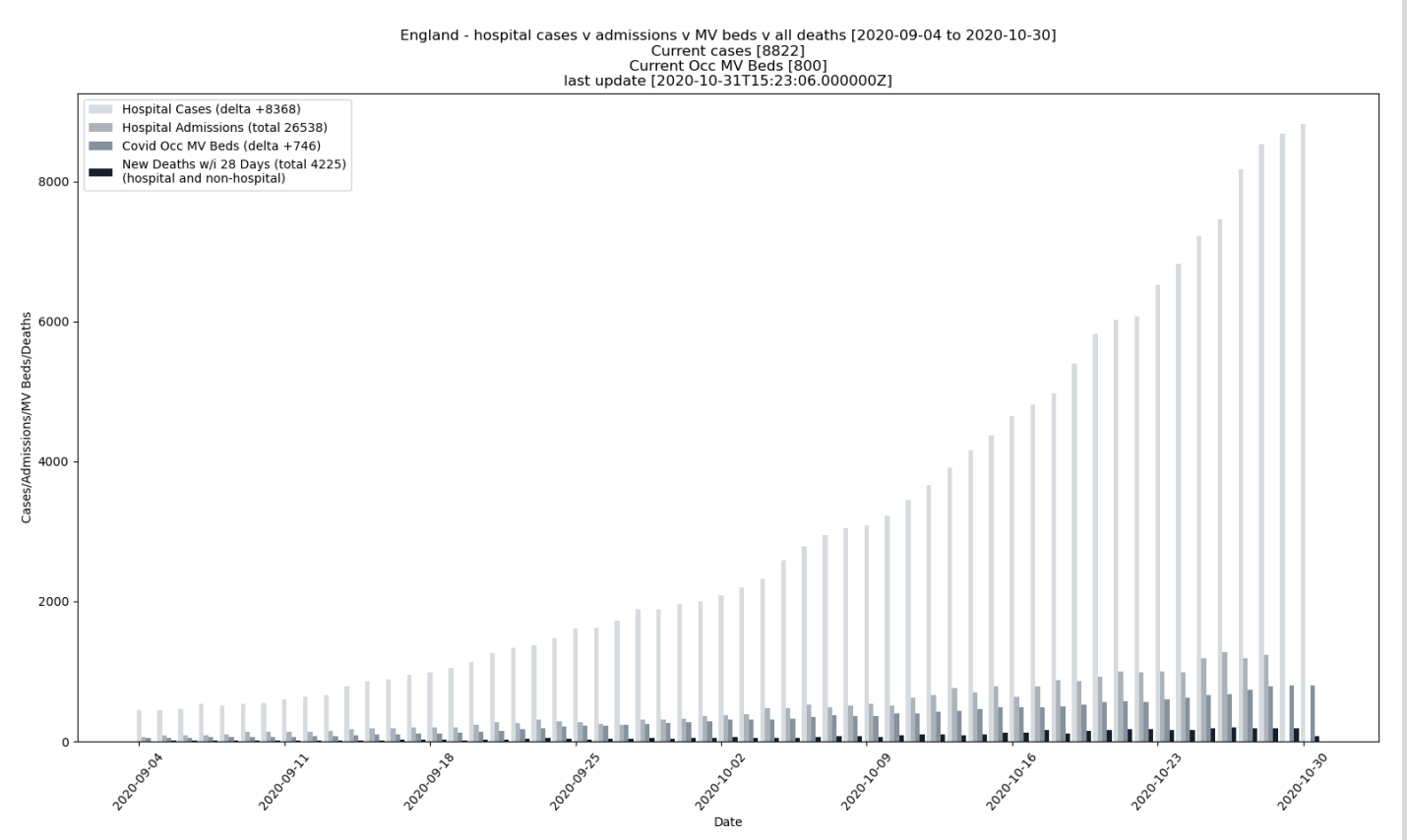


Figure 1

What is the Strict Aliasing Rule and Why Do We Care?

Type Punning, Undefined Behavior and Alignment, Oh My! Strict aliasing is explained.

What is strict aliasing? First we will describe what is aliasing and then we can learn what being strict about it means.

In C and C++, aliasing has to do with what expression types we are allowed to access stored values through. In both C and C++, the standard specifies which expression types are allowed to alias which types. The compiler and optimizer are allowed to assume we follow the aliasing rules strictly, hence the term strict aliasing rule. If we attempt to access a value using a type not allowed it is classified as undefined behavior (UB) [CPP-1]. Once we have undefined behavior, all bets are off. The results of our program are no longer reliable.

Unfortunately, with strict aliasing violations we will often obtain the results we expect, leaving the possibility the a future version of a compiler with a new optimization will break code we thought was valid. This is undesirable and it is a worthwhile goal to understand the strict aliasing rules and how to avoid violating them.

