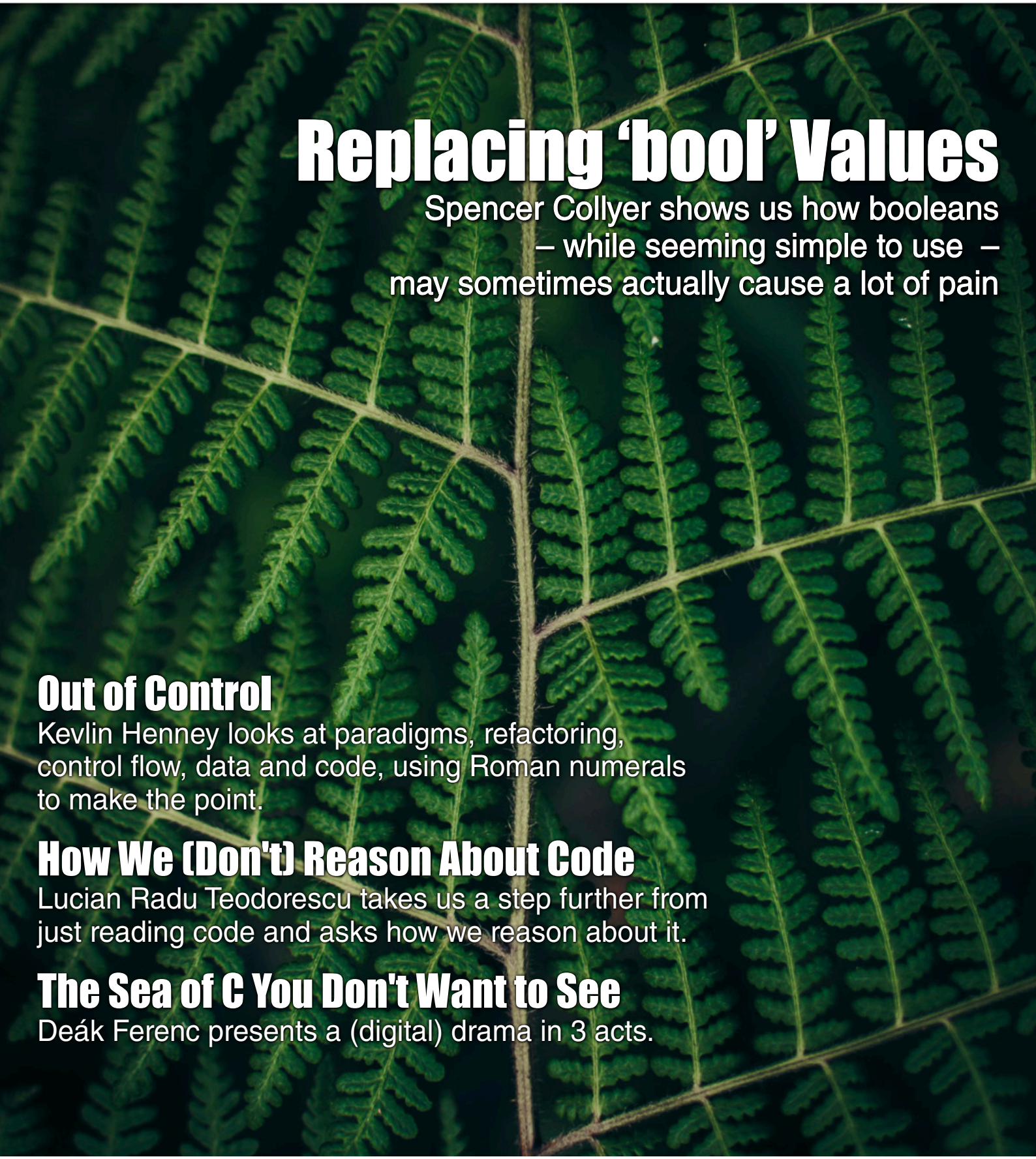
overload 163

JUNE 2021 £4.50



Replacing 'bool' Values

Spencer Collyer shows us how booleans – while seeming simple to use – may sometimes actually cause a lot of pain

Out of Control

Kevlin Henney looks at paradigms, refactoring, control flow, data and code, using Roman numerals to make the point.

How We (Don't) Reason About Code

Lucian Radu Teodorescu takes us a step further from just reading code and asks how we reason about it.

The Sea of C You Don't Want to See

Deák Ferenc presents a (digital) drama in 3 acts.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 163

June 2021

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Pandu Ior,
on Unsplash.

Copy deadlines

All articles intended for publication in Overload 164 should be submitted by 1st July 2021 and those for Overload 165 by 1st September 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

Copyrights and Trade Marks

Geek, Nerd or Neither?

Typecasting can be useful but has many dangers. Frances Buontempo considers how to pick your way through categories.

Sometimes we put labels on stuff or even people to help us navigate our way through life. A magazine might have a couple of pages at the front for an editorial and yet, though the pages may be filled with words, that doesn't mean what is written is really an editorial. I have previously tried using machine learning to generate the words for me [Buontempo14], but the results were suspect. Furthermore, I have recently been revisiting playing with Monte Carlo chains (watch this space for details) and derailed myself yet again. Some may claim having hundreds of thoughts and distractions in place is some kind of neuro-atypical attention deficit hyperactivity disorder, while others may recognize joy in having so many interesting things to find out about and muse on. Sometimes labels help and sometimes labels hinder.

First impressions are a type of labelling; first impressions count. What they count is another matter. If I wear jeans and a t-shirt for a meeting, I may look out of place and yet if I wear a suit in a 'hipster' office I will also look out of place. Nowadays, wearing pyjamas on a video call has become the great leveller. The context influences whether the initial judgement is positive or negative. First impressions also interleave with stereotypes: try looking for an image of a programmer. You will find different people over time and across the globe. In one place, a middle aged man, fed on junk food, sitting in a darkened room. In another, groups of women. It depends. It's easy to forget the clichés about people we use to navigate through life. I've been reading *Software Design X-Ray Patterns* by Adam Tornhill recently, and he touched on the subject, saying "The fundamental attribution error is a principle from social psychology that describes our tendency to overestimate the influence of personality – such as competence and carefulness – as we explain the behavior of people." In other words, we underestimate the context, whether that's looking at some legacy code or at someone's behaviour. You know, thoughts like "Typical Java coder" and the like.

Stereotypes and clichés do give broad brush strokes and often have roots in the actual but, they are a compressed, 'lossy' summary. Worse, it is tempting to extrapolate from one category to another, like a badly trained recommender system. Do not extrapolate from three data points and do not draw conclusions from a coincidence. Well, you can, but if you do, use this as a hypothesis rather than a fact. Challenge your assumptions.

Have you ever described something as 'typical'? Have you been described as, say, a typical geek? Though some of us may proudly wear 'geek' or 'nerd' as a kind of badge, it can be dehumanising. Using a label suggests a programmed system with no way to perform in interesting or unexpected ways. A deterministic system gives the same outputs for the same inputs, every time. Now, AI code tends to use randomness to explore possibilities and appear to have agency. Perhaps we are all programmed bots in a simulation, but that's another matter. I wonder if we could spot a 'typical AI'. Would its behaviour give it away? Philip K Dick explored this in 'Do Androids Dream of Electric Sheep?' The androids tended to give themselves away by lacking empathy, a trait some believe that coders also have, tending to be further along the autistic spectrum than most. I stopped myself from trying to find any evidence for this, because I know, first I'll end up with far too many tabs open in my browser, and second, I will be hunting out evidence to support a statement I just made. This is a form of confirmation bias. Though I know many other tech people who score highly on autism tests, I am not sure if this label is a help or a hindrance.

A first impression, used as a broad brush stroke to find something in common on a first meeting, can be enabling. If you are wearing a Slayer t-shirt that invites me to throw up metal horns \m/ , a chat about certain types of music will ensue. If you mention C++ on your profile, you give me a potential conversation starter. These hints and clues may be esoteric and not universally recognized. That's OK. The point of a label is for disambiguation. In contrast, all the kids in a class turning on the skinny one who wears glasses and calling them 'four eyes' is another matter. Labels to help start learning and communication are useful; labels for bullying are not. For a while, glasses became a fashion must have, with non-prescription specs available along-side others accessories in shops when shops were a thing. An unexpected phenomenon, but looking like a geek was on trend. This did cause some fake geek girl memes, but perhaps is a hint of a change of mindset, rather than purely a cynical marketing ploy? Who knows?

So, what does a geek look like? The internet does insist on glasses, in the main. I think being a geek is more a state of mind: what's inside counts. Wikipedia tells me 'geek' was a pejorative term for "peculiar person, especially one who is perceived to be overly intellectual, unfashionable, boring, or socially awkward" [Wikipedia], and then mysteriously says the negative connotation is due to its "earlier association with carnival performers". Possibly something to do with the word 'geck' in some dialects meaning fool or clown. Sometimes the label 'nerd' is invoked instead of geek, or the words used interchangeably. Do these terms differ? Maybe. My internet search says both wear glasses, but a geek wears a t-shirt whereas a nerd wears a shirt and tie. Again, shallow: it's what's inside that counts. I tend to use 'nerd' for someone who knows their subject in depth, armed with a vast number of facts and snippets of info. In contrast, a geek would dismantle almost anything to find out how it works. I asked twitter if people considered themselves a geek, nerd or neither. Each option got about a third of the votes, so I have a nice balance in my followers. One person did say they use 'nerd' as a very positive term and like nerds because they live for the things they love. Some preferred the

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.

term ‘geek’ and many chose ‘neither’. Not every who codes will look or act identically. Once too often I’ve heard the phrase, “You programmers are all the same” followed by some claim about not being able to create what the users want, or spending time faffing about with whitespace or so on. Incredibly unfair; don’t use caricatures of people to prejudge their character.

Mindlessly applying labels can cause all kinds of harm. And yet, labelling data, particularly images, has become a part of everyday life. My phone has some ‘AI lens’ for the camera app, which tries to label what it sees. Sometimes a cat, sometimes a dog (wrong – it was the cat, but hey), once a waterfall (also wrong – the cat was sitting on a green chair) and failing that ‘portrait’. I presume if I rotate the phone appropriately, it would become ‘landscape’. There must be a way to provide feedback to increase the intelligence of the so-called AI, but I’ve never bothered investigating. Many forms of machine learning or AI required labelled data, so the mind-numbing task of attaching labels cat/dog, good/bad and so on to reams of data must be farmed out to humans. This is now tending to be called human in the loop machine learning. This is quite an old phrase and used to mean a human steering where the machine was going; a much broader idea than a human slave slapping labels on datasets. How many times have you been asked to identify traffic lights when a sign-in like reCAPTCHA [Google] goes wrong? Why are you being asked to do this? To label data for self-driving cars, one presumes. What happens if you deliberately get it wrong a few times? You get more pictures to label. Your choice is compared to others and the majority votes wins. The wisdom of crowds.

Sometimes a crowd is more like an angry mob and shouldn’t be trusted. However, while one individual may be terrible at making a numerical estimate, the average can often be very close to the actual value. Wikipedia (again, forgive me) reports Galton’s “surprise that the crowd at a county fair accurately guessed the weight of an ox when their individual guesses were averaged (the average was closer to the ox’s true butchered weight than the estimates of most crowd members)” [Wikipedia-2] If you play planning poker at work, perhaps you should take the median or mean of the story points and see where that gets you. Guessing, or predicting, the number in a range is one task we leave to stats and AI, and averages or ensemble methods can work well here. Finding the average of a category or label is much harder. You could count the most common, but with equal votes between geek, nerd or neither you will be hard pressed to report the winner. Building consensus requires more than taking an average or declaring ‘nerd’ the winner having received 35.3% of votes, with ‘geek’ and ‘neither’ a close second at 32.4%.

Labels can be useful as a starting point for learning or exploring. We learn language by pointing and declaring “cat” or “dog”, or, in my nephew’s case, “Mummy” while pointing at a leather clad biker in a pub. Fortunately, the biker saw the funny side. When we label data for AI, it can cheat and memorise the labels. Perfect recall is not the same as intelligence. If a computer has a lookup table of outcomes for possible moves in chess, is it displaying intelligence? If a person can recite hundreds of digits or pi, are they exhibiting understanding? Probably not. Does it matter? It depends. I suspect we cannot define intelligence accurately. What we find amazing, in terms of the capabilities of tech or humans, changes over time. The thermostat, to control a heating system, was invented a long time ago. It was referred to as a heat governor, capable of ‘self-acting’ [Ure37]. The idea of self-acting or agency often plays a part in our intuitive sense of what intelligence means, and yet seeing a thermostat as AI would seem weird today. Perhaps one day, photos labelled automatically, chat bots helping customers, self-driving cars and transporters and replicators will be the norm. ‘Normal’ has a time axis, so if you feel out of sorts, like a socially awkward geek or nerd, fear not, you might just be an embodiment giving a glimpse of the future.

When you discover a word you don’t know, how comfortable do you feel using it in a sentence? A child may not be shy about pointing at everyone in sight and declaring them to be “Mummy”, but may become a bit more self-conscious later on. Don’t let that put you off learning new things. Something similar happens when you try to take on board new programming paradigms. Many listen and watch others before having a go, perhaps in the privacy of a bedroom, or shed or wherever. Perhaps you feel like a fraud putting a new language on your CV which you have just started to learn. Maybe you don’t feel you know enough to write an article. These hints of imposter syndrome bite us all from time to time. Don’t let your inner voice tell you, “You can’t do this”. You got this. Fake it till you make it. Try out the new word, write different code, punt an article to a magazine. It might be fun. You might learn something. Don’t let labels, yours or others, box you in.

Let’s avoid too much typecasting. Start with labels, sure, but we all know the trouble inappropriate `reinterpret_cast` can cause if applied carelessly. What matters most is how you self-identify. Are you a geek, a nerd or neither? I don’t mind, but I’m interested in what you want to talk about and what you do. Imagine a world, where you could choose your own label. Imagine if you could choose your own job title. How would you describe yourself? I’d settle for Code Improver. What about you? Whether you count yourself as a geek, nerd or neither is by the by. Above all, you are human and have interesting things to do and say.

References

[Buontempo14] Random (Non)sense, *Overload*, 22(119):2-3, February 2014 https://accu.org/journals/overload/22/119/buontempo_1853/

[Google] reCAPTCHA: https://www.google.com/recaptcha/about/

[Ure37] Andrew Ure, ‘On the thermostat or Heat Governor, a self-acting physical apparatus for regulating Temperature’, The Royal Society, 1837

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

[Wikipedia-2] Wisdom of Crowds: https://en.wikipedia.org/wiki/The_Wisdom_of_Crowds

Replacing 'bool' Values

Booleans seem simple to use. Spencer Collyer considers when they can actually cause a world of pain.

When used in the context of programming, the term *Dimensional Analysis* refers to the technique of defining types to represent the kinds of values used in the program. With the appropriate operations between objects of those types defined the compiler can check the expressions in the code to make sure they are valid. This is not generally possible if you rely on using the fundamental types like `int` or `double`.

For instance, say you have a program that deals with distances, durations, and speeds. It should be obvious that adding or subtracting a distance and a speed are invalid operations, but the compiler would not be able to tell you that this code is incorrect:

```
double distance = 10;
double speed = 2;
double duration = distance - speed;
```

However, if you have types `Distance`, `Duration`, and `Speed`, with only the valid operations between them defined, the compiler can issue an error for this code:

```
Distance distance = 10;
Speed speed = 2;
Duration duration = distance - speed;
```

To be useable, when using this technique most types need to be defined as classes or structures. There are libraries available for many languages that make this task easier – a recent (2018) survey of them for many languages can be found in [Preussner].

However, if you would normally think of using a `bool` variable to hold the value, there are several mechanisms available in the C++ language that can be used instead, with no need for library support. We will outline some of them in this article, as well as try to explain why you might choose to do so.

When reading the problem descriptions and suggested solutions below, and wondering if you want to use them, it is worth applying what I call the TLAMP principle. Pronounced 'tee lamp', it stands for Think Like A Maintenance Programmer. What may seem obvious to you when first writing a piece of code can look completely opaque to someone doing maintenance work on that code in the future. They want the code to be as clear as possible on first reading. That later programmer could be yourself in six months – when you haven't looked at the code for that length of time what seemed obvious when you were writing it may not be so later.

Why bother when bools are so simple?

You might ask why we would bother replacing a `bool` value with some other mechanism when `bool`s are so simple to use. In this section, we will outline some of the problems with using `bool`s that make it worthwhile to at least consider doing so.

Spencer Collyer Spencer has been programming for more years than he cares to remember, mostly in the financial sector, although in his younger years he worked on projects as diverse as monitoring water treatment works on the one hand, and television programme scheduling on the other.

Many of these problems arise because programmers decide to use `bool` variables or parameters just because the value being represented can only take two values. If you get into the habit of only using `bool` for values that are going to be used in boolean expressions, you can avoid them to a large extent.

To illustrate some of the problems we will use the following example[1].

Imagine a water company wants a system written to monitor and control its water network. There is a large amount of equipment on the network, such as sensors for measuring things like flow rate, temperature, chemical concentrations, and also control equipment such as valves and pumps to allow the flows in the network to be controlled. This network has evolved over many years, and the equipment is from different manufacturers and of different ages, with a variety of protocols used to talk to it.

The initial analysis leads to a design in which the connections to this equipment are handled by a `Connection` base class which provides a standard interface, with a set of classes derived from `Connection` that handle the details of each protocol. There is a factory function, called `CreateConnection`, which returns an object of the correct class for each connection. Each class is designed to handle either input or output on the connection. The initial design for the `CreateConnection` function interface looks like the following:

```
ConnectionPtr CreateConnection(
  std::string_view id
, bool is_output);
```

The `is_output` parameter determines whether an output (`true`) or input (`false`) connection is being created.