To understand more about why we care, we will discuss issues that come up when violating strict aliasing rules, type punning since common techniques used in type punning often violate strict aliasing rules and how to type pun correctly, along with some possible help from C++20 to make type punning simpler and less error prone. We will wrap up the discussion by going over some methods for catching strict aliasing violations.

Preliminary examples

Let's look at some examples, then we can talk about exactly what the standard(s) say, examine some further examples and then see how to avoid strict aliasing and catch violations we missed. Here is an example that should not be surprising:

```
int x = 10;
int *ip = &x;

std::cout << *ip << "\n";
*ip = 12;
std::cout << x << "\n";
```

We have an `int*` pointing to memory occupied by an `int` and this is a valid aliasing. The optimizer must assume that assignments through `ip` could update the value occupied by `x`. Listing 1 shows aliasing that leads to undefined behavior.

In the function `foo` we take an `int*` and a `float*`, in this example we call `foo` and set both parameters to point to the same memory location, which in this example contains an `int`. Note, the `reinterpret_cast` [CPP-2] is telling the compiler to treat the expression as if it had the type specified by its template parameter. In this case, we are telling it to treat the expression `&x` as if it had type `float*`. We may naively expect the result of the second `cout` to be `0` but with optimization enabled using `-O2` both gcc and clang produce the following result:

```
0
1
```

which may not be expected but is perfectly valid, since we have invoked undefined behavior. A `float` can not validly alias an `int` object.

```
int foo( float *f, int *i ) {
  *i = 1;
  *f = 0.f;
  return *i;
}
int main() {
  int x = 0;
  std::cout << x << "\n";   // Expect 0
  x = foo(reinterpret_cast<float*>(&x), &x);
  std::cout << x << "\n";   // Expect 0?
}
```

Listing 1

Therefore the optimizer can assume the `constant 1` stored when dereferencing `i` will be the return value since a store through `f` could not validly affect an `int` object. Plugging the code in Compiler Explorer shows this is exactly what is happening:

```
foo(float*, int*): # @foo(float*, int*)
mov dword ptr [rsi], 1
mov dword ptr [rdi], 0
mov eax, 1
ret
```

The optimizer using Type-Based Alias Analysis (TBAA)[1] assumes `1` will be returned and directly moves the constant value into register `eax` which carries the return value. TBAA uses the languages rules about what types are allowed to alias to optimize loads and stores. In this case, TBAA knows that a `float` can not alias an `int` and optimizes away the load of `i`.

Now, to the Rule-Book

What exactly does the standard say we are allowed and not allowed to do? The standard language is not straightforward, so for each item I will try to provide code examples that demonstrate the meaning.

What does the C11 standard say?

The C11 standard[2] says the following in section *6.5 Expressions paragraph 7*:

> An object shall have its stored value accessed only by an lvalue expression[3] that has one of the following types:[88)]
>
> ■ a type compatible with the effective type of the object,

Anonymous An anonymous contribution.

1. Type-Based Alias Analysis: https://www.drdobbs.com/cpp/type-based-alias-analysis/184404273
2. Draft C11 standard is freely available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
3. Understanding lvalues and rvalues in C and C++: https://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c

In C and C++, aliasing has to do with what expression types we are allowed to access stored values through

```
int x = 1;
int *p = &x;
printf("%d\n", *p); // *p gives us an lvalue
// expression of type int which is compatible
// with int
```

- a qualified version of a type compatible with the effective type of the object,

```
int x = 1;
const int *p = &x;
printf("%d\n", *p); // *p gives us an lvalue
// expression of type const int which is
// compatible with int
```

- a type that is the signed or unsigned type corresponding to the effective type of the object,

```
int x = 1;
unsigned int *p = (unsigned int*)&x;
printf("%u\n", *p ); // *p gives us an lvalue
// expression of type unsigned int which
// corresponds to the effective type of the
// object
```

Note: There is a gcc/clang extension[4] that allows assigning `unsigned int*` to `int*` even though they are not compatible types.

- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

```
int x = 1;
const unsigned int *p =
  (const unsigned int*)&x;
printf("%u\n", *p ); // *p gives us an lvalue
// expression of type const unsigned int which
// is a unsigned type that corresponds with to
// a qualified version of the effective type of
// the object
```

- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a sub-aggregate or contained union), or

```
struct foo {
 int x;
};
void foobar( struct foo *fp, int *ip );
// struct foo is an aggregate that includes
// int among its members so it can alias with
// *ip
foo f;
foobar( &f, &f.x );
```

- a character type.

```
int x = 65;
char *p = (char *)&x;
printf("%c\n", *p ); // *p gives us an lvalue
// expression of type char which is a
// character type. The results are not
// portable due to endianness issues.
```

What does the C++17 Draft Standard say

The C++17 draft standard[5] in section *[basic.lval] paragraph 11* says:

> If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:63 (11.1) — the dynamic type of the object,
>
> ```
> void *p = malloc( sizeof(int) ); // We have
> // allocated storage but not started the
> // lifetime of an object
> int *ip = new (p) int{0}; // Placement new
> // changes the dynamic type of the object to int
> std::cout << *ip << "\n"; // *ip gives us a
> // glvalue expression of type int which matches
> // the dynamic type of the allocated object
> ```
>
> (11.2) – a cv-qualified version of the dynamic type of the object,
>
> ```
> int x = 1;
> const int *cip = &x;
> std::cout << *cip << "\n"; // *cip gives us a
> // glvalue expression of type const int which
> // is a cv-qualified version of the dynamic
> // type of x
> ```
>
> (11.3) – a type similar (as defined in 7.5) to the dynamic type of the object,
>
> ```
> int *a[3];
> const int *const *p = a;
> const int *q = p[1]; // ok, read of 'int*'
> // through lvalue of similar type 'const int*'
> ```
>
> (11.4) – a type that is the signed or unsigned type corresponding to the dynamic type of the object,
>
> ```
> // Both si and ui are signed or unsigned
> // types corresponding to each others dynamic
> // types. We can see from this godbolt
> // https://godbolt.org/g/KowGXB) the
> // optimizer assumes aliasing.
> signed int foo( signed int &si,
> unsigned int &ui ) {
>   si = 1;
>   ui = 2;
>   return si;
> }
> ```

4. Why does gcc and clang allow assigning an unsigned int * to int * since they are not compatible types, although they may alias https://twitter.com/shafikyaghmour/status/957702383810658304 and https://gcc.gnu.org/ml/gcc/2003-10/msg00184.html

5. Draft C++17 standard is freely available https://github.com/cplusplus/draft/raw/master/papers/n4659.pdf

Sometimes we want to circumvent the type system and interpret an object as a different type … this is called type punning,

(11.5) – a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,

```
signed int foo( const signed int &si1,
  int &si2);
// Hard to show this one assumes aliasing
```

(11.6) – an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a sub-aggregate or contained union),

```
struct foo {
  int x;
};
// Compiler Explorer example (https://
// godbolt.org/g/z2wJTC) shows aliasing
// assumption
int foobar( foo &fp, int &ip ) {
  fp.x = 1;
  ip = 2;
  return fp.x;
}
foo f;
foobar( f, f.x );
```

(11.7) – a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,

```
struct foo { int x ; };
struct bar : public foo {};
int foobar( foo &f, bar &b ) {
  f.x = 1;
  b.x = 2;
  return f.x;
 }
```

(11.8) – a char, unsigned char, or std::byte type.

```
int foo( std::byte &b, uint32_t &ui ) {
  b = static_cast<std::byte>('a');
  ui = 0xFFFFFFFF;
  return std::to_integer<int>( b ); // b gives
  // us a glvalue expression of type
  // std::byte which can alias an object of
  // type uint32_t
}
```

Worth noting **`signed char`** is not included in the list above, this is a notable difference from C which says a *character type*.

Subtle differences

So although we can see that C and C++ say similar things about aliasing there are some differences that we should be aware of. C++ does not have C's concept of effective type [CPP-3] or compatible type [CCP-4] and C does not have C++'s concept of dynamic type [CCP-5] or *similar type*. Although both have *lvalue* and *rvalue* expressions, C++ also has *glvalue*, *prvalue* and *xvalue* expressions[6]. These differences are mostly out of scope for this article but one interesting example is how to create an object out of malloc'd memory. In C we can set the *effective type*[7], for example, by writing to the memory through an *lvalue* or **`memcpy`**.[8] (See Listing 2.)

```
// The following is valid C but not valid C++
void *p = malloc(sizeof(float));
float f = 1.0f;
memcpy( p, &f, sizeof(float));
  // Effective type of *p is float in C or
float *fp = p;
*fp = 1.0f;  // Effective type of *p is float in C
```

Listing 2

Neither of these methods is sufficient in C++ which requires p**`lacement new`**:

```
float *fp = new (p) float{1.0f} ;
  // Dynamic type of *p is now float
```

Are int8_t and uint8_t char types?

Theoretically neither **`int8_t`** nor **`uint8_t`** have to be **`char`** types but practically they are implemented that way. This is important because if they are really **`char`** types then they also alias similar to **`char`** types. If you are unaware of this it can lead to surprising performance impacts [StackOverflow]. We can see that glibc **`typedef`**s **`int8_t`** [Github-1] and **`uint8_t`** [Github-2] to **`signed char`** and **`unsigned char`** respectively.

This would be hard to change since for C++ it would be an ABI break. This would change name mangling and would break any API using either of those types in their interface.

What is type punning

We have gotten to this point and we may be wondering, why would we want to alias? The answer typically is to *type pun*, often the methods used violate strict aliasing rules.

Sometimes we want to circumvent the type system and interpret an object as a different type. This is called *type punning*, to reinterpret a segment of memory as another type. *Type punning* is useful for tasks that want access to the underlying representation of an object to view, transport or manipulate. Typical areas we find type punning being used are compilers, serialization, networking code, etc…

Traditionally this has been accomplished by taking the address of the object, casting it to a pointer of the type we want to reinterpret it as and then accessing the value, or in other words by aliasing. For example, see Listing 3.

As we have seen earlier this is not a valid aliasing, so we are invoking undefined behavior. But traditionally compilers did not take advantage of strict aliasing rules and this type of code usually just worked, developers

6. 'New' Value Terminology which explains how glvalue, xvalue and prvalue came about http://www.stroustrup.com/terminology.pdf
7. Effective types and aliasing https://gustedt.wordpress.com/2016/08/17/effective-types-and-aliasing/
8. 'constructing' a trivially-copyable object with memcpy https://stackoverflow.com/q/30114397/1708801

```
int x =  1 ;

// In C, not a valid aliasing
float *fp = (float*)&x ;

// In C++, not a valid aliasing
float *fp = reinterpret_cast<float*>(&x) ;

printf( "%f\n", *fp ) ;
```

Listing 3

have unfortunately gotten used to doing things this way. A common alternate method for type punning is through unions, which is valid in C but undefined behavior in C++13[9] (see Listing 4).

This is not valid in C++ and some consider the purpose of unions to be solely for implementing variant types and feel using unions for type punning is an abuse.

How do we Type Pun correctly?

The standard blessed method for *type punning* in both C and C++ is **memcpy**. This may seem a little heavy handed but the optimizer should recognize the use of **memcpy** for *type punning* and optimize it away and generate a register to register move. For example, if we know **int64_t** is the same size as **double**:

```
static_assert( sizeof( double ) ==
  sizeof( int64_t ) );
// C++17 does not require a message
```

we can use **memcpy**:

```
void func1( double d ) {
  std::int64_t n;
  std::memcpy(&n, &d, sizeof d);
  //...
```

At a sufficient optimization level, any decent modern compiler generates identical code to the previously mentioned **reinterpret_cast** method or union method for type punning. Examining the generated code we see it uses just **register mov**.

Type punning arrays

But, what if we want to type pun an array of **unsigned char** into a series of **unsigned int**s and then perform an operation on each **unsigned int** value? We can use **memcpy** to pun the **unsigned char array** into a temporary of type **unsigned int**. The optimizer will still manage to see through the **memcpy** and optimize away both the temporary and the copy and operate directly on the underlying data (Listing 5).

In the example, we take a **char* p**, assume it points to multiple chunks of **sizeof(unsigned int)** data, we type pun each chunk of data as an **unsigned int**, compute **foo()** on each chunk of type punned data and sum it into **result** and return the final value.

The assembly for the body of the loop shows the optimizer reduces the body into a direct access of the underlying **unsigned char array** as an **unsigned int**, adding it directly into **eax**:

```
add     eax, dword ptr [rdi + rcx]
```

```
union u1
{
  int n;
  float f;
};
union u1 u;
u.f = 1.0f;
printf( "%d\n", u.n ); // UB in C++ n is not the
                       // active member
```

Listing 4

9. Unions and memcpy and type punning: https://stackoverflow.com/q/25664848/1708801

```
// Simple operation just return the value back
int foo( unsigned int x ) { return x ; }
// Assume len is a multiple of sizeof(unsigned
// int)
int bar( unsigned char *p, size_t len ) {
  int result = 0;
  for( size_t index = 0; index < len;
       index += sizeof(unsigned int) ) {
    unsigned int ui = 0;
    std::memcpy( &ui, &p[index],
      sizeof(unsigned int) );
    result += foo( ui ) ;
  }
  return result;
}
```

Listing 5

Listing 6 is the same code but using **reinterpret_cast** to type pun (violates strict aliasing).

C++20 and bit_cast

In C++20 we may gain **bit_cast**[10], which gives a simple and safe way to type-pun as well as being usable in a constexpr context.