An additional requirement is for some users to have elevated permissions on some connections. This allows for operations like controlling pump speeds to alter flow rates, for instance. To handle this, a second `bool` parameter is added to indicate if the user is privileged or not.

During testing of the system, it is found that some parts of the network are so old that they only support 7-bit data. As a result, communications over these connections have to be encoded from binary to ASCII. To indicate this a further `bool` parameter is added to the function, called `is_encoded`, to indicate if this encoding is required or not.

Finally, a security review of the system raises concerns that some of the connections go over public networks, and a requirement is made that those connections need to be encrypted. A final parameter is added to the function called `is_encrypted` which indicates if the connection needs to be encrypted or not.

The final prototype for the function now looks like Listing 1, overleaf.

1. This example may seem contrived, but I once worked on a system that had many functions with three or four `bool` parameters. A lot of the calls were done using literal values for some or all of the parameters, and only checking the surrounding code could confirm whether the values were correct.

Unless a programmer knows the function prototype off by heart, it would be easy for them to get the parameter order wrong, and the compiler won't warn about it

```
ConnectionPtr CreateConnection(
  std::string_view id
  , bool is_output
  , bool is_authorised
  , bool is_encoded
  , bool is_encrypted);
```

Listing 1

The meaning of true

Or rather, the meaning of **`true`**. And, indeed, **`false`**. In many cases where a variable can take just two values, and so at first looks like a good candidate to use a **`bool`**, it is not obvious which value should map to **`true`** and which to **`false`**.

The **`is_output`** parameter in the **`CreateConnection`** function is a perfect example of this. The parameter allows the caller of the function to determine if an outgoing or incoming connection is required, but other than the name of the parameter there is nothing that indicates which of those is selected by passing **`true`** and which by passing **`false`**.

You could argue that the name of the parameter shows how it is used, but that relies on anyone reading the code either knowing the prototype because they have seen if before, or else are willing to look it up. Neither of which is guaranteed to be done by a maintenance programmer who is under pressure to get a fix out quickly.

All bools look the same

In many cases, the **`bool`** values do match what we would expect for a given parameter, but they can still be problematic, especially if you have more than one **`bool`** in the parameter list. This is because all **`bool`**s look the same to the compiler.

The **`CreateConnection`** function illustrates this problem. If we ignore the problem with it outlined above, it is reasonable that the **`is_output`** parameter is the first **`bool`** in the list, as the direction of the connection is the most important property it has.

Good arguments could be made for any order of the other three **`bool`** parameters however – the one chosen here has arisen simply because of the order the requirement for them came up in the development process. For instance, it could be argued that the **`is_encoded`** and **`is_encrypted`** parameters are the wrong way around for an outbound connection, as encryption occurs before encoding when sending a message.

Unless a programmer knows the function prototype off by heart, it would be easy for them to get the parameter order wrong, and the compiler won't warn about it. Only extensive testing will ensure all calls are correct.

What can be even more confusing for someone reading the code later is if it uses named variables for the parameters, but gets them in the wrong order. For instance, consider the code in Listing 2.

This will work, in the sense of giving the expected result, because the **`is_encoded`** and **`is_encrypted`** variables have the same value.

```
bool is_encoded =
     /* code that sets value to true */;
bool is_encrypted =
     /* code that sets value to true */;
...
auto connptr = CreateConnection(id, is_output,
  is_authorised, is_encrypted, is_encoded);
```

Listing 2

However, if one of those values needs to change later, or someone copies the code elsewhere and changed one of the values, the result would be incorrect, but it wouldn't be obvious why unless the person reading the code recognises that the last two parameters are in the wrong order.

The compiler cannot report this problem because it just sees the types of parameters passed in. The names of the variables are relevant only to tell it where to read the parameter value from – it doesn't check that they match the names in the function prototype.

Note: This problem doesn't just apply to the **`bool`** type of course – lists of parameters all with the same type can be problematic when trying to work out what each parameter means. This article doesn't deal with that situation but it is worth being aware of it.

Conversions to and from bool

The built-in C++ scalar types all implicitly convert to and from the **`bool`** type. This implicit conversion is useful when writing code that tests that a value is not zero or a null pointer.

Some classes in the standard library also provide an **`operator bool`** to test that an object is in a valid state – for instance, the **`std::basic_ios`** class that is the base of many iostreams classes class provides one to check if an error has occurred on the stream.

Another use for this implicit conversion is in the **`!!`** pseudo-operator, which can be used to return the **`bool`** equivalent of an expression[2] in any cases where automatic conversion doesn't happen.

However, this implicit conversion can cause problems if it happens when you are not expecting it. For instance when calling a function, if you pass a scalar value in a parameter that expects a **`bool`**, it will be converted.

Consider the code in Listing 3. The two **`PrintArgs`** functions simply output their prototype and the values they have been called with. The second one allows the **`bool`** parameter to be defaulted, hence why the **`short`** is placed before it in the parameter list.

Unfortunately, when this program is compiled, the line labelled `// 3` fails to compile. The output in Listing 4 shows the errors when the code is compiled with the GCC on my Linux system.

2. I have seen this pseudo-operator referred to as the 'normalise operator'. The way it works is by relying on the right-to-left binding of the **`!`** operator. The right-hand **`!`** applies to the operand, forcing it to the **`bool`** equivalent and then negating the result. The left-hand **`!`** then applies to the resulting value and negates it again, giving us back the **`bool`** equivalent of the original operand.

the overload resolution process is done, and we still have two candidates with no way to pick between them, and hence the call is ambiguous

```
#include <iostream>
#include <string_view>

void PrintArgs(const std::string& s,
  bool to_uc = false)
{
    std::cout
      << "Called PrintArgs(string, bool) with ("
      << s << ", " << to_uc << ")\n";
}
void PrintArgs(const std::string& s, short len,
  bool to_uc = false)
{
  std::cout << "Called PrintArgs(string, short,
    bool) with (" << s << ", " << len << ",
    " << to_uc << ")\n";
}

int main()
{
    std::cout << std::boolalpha;
    PrintArgs("Abc");          // 1
    PrintArgs("Abc", true);    // 2
    PrintArgs("Abc", 2);       // 3
    PrintArgs("Abc", 2, true); // 4
}
```

Listing 3

The problem arises during the overload resolution process to decide which function should be called. The full details of overload resolution are complex (see [CppRef1]) but the case here is relatively simple. An important point is that an integer with no suffix in the code has type **int** so the **2** in the problematic call has type **int**.

When the compiler sees the call in the line labelled `// 3`, it first finds all the declared functions named **PrintArgs** and adds them to the overload set. It then checks each one to see if it matches the arguments given. This proceeds as follows:

- For the two-parameter function, the **"Abc"** can be converted to a **std::string**, so the first argument matches the first parameter. The **2** is an **int**, and it can be implicitly converted to the **bool** type of the second parameter. Both arguments match the function parameters, so the function is a candidate.
- For the three-parameter function, the **"Abc"** is a match as above. The **2** is an **int**, and that can be implicitly converted to a **short** using a narrowing conversion. The third argument is missing but the parameter has a default value, so it is ignored in the matching. The arguments match the parameter list for this function, so it is also a candidate.

At this point, the overload resolution process is done, and we still have two candidates with no way to pick between them, and hence the call is ambiguous.

To solve the ambiguity the programmer changes the second definition so it looks like the one in Listing 5 (overleaf). Unfortunately, the default value for the **bool** parameter can no longer be used, but the ambiguity no longer occurs.

The program now compiles without any problems and appears to run fine as well, producing the output in Listing 6. However, looking closely at the output shows that the output from the lines labelled `// 3` and `// 4` do not match the arguments in the code. This is again because of implicit conversions.

In the case of the call in line `// 3`, the **2** is converted from **int** to **bool**, ending up with the value **true**.

In the case of the call in line `// 4`, the **2** in the second argument is again converted from **int** to the **bool** value **true**, and the **true** in the third argument is converted from **bool** to **short**, ending up with the value **1**.

This kind of bug can arise if you change the interface of a function and rely on the compiler to catch any calls with incorrect arguments. As can be seen in this example, it does not always issue warnings or errors for calls that you should have changed. A refactoring tool may be able to find them, or you might simply have to check each call by hand.