The following is an example of how to use **bit_cast** to type pun an **unsigned int** to **float**:

```
std::cout << bit_cast<float>(0x447a0000) << "\n";
//assuming sizeof(float) == sizeof(unsigned int)
```

In the case where *To* and *From* types don't have the same size, it requires us to use an intermediate struct[11]. We will use a struct containing a **sizeof(unsigned int)** character array (*assumes 4 byte* **unsigned int**) to be the *From* type and **unsigned int** as the *To* type. (See Listing 7.)

It is unfortunate that we need this intermediate type but that is the current constraint of **bit_cast**.

What is the common initial sequence

The common initial sequence is defined in the draft standard [Standard-1, para 22], which gives the following examples to demonstrate the concept:

```
struct A { int a; char b; };
struct B { const int b1; volatile char b2; };
struct C { int c; unsigned : 0; char b; };
struct D { int d; char b : 4; };
struct E { unsigned int e; char b; };
```

The common initial sequence of A and B comprises all members of either class.

The common initial sequence of A and C and of A and D comprises the first member in each case.

The common initial sequence of A and E is empty.

```
// Assume len is a multiple of sizeof(unsigned int)
int bar( unsigned char *p, size_t len ) {
  int result = 0;
  for( size_t index = 0; index < len;
       index += sizeof(unsigned int) ) {
    unsigned int ui = *reinterpret_cast
      <unsigned int*>(&p[index]);
    result += foo( ui );
  }
  return result;
}
```

Listing 6

10. Revision two of the **bit_cast<>** proposal http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0476r2.html
11. How to use **bit_cast** to type pun a unsigned char array https://gist.github.com/shafik/a956a17d00024b32b35634eeba1eb49e

```
struct uint_chars {
 unsigned char arr[sizeof( unsigned int )] = {} ;
   // Assume sizeof( unsigned int ) == 4
 };
 // Assume len is a multiple of 4
 int bar( unsigned char *p, size_t len ) {
   int result = 0;

   for( size_t index = 0;
     index < len; index += sizeof(unsigned int) )
   {
     uint_chars f;
     std::memcpy( f.arr, &p[index],
       sizeof(unsigned int));
     unsigned int result =
       bit_cast<unsigned int>(f);
     result += foo( result );
   }
   return result;
}
```

Listing 7

It says that we are allowed to read the non-static data member of the non-active member if it is part of the common initial sequence of the structs [Standard-1, para25]

```
struct T1 { int a, b; };
struct T2 { int c; double d; };
union U { T1 t1; T2 t2; };
int f() {
  U u = { { 1, 2 } };   // active member is t1
  return u.t2.c;        // OK, as if u.t1.a were
                        // nominated
}
```

Note, this is not allowed in a constant expression context [Standard-2, para 5.9]. So something like Listing 8 would be ok.

Note that this relies on unions [Standard-3 para 6.3]. This says if the assignment is starting the lifetime of the proper type with limitations such as using a built-in or a trivial assignment operator, the example in Listing 9 invokes undefined behavior.

There can be other tricky cases to watch out for (see Listing 10).

It is likely the common initial sequence rule was put in place to allow discriminated union without having the discriminator outside the union and therefore likely have padding between the discriminator and the union itself, for example:

```
union { struct { char kind; ... } a;
struct { char kind; ... } b; ... };
```

```
union U {
  U(int x) : a{.x=x}{}
  struct { int x; } a;
  struct { int x; } b;
};

int f() {
  U u(10);
  u.b.x = 20; // change active member,
              // starts lifetime of b
  u.a.x = 20; // change active member again,
              // starts lifetime of a

  return u.b.x; // ok common initial sequence
}
int main() {
  int a = f();
}
```

Listing 8

```
union U {
  U(int x) : a{.x=x}{}
  struct {
    int x;
    auto &operator=(int r) {
      x = r ;
      return *this;
    }
  } a;
  struct {
    int x;
    auto &operator=(int r) {
      x = r ;
      return *this;
    }
  } b;
};
int f() {
  U u(10);

  u.b = 20; // Does not change the active member
            // assignment is not trivial
            // and UB b/c of store to out of
            // lifetime object
  u.a = 20; // Does not change the active member
            // assignment is not trivial
            // and UB b/c of store to out of
            // lifetime object

  return u.b.x; // still common initial sequence
       // but we have already invoked UB so not ok
}
```

Listing 9

So the common initial sequence rule would allow us to read the **kind** discriminator regardless of which member was active.

Alignment

We have seen in previous examples that violating strict aliasing rules can lead to stores being optimized away. Violating strict aliasing rules can also lead to violations of alignment requirement. Both the C and C++ standard state that objects have *alignment requirements* which restrict where objects can be allocated (*in memory*) and therefore accessed.[12] C11 section *6.2.8 Alignment of objects* says:

> Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the **_Alignas** keyword.

```
union A {
  struct { int x, y; } a;
  struct { int x, y; } b;
};
int f() {
  A a = {.a = {}};
  a.b.x = 1; // Change active member,
    // starts lifetime of b, there is no
    // initialization of y
  return a.b.y; // UB
}
```

Listing 10

12. Unaligned access: https://en.wikipedia.org/wiki/Bus_error#Unaligned_access

The C++17 draft standard in section *[basic.align] paragraph 1*:

> Object types have alignment requirements (6.7.1, 6.7.2) which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier (10.6.2).

Both C99 and C11 are explicit that a conversion that results in a unaligned pointer is undefined behavior, section 6.3.2.3 *Pointers* says:

> A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned) for the pointed-to type, the behavior is undefined. …

Although C++ is not as explicit, I believe this sentence from *[basic.align] paragraph 1* is sufficient:

> …An object type imposes an alignment requirement on every object of that type;…

An example

So let's assume:

- **`alignof(char)`** and **`alignof(int)`** are 1 and 4 respectively
- **`sizeof(int)`** is 4

Then type punning an array of char of size 4 as an **`int`** violates strict aliasing but may also violate alignment requirements if the array has an alignment of 1 or 2 bytes.

```
char arr[4] = { 0x0F, 0x0, 0x0, 0x00 };
  // Could be allocated on a 1 or 2 byte boundary
int x = *reinterpret_cast<int*>(arr);
 // Undefined behavior we have an unaligned
 // pointer
```

Which could lead to reduced performance or a bus error[13] in some situations. Whereas using **`alignas`** to force the array to the same alignment of **`int`** would prevent violating alignment requirements:

```
alignas(alignof(int)) char arr[4] =
  { 0x0F, 0x0, 0x0, 0x00 };
int x = *reinterpret_cast<int*>(arr);
```

Atomics

Another unexpected penalty to unaligned accesses is that it breaks atomics on some architectures. Atomic stores may not appear atomic to other threads on x86 if they are misaligned.[14]

Catching strict aliasing violations

We don't have a lot of good tools for catching strict aliasing in C++, the tools we have will catch some cases of strict aliasing violations and some cases of misaligned loads and stores.

gcc using the flag **`-fstrict-aliasing`** and **`-Wstrict-aliasing`**[15] can catch some cases although not without false positives/negatives. For example the cases in Listing 11[16] will generate a warning in gcc, although it will not catch this additional case:

```
int *p;
p=&a;
printf("%i\n",
  j = *(reinterpret_cast<short*>(p)));
```

```
int a = 1;
short j;
float f = 1.f; // Originally not initialized but
  // tis-kernel caught it was being accessed w/
  // an indeterminate value below
printf("%i\n", j =
  *(reinterpret_cast<short*>(&a)));
printf("%i\n", j =
  *(reinterpret_cast<int*>(&f)));
```

Listing 11

Although clang allows these flags it apparently does not actually implement the warnings.[17]

Another tool we have available to us is ASan[18], which can catch misaligned loads and stores. Although these are not directly strict aliasing violations they are a common result of strict aliasing violations. For example the following cases[19] will generate runtime errors when built with clang using **`-fsanitize=address`**:

```
int *x = new int[2];       // 8 bytes: [0,7].
int *u = (int*)((char*)x + 6); // regardless of
  // alignment of x this will not be an aligned
  // address
*u = 1;                    // Access to range [6-9]
printf( "%d\n", *u );      // Access to range [6-9]
```

The last tool I will recommend is C++ specific and not strictly a tool but a coding practice, don't allow C-style casts. Both gcc and clang will produce a diagnostic for C-style casts using **`-Wold-style-cast`**. This will force any undefined type puns to use **`reinterpret_cast`**, in general **`reinterpret_cast`** should be a flag for closer code review. It is also easier to search your code base for **`reinterpret_cast`** to perform an audit.

For C we have all the tools already covered and we also have tis-interpreter[20], a static analyzer that exhaustively analyzes a program for a large subset of the C language. Given a C verions of the earlier example where using **`-fstrict-aliasing`** misses one case (Listing 12), tis-interpeter is able to catch all three. The example in Listing 13 invokes tis-kernal as tis-interpreter (output is edited for brevity).