This kind of problem with implicit conversions can arise in other cases, but the one going to or from a **bool** is more insidious because the values of a **bool** are fundamentally different from the values of a scalar type, in

```
conversion-1.cpp: In function 'int main()':
conversion-1.cpp:19:23: error: call of overloaded 'PrintArgs(const char [4], int)' is ambiguous
   19 |     PrintArgs("Abc", 2);       // 3
      |                       ^
conversion-1.cpp:4:6: note: candidate: 'void PrintArgs(const string&, bool)'
    4 | void PrintArgs(const std::string& s, bool to_uc = false)
      |      ^~~~~~~~~
conversion-1.cpp:9:6: note: candidate: 'void PrintArgs(const string&, short int, bool)'
    9 | void PrintArgs(const std::string& s, short len, bool to_uc = false)
      |      ^~~~~~~~~
```

Listing 4

each additional parameter replaced doubles the number of new functions required

```
#include <iostream>
#include <string_view>

void PrintArgs(const std::string& s,
  bool to_uc = false)
{
  std::cout
    << "Called PrintArgs(string, bool) with
    (" << s << ", " << to_uc << ")\n";
}
void PrintArgs(const std::string& s, bool to_uc,
   short len)
{
  std::cout << "Called PrintArgs(string, bool,
    short) with (" << s << ", " << to_uc << ",
    " << len << ")\n";
}
int main()
{
  std::cout << std::boolalpha;
  PrintArgs("Abc");           // 1
  PrintArgs("Abc", true);     // 2
  PrintArgs("Abc", 2);        // 3
  PrintArgs("Abc", 2, true); // 4
}
```

Listing 5

```
Called PrintArgs(string, bool) with (Abc, false)
Called PrintArgs(string, bool) with (Abc, true)
Called PrintArgs(string, bool) with (Abc, true)
Called PrintArgs(string, bool, short) with (Abc,
  true, 1)
```

Listing 6

that they are logical truth values, not numbers. The fact that the C++ spec dictates that `false` maps to a value of 0 and `true` maps to a value of 1 when converted to a number is just a convention to allow the conversion to occur. Other languages don't allow such conversion, or if they do they use different mappings[3].

It may not matter to you if an `int` gets converted to a `short` as long as the value doesn't change, but with a `bool` you are going from a logical value to a number or from a number to a logical value, which is a more fundamental change, and one that may well make no sense in the context of the code.

3. Anyone old enough to have used one of the microcomputers released during the 1980s home computer boom might remember that the BASIC built into many of them used -1 for the 'true' value, presumably because the representation of that value has all bits set to 1. Sinclair Basic, as used on the ZX81 and Spectrum, went its own way and used 1 for the 'true' value.

More than two values

It might sound trite to say it, but a `bool` value can only hold two different values. This may become a problem if you realise that a parameter needs to hold more than two values.

For instance, in our water company example, the binary-to-ASCII encoding on some connections might need doing using UUencoding [Wikipedia-1], while others might use Base64 [Wikipedia-2].

With just two values for `is_encoded` and one of those used to indicate no encoding is required, you cannot represent those two different types of encoding in the parameter. You have two options in this case – either add another parameter to give the encoding or else convert the `bool` parameter to some other kind that can represent three (or more) values. The first extends the function interface even more, and the second has all the possible problems associated with conversion to/from `bool` given above.

Alternatives to bool

We have seen why you might want to avoid using `bool` variables and parameters, now we will show some methods that you can use to do so. As mentioned previously, all of these are available from the core language, with no library support required.

Some of these methods are designed primarily for replacing function parameters, while the others are more general and can be used to replace variables as well.

Split one function into two (or more)

Rather than having a single function with different functionality selected by passing a `bool` parameter, split the functionality into two different functions, with their names indicating what is being done. Any common functionality can be split off into a third function that the two new functions call.

This is particularly useful for the case where it is not obvious what the mapping from the `true` or `false` values to the selected functionality is.

In our water company example, rather than passing the `is_output` parameter, you would instead create functions called `CreateOutboundConnection` and `CreateInboundConnection`, where the names indicate what type of connection is being created.

This method is fine for replacing one or maybe two parameters. The problem with doing more than that is that each additional parameter replaced doubles the number of new functions required. Also, with descriptive function names, they can get unmanageably long very quickly.

Using a flags variable

This method involves replacing one or more `bool` values with a variable holding a collection of single-bit fields. This will generally be an integer value or a `std::bitset`.

An example of a flags variable in the standard library is the mode parameter of the `std::ifstream` and `std::ofstream` constructors, which uses the `std::ios_base::openmode` type.

```
#include <iostream>

struct A
{
  int a1 : 1;
  int a2 : 1;
  int a3 : 1;
};
struct B
{
  unsigned int b1 : 1;
  unsigned int b2 : 1;
  unsigned int b3 : 1;
};
int main()
{
  A a; a.a1 = 1; a.a2 = 1; a.a3 = 1;
  std::cout << a.a1 << " " << a.a2 << " "
            << a.a3 << "\n";
  B b; b.b1 = 1; b.b2 = 1; b.b3 = 1;
  std::cout << b.b1 << " " << b.b2 << " "
            << b.b3 << "\n";
}
```

Listing 7

When using this method with an integer, you would normally define a set of constants, one for each flag value. The value of each constant has its particular flag bit set to 1, all other bits set to 0, so the constant represents the flag being turned on. You then use normal binary operations to turn on the flags and to test if they are turned on or not.

You can do the same when using a `std::bitset`, but you also have the option of accessing individual bits using the `[]` operator or the `test()` function, which take the position of the bit in the bitset to check and return true if it is set to 1, else false.

One advantage of using a flag variable is that the user just has to turn on the flags they want, and all the others default to off. On the other hand, it is awkward to explicitly say that a flag is turned off, should you wish to do so.

If you find a flag needs more than two values, you just need to increase the size of the field and adjust the constants appropriately. If you are using a bitset, the direct bit access through `[]` or `test()` could not be used in this case.

A useful trick in case this might happen is to not make bitfields adjacent to each other when they are first defined. For instance with four flags in a four byte integer, set the fields up as the lowest bit in each byte. That way if you do need to increase the number of values represented by a flag, you won't have to change any of the constants that don't relate to that flag.

Using a flags structure

This method uses a structure to hold the flags. The structure members can be either `bool`s or single-bit bitfields.

If using this method, you can directly set the individual fields to turn the flag on or off. For the bitfields version you would usually use `0` for off and `1` for on.

If using the bitfield version you need to define them as unsigned, as they are just one bit wide. If they are defined as signed then setting the value to `1` will end up with it being treated as `-1`. Listing 7 illustrates this. Checking the output, you can see that structure with `int` fields outputs `-1` for each one, while the structure with `unsigned int` values outputs `1` for them:

```
-1 -1 -1
1 1 1
```

If you don't want to create a variable of the structure type to pass to a function you can use an initializer-list as the parameter and the structure will be created for you. Listing 8 shows examples of both types.

```
#include <iostream>

struct BitFlags
{
  unsigned int flag1 : 1;
  unsigned int flag2 : 1;
  unsigned int flag3 : 1;
};
struct BoolFlags
{
  bool flag1;
  bool flag2;
  bool flag3;
};
void fbit(BitFlags flags)
{
  std::cout << flags.flag1 << " " << flags.flag2
            << " " << flags.flag3 << "\n";
}
void fbool(BoolFlags flags)
{
  std::cout << std::boolalpha << flags.flag1
            << " " << flags.flag2 << " "
            << flags.flag3 << "\n";
}
int main()
{
  BitFlags bitflags;
  bitflags.flag1 = 0;
  bitflags.flag2 = 1;
  bitflags.flag3 = 0;
  fbit(bitflags);
  fbit({1, 0, 1});

  BoolFlags boolflags;
  boolflags.flag1 = false;
  boolflags.flag2 = true;
  boolflags.flag3 = false;
  fbool(boolflags);
  fbool({true, false, true});
}
```

Listing 8

The advantage of setting up a variable before passing it to the function is that someone reading the code later can see exactly which flags are being set, whereas when using an initializer list they have to know what the structure looks like to know which flags are being set.

When using the bitfield version, if you need to extend a field to hold more than two fields you can just extend its width. For the `bool` version, you can just replace the `bool` with a different type.

Using enums

This method simply uses enums with two enumerators defined. Using appropriate names means the values can be self-documenting. Either scoped or unscoped enums can be used.

Unscoped enums have the disadvantage that the enumerators are defined in the scope enclosing the enum, so you cannot have the same enumerator name in two enums that will be used at the same time. On the other hand, it does mean that the enumerators can be used with no qualification.

For scoped enums the enumerators are defined in the scope of the enum, so two enums can have enumerators with the same name if that makes sense. This does mean that they have to be qualified with the enum name when used.

If an unscoped enum is passed as a function parameter that expects an integer, the value in the enum variable will be converted to an integer. This does not happen for a scoped enum – no conversion takes place.

```
#include <iostream>
#include <string_view>

enum class RedBlue { Red, Blue };
std::ostream& operator<<(std::ostream& ostr,
  const RedBlue conv)
{
  ostr
     << (conv == RedBlue::Red ? "Red" : "Blue");
    return ostr;
}
void PrintArgs(const std::string& s,
  RedBlue to_uc = RedBlue::Red)
{
  std::cout << "Called PrintArgs(string, RedBlue)
    with (" << s << ", " << to_uc << ")\n";
}
void PrintArgs(const std::string& s, short len,
  RedBlue to_uc = RedBlue::Red)
{
  std::cout
    << "Called PrintArgs(string, short, RedBlue)
    with (" << s << ", " << len << ",
    " << to_uc << ")\n";
}
int main()
{
  std::cout << std::boolalpha;
  PrintArgs("Abc");                    // 1
  PrintArgs("Abc", RedBlue::Blue);     // 2
  PrintArgs("Abc", 2);                 // 3
  PrintArgs("Abc", 2, RedBlue::Blue); // 4
}
```

Listing 9

Listing 9 is the scoped enum equivalent of Listing 3. This version compiles with no ambiguous function calls detected, and if you run the resulting program you will see that the `PrintArgs` functions called in each case are the correct ones. The output for the program is shown in Listing 10.

C++20 and using enum

The point was made above that when using scoped enums you need need to precede the enumeration name with the scoped enum name. This has been addressed in C++20 with the addition of the `using enum` construct to pull all the names in the named enum into the current scope.

A brief description of this facility can be found at [CppRef2] – look for *Using-enum-declaration*. The facility was added by P1099r5 [P1099r5], and a fuller description of it can be found by reading that (brief) paper.

As of the time of writing (April 2021), the C++20 language features pages for GCC (at version 11) and MSVC (at VS 2019 16.4) show this feature as being implemented. The equivalent Clang page shows this feature has not yet implemented.

Problems versus suggested alternatives

In this section, we will check if the suggested alternatives solve any of the problems outlined.

```
Called PrintArgs(string, RedBlue) with (Abc, Red)
Called PrintArgs(string, RedBlue) with (Abc, Blue)
Called PrintArgs(string, short, RedBlue) with
  (Abc, 2, Red)
Called PrintArgs(string, short, RedBlue) with
  (Abc, 2, Blue)
```

Listing 10

The meaning of true

Splitting into two functions works, as long as you use sensible names for the new functions.

Using a flags variable mostly works, as long as you use sensible names for the constants representing the flags. As noted in the description it is not as simple to explicitly indicate the flag is turned off.

Using a flags structure works as long as the structure members have sensible names. Unlike the case above, it is also simple to set the correct member to indicate the flag is turned off.

Using enums works as long as the enumerators have sensible names.

All bools look alike

Splitting into two functions can work if you only have two `bool` parameters, but any more than that and it becomes impractical.

Using a flags variable or a flags structure works as we no longer have multiple variables.

Using enums works because all enums are distinct from each other.

Conversions to and from bool

Splitting into two functions works for the parameter that has been removed, although any remaining `bool` parameters being passed could still suffer from conversion.

Using a flags variable held in an integer can undergo all the normal integer conversions, so it does not solve this problem.

Using a flags variable held in a `std::bitset` is better because you cannot assign an integer to a `bitset` or vice versa. Note however that you can initialize a `bitset` with an integer, so passing an integer to a function when it expects a `bitset` will use the integer to initialize the `bitset`.

Using a flags structure works as structs do not implicitly convert to anything else.

Using unscoped enums partially solves the conversion problem. An integer or floating-point type cannot be converted to the enum type implicitly[4]. On the other hand, values of the enum type are implicitly convertible to integral types.

Using scoped enums solves the implicit conversion problem completely[4, 5].

More than two values

Splitting into two functions could solve this problem as you just need to add a function for each new value. If your functions are handling two conditions then you'll need a new function for each possible new combination, so it may be worth redesigning at this point to stop the number of functions from exploding.

Using a flags variable works as you can just increase the number of bits each flag uses to represent its value. You do have to be careful that the constants for different flags don't overlap each other.

Using a flags structure works by allowing you to easily determine the size of each member of the structure. Unlike for the flags variable above you do not need to keep fields separated manually.

Using enum types works as you just need to add new enumerators for the new values. If using unscoped enums you have to be careful not to create any name clashes with enumerators belonging to other unscoped enum types.

Potential disadvantages with suggested alternatives

This section will discuss some potential disadvantages with the suggested alternatives, and hopefully show that they are either not a problem or else the pros outweigh the cons.

4. Although you can use an explicit cast, such as a `static_cast`, to convert integer, floating-point, or enumeration values to an enum type, whether unscoped or scoped.
5. Scoped enum values can be converted to integer values using a `static_cast` though.

More verbose code

All of the alternatives suggested make the code more verbose. For most of them this is simply a case of replacing code like

```
if (x) { ... }
```

with an explicit test like

```
if (x == value) { ... }
```

It could be argued that making the test explicit does make the code more self-documenting, so should not be seen as a disadvantage.

The alternative using constants to define flag bits, either in an integer or a **`std::bitset`**, does have code that looks more complicated, as you have to use a binary 'and' to isolate the flag bit and test if it is set, like

```
if ((x & flagbit) == flagbit)
```

or if you are happy to rely on the implicit conversion to **`bool`** you can use

```
if (x & flagbit)
```

instead. Neither is as clear as the simple test against a value. On the other hand with the **`std::bitset`** you can use the **`[]`** operator or **`test`** function to check a bit at a position.

Namespace pollution

All of the suggested alternatives insert new entities into the current namespace, whether functions, constants, or types. All of those entities introduce new names into the current namespace which wouldn't need to exist if you just used **`bool`** values. This will cause problems if they clash with any names already in that namespace.

Of course, this isn't specific to this case – it occurs whenever you add new entities to a scope, so do whatever you normally would to get around it.

An easy solution is to add the new entities in their own namespace. This does mean that the names need the namespace as an extra qualifier, but you can use a **`using`** declaration to bring the name into the current namespace. If the new entities are only used in a single `\*.cpp` file you can put them in an anonymous namespace in that file and you won't even need the extra qualifier.

Size and speed of compiled programs

A common concern when using the alternatives is that the code will be larger and/or slower than when using **`bool`**s. This should not be a concern as modern compilers are intelligent enough to recognise what the code is doing and optimizing it appropriately.

Sample code to show this can be found on [BitBucket], [GitHub], or [GitLab], depending on your preferred supplier. The various `\*.cpp` files each demonstrate one alternative, except the `bools.cpp` one which shows the original form with **`bool`** variables.

The `find-medians.sh` shell script in that directory runs all the programs and captures the runtimes, then works out the median and mode runtimes for each one. Running this script on my main machine gives the runtimes shown in Table 1 for code optimized with **`-O3`**.

Test Case	Median	Mode
Bools	387	387
Bitset with constants	387	388
Bitset with indexing	387	388
Scoped enum	387	387
Unscoped enum	387	387
Functions	382	382
Integer flags	387	387
Struct with bitfields	387	386
Struct with bools	387	387

Table 1

As can be seen, the runtimes for the optimized programs are virtually identical for all the programs. This shows that you don't lose much if any speed when using the alternatives.

As far as code size is concerned, for the optimized code the program sizes range from 17160 bytes for `functions.opt` to 17320 for `bitset-consts.opt`. The `bools.cpp` file is 17272 bytes. So there is little difference in code size either.

So no more bools then?

It might seem that this article is saying that you shouldn't use **`bool`** values in your programs at all. This is not the intention.

One target is the use of **`bool`**s in what might be termed long-range code. What do we mean by **`long-range`** code?

Calling a function is long-range, as you are leaving the current function's scope and entering the called one. You should think carefully before using **`bool`**s as parameters of functions. As this article has tried to show, there are alternatives which can be both safer and clearer, with little or no loss of program speed.

Code in a single function could also be considered long-range if the whole usage cannot be seen on a single screenful of code[6]. Using a **`bool`** to store the result of a logical operation which is used in the immediately following code is fine, as it's obvious what is going on. Even if the value is only used once, if it simplifies a condition expression it can still be valid to do so.

Another target is the use of **`bool`** for class member variables. This is an ideal case for using one of the alternatives, especially enums. Classes provide their own scope, so the potential for namespace pollution is immediately reduced. And if the member variable is private (as they should normally be), all the code using it will be written by the class maintainer, so the users of the class won't have to handle it at all.

So in summary, if the use of the **`bool`** would be obvious from the immediate context of the code, it is fine to use it. In all other cases, consider using an alternative. This article provides several such alternatives as a starting point. ■

References

[BitBucket] https://bitbucket.org/dustycorner/articles/src/master/replacing-bool-values/testcode

[CppRef1] https://en.cppreference.com/w/cpp/language/overload_resolution

[CppRef2] https://en.cppreference.com/w/cpp/language/enum

[GitHub] https://github.com/dustycorner/articles/tree/master/replacing-bool-values/testcode

[GitLab] https://gitlab.com/dustycorner/articles/-/tree/master/replacing-bool-values/testcode

[P1099r5] Gašper Ažman and Jonathan Müller, 'Using Enum', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1099r5.html

[Preussner] 'Dimensional Analysis in Programming Languages', https://gmpreussner.com/research/dimensional-analysis-in-programming-languages

Wikipedia-1] UUencode: https://en.wikipedia.org/wiki/Uuencoding

[Wikipedia-2] Base64: https://en.wikipedia.org/wiki/Base64

6. And by a single screenful of code I don't mean using huge monitors and small fonts to get 150+ lines of code on a screen at a time. Think more like 40 to 50 lines maximum, so a quick scan up and down is easy to do.

How We (Don't) Reason About Code

Reading code is time consuming. Lucian Radu Teodorescu takes it one step further and asks how we reason about code.

I think it's well known in the software industry that we spend more time reading code than writing it. One person who helped popularise this idea is Kevlin Henney. See Figure 1 for two tweets posted by Kevlin arguing this idea [Henney20]. At that point, I also argued that we should probably be using the terms 'Reasoning' or 'Understanding' instead of 'Reading' to reflect more accurately what software developers do when they sit in front of the code.

Five minutes after posting my tweet, one question found its way into my head, and then dark thoughts followed. What does it mean to *Reason* about code? After letting these thoughts multiply and form connections in the back of my mind for several months, this article tries to provide an answer to that question.

We will try to find out how proper reasoning about code should be undertaken, and then we will inspect some very interesting implications of our ability to reason about code.

What would proper reasoning mean?

According to the Merriam-Webster dictionary [MW], *Reason* means (selected entries):

- a rational ground or motive
- a sufficient ground of explanation or of logical defense; especially : something (such as a principle or law) that supports a conclusion or explains a fact
- the power of comprehending, inferring, or thinking especially in orderly rational ways

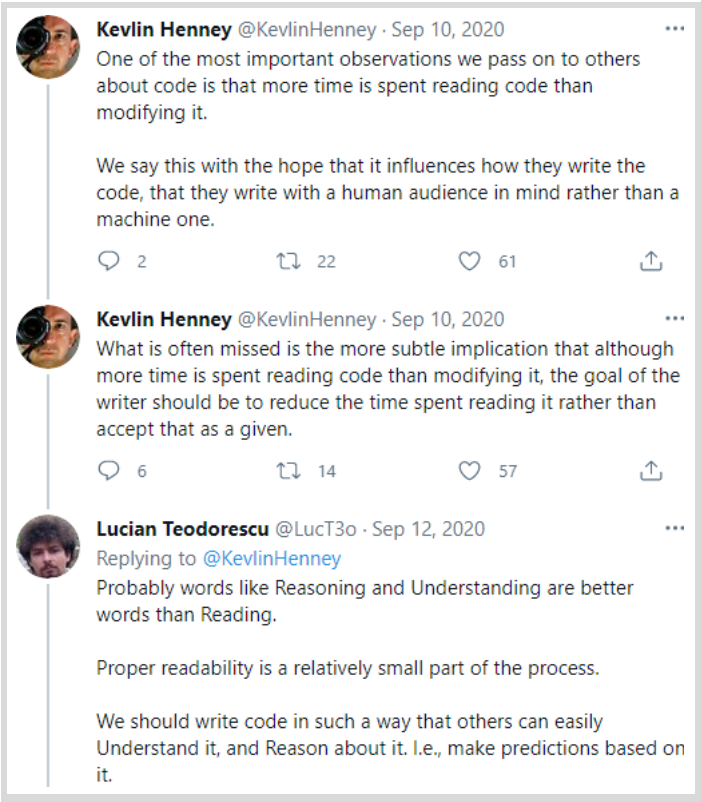


Figure 1

There are other, softer, meanings of the word *Reason*, but, as we are in an engineering discipline, we should rely on science; so we will take the most scientific definition we can get.

For this reason, for the rest of the article I will assume that *reasoning about the code* means to comprehend the code, its exact meaning and its limitations, and to be able to properly draw conclusions about its implications – all done in a logical/mathematical way.

For example, if `i` is an integer, then reasoning about the statement i++ should include:

- whether the expression is valid
- what would be the value of `i` after the increment
- what would be the returned value
- when the new `i` will be higher than the old `i` (not always true!)
- what happens when we reach the maximum value allowed for the integer type

There are multiple ways of formalising this analysis. One common method used in Software Engineering is by using Hoare rules [Hoare69]. In this formalism, for each command/instruction C we have a set of pre-conditions $\{P\}$, and a set of postconditions $\{Q\}$ – typically represented as a triple $\{P\}C\{Q\}$. If the preconditions $\{P\}$ hold, then after executing C the postconditions $\{Q\}$ will also hold.

Therefore, reasoning about a code C means finding out all the properties that belong to the set $\{Q\}$. And, of course, by *finding out* we should mean *proving*. That is, *reasoning about code* should mean a process of mathematically proving the implications of the code.

Please note that, in this context, the preconditions $\{P\}$ and postconditions $\{Q\}$ refer to mental activities, and have little to do with the definition of the programming language. They refer the set of thoughts that are in our heads, if we were to reason about these mathematically.

This type of reasoning is something that we don't do daily when programming, but let's analyse what it entails.

An analysis of for loops

Let us try to analyse the two `for` loops shown in Listing 1 and Listing 2. The intent of both code fragments is to print the content of `vec`.

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

reasoning about the code means to comprehend the code, its exact meaning and its limitations, and to be able to properly draw conclusions about its implications

Before this, let's make a few simplifying assumptions:

- we assume all the constructs work as specified in the language standard (I'm assuming C++ in this case)
- we assume all the function calls behave like they are intended to, without us needing to analyse the implementation (i.e., using abstractions simplifies reasoning)
- we indulge ourselves not to be extremely formal

In other words, we want to avoid becoming like *Principia Mathematica* [Whitehead10], where the equality 1 + 1 = 2 was proved only at page 379 of the book.

Let's also assume the following preconditions are true at the point where the code appears:

- P_1: we are targeting a conformant C++11 compiler
- P_2: vec has type `std::vector<int>`, with usual semantics
- P_3: `int` denotes a 32-bit integer
- P_4: the size of the vector is less than 1000 elements
- P_5: function `print` is defined with the following signature: `void print(int)`

We will attempt to list all the postconditions for the two cases, without attempting to prove all of them.

```
for (auto el: vec)
  print(el);
```

Listing 1

```
for (int i=0; i<vec.size(); i++)
  print(vec[i]);
```

Listing 2

Ranged for loop

Postcondition Q^1_1. The syntax of the code in Listing 1 is valid. (This follows from the language specification. This should have been broken down in multiple items, but let's not try to be overzealous)

Postcondition Q^1_2. The range expression (`vec`) is semantically valid.

Postcondition Q^1_3. The `for` loop, if valid, would iterate over all the elements of the given vector. (By the meaning of `for`-loops as defined by the C++ standard.)

Postcondition Q^1_4. The range declaration (`auto el`) is semantically valid. (From the language rules, and considering also that range expression is valid.)

Postcondition Q^1_5. `el` is a variable of type `int`.

Postcondition Q^1_6. The call `print(el)` is semantically valid (and prints the value of `el`).

Postcondition Q^1_7. The `for` loop is semantically valid. (Consequence of the other preconditions.)

Postcondition Q^1_8. The `for` loop will print all the values in the given vector. (Consequence of Q^1_3 and Q^1_7.)

We were not extremely formal, but we managed to prove in 8 steps (more accurately, with 8 postconditions) the semantics and the behaviour of the `for` loop in Listing 1. Thus, reasoning about the code from Listing 1, with the above assumptions, should entail finding and proving these 8 postconditions.

Please note that if we had more complex types in the vector, or we used a reference for the `el` variable, or if we had had to do implicit conversions, the analysis would have been more complex.

Classic for loop

Let's now turn our attention to the `for` loop in Listing 2.

Postcondition Q^2_1. The syntax of the code in Listing 2 is valid.

Postcondition Q^2_2. The init statement (`int i=0`) is semantically valid. (This should have been broken down, as we should have proved first that the constant `0` is compatible with type `int`, but again, let's not be too overzealous.)

Postcondition Q^2_3. `i` is a variable of type `int`.

Postcondition Q^2_4. The initial value of `i` is `0`.

Postcondition Q^2_5. The expression `vec.size()` is valid, and has the type `size_t` (this again is more complex than stated here, as the return type is a dependent type – but let's keep things as simple as we can).

Postcondition Q^2_6. The expression `i<vec.size()` is valid, and has the type `bool`. (This comes from the fact that there is an operator `<` defined between `int` and `size_t`, with the usual meaning.)

Postcondition Q^2_7. The expression `i++` is semantically valid.

Postcondition Q^2_8. The expression `vec[i]` is semantically valid, well-defined and has the type `int`, assuming `i` is in range 0..*N*, where *N*-1 is the number of elements in the vector. (To prove this one needs to take into consideration the implicit conversion from `int` to `size_t`, consider the multiple function specialisations and consider the constness of `vec`.)

Postcondition Q^2_9. Assuming that `i` is in range 0…*N*, where *N*-1 is the number of elements in the vector, then the statement `print(vec[i])` is semantically valid and well-defined.

Postcondition Q^2_{10}. Assuming that `i` is in range 0…*N*, where *N*-1 is the number of elements in the vector, then the `for` loop from Listing 2 is semantically valid and well-defined.

Now, at this point, the reader might think that we are done; compared to the `for` loop from Listing 1, it's not even that bad: 10 postconditions instead of 8. But we haven't dealt yet with the trickiest part: making sure that the code does what we want to do (print all the values in the vector), i.e., something similar to Q^1_8. OK, so let's tackle that.

Postcondition Q^2_{11}. Unless `i` reaches `INT_MAX`, then `i` can only monotonically grow. (To prove this we must notice that the only operation that changes `i` is `i++`, and this can only increase the value of `i`, but only if `i` hasn't the value of `INT_MAX`.)

oftentimes, small details can have huge, and potentially disastrous, consequences

Postcondition Q^2_{12}. The value of `i` can only be positive if `i` never becomes I`NT_MAX`. (Coming from the previous postcondition, coupled with the initialisation value.)

Postcondition Q^2_{13}. If `i` is never `INT_MAX` then `i` cannot be greater than *N*, where *N* is the size of the vector (but can be equal to *N*).

Postcondition Q^2_{14}. The value of `i` is between 0 and *N*. (This results from the previous postcondition, and from P_4).

Postcondition Q^2_{15}. The body of the loop will be called with values of `i` between 0 and *N*-1. (Once `i` becomes *N*, it will not be called any more; results from the semantics of the `for` loop.)

Postcondition Q^2_{16}. The body of the loop will be called each time with a new value of `i`, monotonically increasing. (People may forget this, but this is a crucial fact that makes the `for` loop work.)

Postcondition Q^2_{17}. The expression `vec[i]` is well-defined. (This can be proved from Q^2_{15} and Q^2_{8}.)

Postcondition Q^2_{18}. The statement `print(vec[i])` is well-defined.

Postcondition Q^2_{19}. The expression `vec[i]` will represent during the iteration all the values from the input vector. (This results from the semantics of the vector, Q^2_{16} and Q^2_{15}; it is similar to Q^1_{2}.)

Postcondition Q^2_{20}. The `for` loop in Listing 2 will print all the values in the given vector.

Phew, we reached the end. We showed that the code in Listing 2 is syntactically correct, is semantically correct, and it performs the intended function (prints all the values in the given vector). This time it took us 20 steps to come to this conclusion, compared to 8 steps needed for Listing 1. Although I was clearly trying to make a point, I tried not to exaggerate the number of postconditions. I tried to follow in my head the actual steps to prove mathematically the functioning of the two code snippets, and then wrote the lemmas that would have appeared.

Subtlety matters

Some readers might argue that we've gone to great lengths to prove something that is obvious to most programmers. And there is some truth in that – we'll cover that a bit later. But, oftentimes, small details can have huge, and potentially disastrous, consequences.

Take a look, for example, at this code:

```
for (size_t i=vec.size()-1; i>=0; i--) {...}
```

It looks perfectly normal, but it's utterly wrong; I can't count how many times I've been bitten by code like this. Or, for the same type of code, forget to subtract 1 from the size of the vector, or to perform a strict comparison with zero. That entirely changes the behaviour of the code.

Also think of cases in which the size of the vector can overflow the capacity of the index type.

There are also variants that do not change the behaviour of the code, and yet they look different enough. One simple example is changing `i++` to `++i` or `i+=1`. These changes have a non-zero mental cost when reading the code, which proves that our brain has to do a bit more processing to properly or improperly 'reason' about the code.

A new metric for complexity

The whole exercise we undertook with the two loop structures yielded the conclusion that the classical 3-part `for` loop is more complex than the ranged-`for` loop. It is more complex in terms of number of steps to reason about it, and it is also more complex considering the number of preconditions we have to use from that point on when reading the code inside the for loop (postconditions turn into preconditions).

But, not only can we say that one is more complex than the other, we can also quantify the difference in complexity. We can obtain this by the following formula:

$$ReasoningComplexity(\{P\}C\{Q\}) = |Q|-|P|$$

In plain English, the reasoning of complexity of the code is the number of post-conditions added by that piece of code that were not present in the initial assumptions (we assume that the assumptions before the code are also kept). Or, informally, reasoning complexity is the number of *ideas* we have to entertain while trying to mathematically reason about the code.

This metric can give a better indication of the complexity of a code than cyclomatic complexity [McCabe76] or Halstead complexity [Halstead77]. Cyclomatic complexity completely avoids all the code complexities that do not involve branches. In other words, a very complex calculation without branches is equally complex to a simple `x=0` assignment. On the other hand, the Halstead complexity measures the complexity purely from a syntactic perspective, without including any semantic elements.

In ideal conditions, the metric we defined above would measure how much effort does a person need to spend to (properly) reason about a particular code.

Pattern matching and limitations for reasoning complexity

The above reasoning complexity has one major flaw: it assumes that programmers properly reason about the code, in the mathematical sense. And, that is far from the truth. I think in this article it's the first time I've tried to reason about a piece of code, and even here, I took many shortcuts.

Instead, programmers do something more like pattern matching on the code. That is, most C++ programmers have seen code similar to the one in Listing 2, and they know what the code does without going over the 20 steps of post-conditions. And, I think that would apply to many programmers who haven't programmed in C or C++ too, as the ideas about classic `for` loops are so common in programming.

I like to describe this process in the following way: a person sees a code fragment, and that code fragment starts to *resonate* with other code that the person has seen before. One can immediately see the similarities and the differences with other code, and one can immediately (but not always logically) draw conclusions. Most of these conclusions can be true, even without making once a proper reasoning.

it tends to be easier for people to read code in the order 'variable predicate constant' rather than 'constant predicate variable'

More generally, a person makes sense of new experiences by resonating with past experiences. And this doesn't apply only to software. One sees a door, one knows that it may be opened; one sees a chair, and knows that someone might sit on it; one sees an animal like a wolf, one knows that it might do harm.

The fact that our senses are cheating on us is a well-known fact, and yet, every day we rely on our senses to understand what's happening around us. Pure rationalism doesn't get us very far (i.e., you can't be really sure of a single thing), so this type of incomplete reasoning is the only way for us humans to live.

This way of incomplete reasoning we also apply to code, for better or for worse.

A more practical complexity measure

To make our reasoning complexity more practical, we somehow need to incorporate past experience in our formula. Let's assume that for a given person X, we have a set of Hoare triplets forming the past experience of that person: $PE(X)=\{\{P_i\}C_i\ \{Q_i\}\}$.

With this, the complexity associated for person X to infer conclusions (not mathematical, but rather in a more empirical approach) about a given code is given by:

$$InferenceComplexity(X,\{P\}C\{Q\}) = min(dist(\{P\}C\{Q\},\{P'\}C'\{Q'\}) \mid \{P'\}C'\{Q'\} \in PE(X)$$

with the function *dist* defined something like:

$$dist(\{P\}C\{Q\},\{P'\}C'\{Q'\}) = |Q| - |P \cup Q'|, \text{ for } C \approx C'$$

Now, the above definition is somewhat ambiguous, and that's almost intentional. Having a non-ambiguous formula here would mean that we thoroughly understand how the brain works, which is far from the truth.

The main idea is that, when trying to define the complexity for a given code, we need to consider also the previous experience of the programmer reading the code. The more the programmer has seen and made inference about that type of code, the simpler the code would be. This is why, for most programmers, the code from Listing 2 is roughly as complex as the code from Listing 1 – we've seen that code pattern many times, and we know what it means.

A few takeaways and examples

Now that we have two complexity measures, let's pick some random examples and see how they can be used for analysing various situations.

There is no one single style to rule them all

What I find to be easy to understand may not be for the next programmer. And vice versa. We constantly have to be aware of our biases.

We can have some general reasoning why certain things might be simpler than other things, but we should always remember that these are context-dependent.

One good example for this is coding styles (read formatting). I've seen many passionate arguments from a lot of people, arguing that a certain code style is better than another – some coming from people I respect a lot. But, after a given point, every style choice will be received well by some programmers and painfully by others.

In this category, a good example in the C++ community is the east-const vs west-const debate (I won't even post a link to it).

My takeaway from this is that I would get familiar with multiple styles of programming, looking at the essential properties of the software, and not at the code style.

Complexity for smaller operations

Apart from the style issues, there are certain practices which seems (to me) very odd. People sometimes add more complexity to the code (in the absolute mode) in the hope of making it simple. This seems a bit counter-intuitive.

Please see Listing 3. In the first `if` condition, instead of just checking the value of `my_bool`, we make a new comparison with constant `true`. This is clearly more complex than just the code `if (my_bool)`, at least for an analytical mind.

The second `if` clause in Listing 3 is called Yoda conditions [WikiYoda], a reference to Yoda from the Star Wars film, who frequently reverses words. In many languages, people express simple statements in the form 'subject verb complement'; for example "The grass is green"; we would not typically say "green the grass is". I think most English-speaking people can understand what "green the grass is" means, but, at the same time, it would be against people's expectations, and would add extra mental processing (small, but it's there).

Translating this into software, it tends to be easier for people to read code in the order 'variable predicate constant' rather than 'constant predicate variable'. There were historical reasons for why the Yoda conditions were preferred, but I strongly suggest using the proper tools to solve those problems, and not change the coding style, adding mental effort for most `if` statements.

But then again, this is a styling issue.

Monads

We can roughly divide the programming world in two: those who know about monads and love them and those who hate monads as they consider them completely abstract. (There is also a joke that most programmers read

```
bool my_bool = ...;
if (my_bool == true) ...

int my_int = ...;
if (0 == my_int) ...
```

Listing 3

We, as programmers, tend not to reason in the mathematical sense, but rather infer conclusions based on previous experience

about monads every couple of months, and then immediately forget about them – but let's ignore this category.)

For functional programmers, using monads is a day-to-day job, so the inference complexity for reading code with monads is lower than a person who doesn't use monads frequently.

In reality, monads are simple concepts, and they are heavily used by imperative programmers as well, just that most of the time they don't pay attention to it. I'll try a very short pitch to the C++ developers.

A monad is a type wrapper (read template type), plus a *type convertor* function (called *unit* or *return*) that transforms a value into a value of the wrapped type, and another function called *combinator* (sometimes named *bind* or *flatmap*) that transform that monadic type.

The `std::optional<T>` and `std::vector<T>` are well known monadic types; the constructors for these types that take `T` as parameters can be considered the type convertor; functions with the declaration shown in Listing 4 (or similar) can act as combinators.

The bottom line is that, the reasoning complexity is actually lower than the perceived complexity. So, the inference complexity can also have negative feedback loops. Interesting…

Functional programming vs OOP

After discussing monads, it makes sense to discuss functional programming vs OOP.

It would probably make sense to start an analysis on the reasoning complexity for common structures in the two paradigms, but that's too far beyond the goal of this article. But, I think it's safe to say that they probably have the same reasoning complexity.

This means that OOP people have a bias against the complexity coming from functional programming, and functional programming people will have a bias against the complexity coming from OOP programs.

At least for me, with the two metrics of complexity in front of me, I now realise that the major difference between the two paradigms is the perceived complexity. The way to resolve this difference is by learning: people need to get accustomed to the other style of programming.

As linguistics would say, learning a new language means learning a new way to think.

```
template <typename T1, typename T2>
optional<T2> bind(const optional<T1>& x,
  function<optional<T2>(T1)> f);

template <typename T1, typename T2>
vector<T2> bind(const vector<T1>& x,
  function<vector<T2>(T1)> f);
```

Listing 4

Conclusions

The article tries to analyse what reasoning about code properly means. Formally, we argue that reasoning means deriving the implications of the code. Based on this, we define a new complexity metric, that essentially counts the number of new post-conditions that were added after executing that code.

But this metric doesn't seem to properly apply in practice. We, as programmers, tend not to reason in the mathematical sense, but rather infer conclusions based on previous experience; we can be occasionally wrong, but it tends to work really well in practice. Thus, we hint at another complexity metric (very informally defined) that considers the experience of the programmer.

Both complexity metrics tend to relate to the effort that the programmer makes (or is assumed to make) when trying to make sense of the code.

This opens the discussion of assessing the complexity of code, from two directions, which are sometimes contradictory. From one perspective, a reasoning complexity tries to have an absolute, independent position for assessing the complexity of the code. This can provide a good basis for discussion, but it often may be impractical. On the other side, the perspective of using inferred complexity tries to look more from the programmer's point of view, but loses objectivity. This can be used to discuss complexity related to social issues (coding styles, paradigms, etc.).

But, maybe more importantly, as the inferred complexity is learned, we may learn to rely less on it, and use the reasoning complexity. That would provide us better opportunities to reason about reasoning about code. ■

References

[Halstead77] Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977

[Henney20] Kevlin Henney, One of the most important observations…, Twitter, https://twitter.com/kevlinhenney/status/1303989725091581952?s=21, 2020

[Hoare69] C. A. R. Hoare, An axiomatic basis for computer programming. *Communications of the ACM*. 12 (10): 576–580, 1969.

[McCabe76] T. J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering* (4), 1976

[MW] Merriam-Webster, Reason, https://www.merriam-webster.com/dictionary/reason, 2021

[WikiYoda] Wikipedia, Yoda conditions, https://en.wikipedia.org/wiki/Yoda_conditions

[Whitehead10] Alfred North Whitehead; Bertrand Russell, Principia Mathematica, vol 1/2/3 (1 ed.), Cambridge: Cambridge University Press, 1910/1912/1913, https://quod.lib.umich.edu/cgi/t/text/text-idx?c=umhistmath;idno=AAT3201.0001.001

Out of Control

An essay on paradigms, refactoring, control flow, data, code, dualism and what Roman numerals ever did for us. Kevlin Henney takes us on a whirlwind tour.

Looking at something from a different point of view can reveal a hidden side. With physical objects this hidden side can be literal, hence why technical drawings of three-dimensional objects are often made using a multi-view projection, such as a plan view and elevation views. With abstract concepts the hidden side is more figurative. In software architecture, for example, view models, such as 4+1 [Kruchten95] or ODP viewpoints [ODP], can be used to bring different concerns of a system into focus, such as user interaction, governance, behaviour, code structure, information model, physical distribution, infrastructure, etc.

Numeral systems also have this quality. Changing how numbers are represented doesn't change the numbers represented, but it does change how we think about them. Thinking about integers in binary, for example, reveals different patterns (and failures) than when we think about them in decimal. Binary also chunks more easily into hexadecimal. Thinking about RGB triplets makes more sense in hex than in decimal. And so on. Changing base can reveal new possibilities [Deigh17].

The same shift in perspective is also possible moving from a positional system, such as the common Hindu–Arabic system for decimals [Wikipedia-1], to one based on abbreviations of values of different magnitudes, a sign–value system (*sign* in the semiotic sense of *symbol* rather than as a positive/negative indicator) [Wiktionary], such as Roman numerals [MathsIsFun]. And somewhere between these two numeral systems – and more globally – we find the abacus, a bi-quinary coded decimal system [WordFriday] (like Roman numerals), built on a simpler sign system, but organised positionally (like the Hindu–Arabic system) to enable rapid arithmetic.

The Roman numeral system was used across Europe before the widespread adoption of the Hindu–Arabic system. It was perhaps more uniformly standardised following the fall of the Western Empire than during the heyday of Roman rule.

That would be the end of the history lesson if it were not for the persistence of the numeral system long after the Renaissance and into the modern day. The most likely places you will encounter Roman numerals these days include old buildings, analogue clock faces, sundials, chords in music, copyright years for BBC programmes and coding katas for programmers [Wikipedia-2].

The standard form can comfortably represent 1 to 3999 (I to MMMCMXCIX), although sometimes 4000 and beyond can be expressed (with 4000 as MMMM); numbers beyond this range would typically have been written as words, and would often be approximated. There was no notation for zero – or even a concept of zero as a number – so in medieval texts this would have been written as *nulla* or abbreviated to *N*. And, without zero, there were certainly no negative numbers.

Kevlin Henney is a consultant, speaker, writer and trainer. His interests include programming languages, software architecture and programming practices. Kevlin loves to help and inspire others, share ideas and ask questions. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*, editor of *97 Things Every Programmer Should Know* and co-editor of *97 Things Every Java Programmer Should Know*.

```
def roman(number):
  result = ""
  while number >= 1000:
    result += "M"
    number -= 1000
  if number >= 900:
    result += "CM"
    number -= 900
  if number >= 500:
    result += "D"
    number -= 500
  if number >= 400:
    result += "CD"
    number -= 400
  while number >= 100:
    result += "C"
    number -= 100
  if number >= 90:
    result += "XC"
    number -= 90
  if number >= 50:
    result += "L"
    number -= 50
  if number >= 40:
    result += "XL"
    number -= 40
  while number >= 10:
    result += "X"
    number -= 10
  if number >= 9:
    result += "IX"
    number -= 9
  if number >= 5:
    result += "V"
    number -= 5
  if number >= 4:
    result += "IV"
    number -= 4
  while number >= 1:
    result += "I"
    number -= 1
  return result
```

Listing 1

It works, but...

The Python code in Listing 1 shows one way of converting a number to a corresponding string of Roman numerals.

Leaving aside questions of type and range validation, this code works. We can find support for this claim through reasoning, review and running against the sample of test cases in Listing 2.

If, however, we look at this code as a stepping stone rather than as an end state, it becomes an example and opportunity

```
cases = [
  # Decimal positions correspond to numerals
  [1, "I"],
  [10, "X"],
  [100, "C"],
  [1000, "M"],
  # Quinary intervals correspond to numerals
  [5, "V"],
  [50, "L"],
  [500, "D"],
  # Multiples of decimal numerals concatenate
  [2, "II"],
  [30, "XXX"],
  [200, "CC"],
  [3000, "MMM"],
  # Non-multiples of decimals concatenate in
  # descending magnitude
  [6, "VI"],
  [23, "XXIII"],
  [273, "CCLXXIII"],
  [1500, "MD"],
  # Numeral predecessors are subtractive
  [4, "IV"],
  [9, "IX"],
  [40, "XL"],
  [90, "XC"],
  [400, "CD"],
  # Subtractive predecessors concatenate
  [14, "XIV"],
  [42, "XLII"],
  [97, "XCVII"],
  [1999, "MCMXCIX"]
]
failures = [
  [number, expected, roman(number)]
  for number, expected in cases
  if expected != roman(number)
]
assert failures == [], str(failures)
```

Listing 2

So, sure, the function works, but it ain't pretty. It's the kind of code only an enterprise programmer could love. Or, perhaps, a Pascal programmer: this control-flow–heavy approach was used in *Pascal: User Manual and Report* by Kathleen Jensen and Niklaus Wirth. Given that Pascal was often