Finally there is TySan[21] [Finkel17] which is currently in development. This sanitizer adds type checking information in a shadow memory segment and checks accesses to see if they violate aliasing rules. The tool potentially should be able to catch all aliasing violations but may have a large run-time overhead.

```
int a = 1;
short j;
float f = 1.0 ;

printf("%i\n", j = *((short*)&a));
printf("%i\n", j = *((int*)&f));

int *p;

p=&a;
printf("%i\n", j = *((short*)p));
```

Listing 12

13. A bug story: data alignment on x86 http://pzemtsov.github.io/2016/11/06/bug-story-alignment-on-x86.html
14. Demonstrates torn loads for misaligned atomics https://gist.github.com/michaeljclark/31fc67fe41d233a83e9ec8e3702398e8 and tweet referencing this example https://twitter.com/corkmork/status/944421528829009925
15. gcc documentation for **`-Wstrict-aliasing`** https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#index-Wstrict-aliasing
16. Stack Overflow questions examples came from https://stackoverflow.com/q/25117826/1708801
17. Comments indicating clang does not implement **`-Wstrict-aliasing`** https://github.com/llvm-mirror/clang/blob/master/test/Misc/warning-flags-tree.c
18. ASan documentation https://clang.llvm.org/docs/AddressSanitizer.html
19. The unaligned access example take from the Address Sanitizer Algorithm wiki https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm#unaligned-accesses
20. TrustInSoft tis-interpreter https://trust-in-soft.com/tis-interpreter/, strict aliasing checks can be run by building tis-kernel https://github.com/TrustInSoft/tis-kernel
21. TySan patches, clang: https://reviews.llvm.org/D32199 runtime: https://reviews.llvm.org/D32197 llvm: https://reviews.llvm.org/D32198

```
./bin/tis-kernel -sa example1.c
...
example1.c:9:[sa] warning: The pointer (short
  *)(& a) has type short *. It violates strict
  aliasing rules by accessing a cell with
  effective type int.
...
example1.c:10:[sa] warning: The pointer (int *)
  (& f) has type int *. It violates strict
  aliasing rules by accessing a cell with
  effective type float.
  Callstack: main
...
example1.c:15:[sa] warning: The pointer (short
  *)p has type short *. It violates strict
  aliasing rules by accessing a cell with
  effective type int.
```

Listing 13

Conclusion

We have learned about aliasing rules in both C and C++, what it means that the compiler expects that we follow these rules strictly and the consequences of not doing so. We learned about some tools that will help us catch some misuses of aliasing. We have seen a common use for type aliasing is type punning and how to type pun correctly.

Optimizers are slowly getting better at type based aliasing analysis and already break some code that relies on strict aliasing violations. We can expect the optimizations will only get better and will break more code we have been used to just working.

We have standard conformant methods for type punning and in release and sometimes debug builds these methods should be cost free abstractions. We have some tools for catching strict aliasing violations but for C++ they will only catch a small fraction of the cases and for C with tis-interpreter we should be able to catch most violations.

Thank you to those who provided feedback on this write-up: JF Bastien, Christopher Di Bella, Pascal Cuoq, Matt P. Dziubinski, Patrice Roy, Richard Smith and Ólafur Waage.

Of course in the end, all errors are the author's. ■

References

[CPP-1] 'Undefined behavior' https://en.cppreference.com/w/cpp/language/ub

[CPP-2] 'reinterpret_cast conversion' https://en.cppreference.com/w/cpp/language/reinterpret_cast

[CPP-3] 'Effective type' https://en.cppreference.com/w/c/language/object#Effective_type

[CCP-4] 'Compatible types' https://en.cppreference.com/w/c/language/type#Compatible_types

[CCP-5] 'Dynamic type' https://en.cppreference.com/w/cpp/language/type#Dynamic_type

[Finkel17] Hal Finkel (2017) 'The Type Sanitizer: Free Yourself From from -fn-strict-aliasing', presentation at the *LLVM Developers' Meeting*, available from https://www.youtube.com/watch?v=vAXJeN7k32Y

[Github-1] int8_t: https://github.com/lattera/glibc/blob/master/sysdeps/generic/stdint.h#L36

[Github-2] unint8_t: https://github.com/lattera/glibc/blob/master/sysdeps/generic/stdint.h#L48

[StackOverflow] 'Using this pointer causes strange deoptimization in hot loop' https://stackoverflow.com/questions/26295216/using-this-pointer-causes-strange-deoptimization-in-hot-loop

[Standard-1] Draft standard, Classes: Class members: General, paragraphs 22 (http://eel.is/c++draft/class.mem#general-22) and 25 (http://eel.is/c++draft/class.mem#general-25)

[Standard-2] Draft standard, Expressions: Constant expressions, para 5.9: http://eel.is/c++draft/expr.const#5.9

[Standard-3] Draft standard, Classes: Unions: General, paragraph 6.3: http://eel.is/c++draft/class.union#general-6.sentence-3

Afterwood

Design Patterns emerged last century. Chris Oldwood explains why he thinks they are still relevant.

A perennial question that crops up fairly regularly on social media questions the value of design patterns. The clickbait-styled affairs declare that design patterns and the similarly titled Gang of Four (GoF) book have little relevance in today's programming toolbox and should simply be ignored as an anachronism. The more curious programmer that seeks more substance may struggle to see past a movement which superficially appears to have seen better days.

And who can blame them? If you're a fresh-faced programmer who comes across a book from 1994 with example C++ code that passes around raw pointers coupled with 90's style UML class diagrams of inheritance hierarchies, wouldn't you be somewhat concerned? The nail in the coffin would probably be the number of patterns in the seminal GoF book which evaporate once you consider C++ has supported lambda functions for almost a decade now – functors are so last Millennium! Even the Double-Checked Lock, should one be inclined to touch the pariah that is Singleton, finally works by design.

Java, which didn't exist at the time the book was written but was still hugely popular during the rise of the movement, has even sprouted lambda functions to remove the burden of a few recurring problems the book addressed. While on the subject of Java, some of the aversion to design patterns must lie with that unfortunate practice of naming classes by chaining together the names of design patterns, as if trying to win at 'Design Pattern Bingo'. I have personally been the recipient of code which contained the monstrosity:

```
public class UnitOfWorkXxxServiceFacadeDecorator
```

(In this instance, the entire class hierarchy was replaced by a couple of succinctly named C# extension methods that wrapped a caller-provided closure.)

So, are they right then? Are design patterns a waste of time – a solution to a problem in a context where that context was the early '90s and the problem was simply poorly designed programming languages?

No, I don't think so. The Gang of Four legacy is not some old-fashioned C++ code and a bunch of UML diagrams with large class hierarchies; it's the *ubiquitous language* they created. The value was never in how they realized the solution in code but in the *vocabulary* they chose for describing the recurring problem they identified. By distilling the essence into just a single word (or three), they have given us the power to communicate complex ideas easily. Imagine how much longer a design meeting would take if you had to keep saying 'a one-to-many relationship between objects so that when one object changes all its dependants are notified and updated automatically' instead of simply saying 'observer'?

Even if some design patterns use terms that already had a fairly established meaning, such as Proxy (GoF), Cache (POSA3), and Pool (POSA3), the community has taken those ideas and given them a more *usable* definition. Instead of leaving them to remain ambiguous, they have given them clarity. As Dijkstra once said:

> The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Each pattern does not exist in a vacuum. Together, it and other related patterns form their own *language* that allows people to explore design options in a specific problem space without getting dragged into the implementation minutiae too soon. Once the foundation patterns in the original GoF text and its ilk, such as the first volume of the *Pattern Oriented Software Architecture* (POSA) series, had arrived further catalogues appeared that focused on particular topics such as resource management, concurrency, integration, etc. Both the POSA and *Pattern Languages of Program Design* (PLoPD) series stretched to 5 volumes and the annual PLoP conference continues today despite starting back in 1994. To paraphrase Mark Twain: any rumours of the patterns community's demise are greatly exaggerated.

Many of those early patterns are still relevant to today's design discussions because the vocabulary is still with us. The aforementioned 'observer' might have matured over the years and more closely resembles its sibling – Multicast, introduced in the follow-up book *Pattern Hatching* – but it's still there at the heart of any event driven system. The terms Adapter, Facade, Factory, and Proxy had a life before the GoF and naturally continue to live on, albeit with a more refined definition. Iterators are a fundamental part of C++, Java, and C#, and Decorators became a key idiom in Python. Decorators in Python are considered far less rigid than the GoF definition, more akin to Aspect Oriented Programming, but that might be because the AOP term post-dates the Decorator pattern; either way, there are big similarities in the *metaphor*, and that aids communication. One pattern that definitely seems to have gained in popularity over the years is Builder (coupled with fluent interfaces), no doubt due to the rise of automated testing where non-trivial object graphs created in tests need the salient details to remain prominent

Patterns aside, the GoF book still has plenty to say. Section 1.6 alone contains a useful summary of many of the core ideas behind object-orientation including what is probably one of *the* most significant programming axioms – program to an interface, not an implementation

The Conclusion chapter in *Design Patterns* starts by saying "It's possible to argue this book hasn't accomplished much." I think it's fair to say that 25 years later you would be hard pushed to agree with this statement. While the example code is dated and the ubiquity of closures in modern programming languages have consigned some of the solutions to the history books, the vocabulary appears to be timeless. Their concise lexicon is the real legacy of the Gang of Four, and I for one thank them for helping me spend less time in meetings and having more productive design conversations. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the lounge below his bedroom. With no Godmanchester duck race to commentate on this year, he's been even more easily distracted by messages to gort@cix.co.uk or @chrisoldwood

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe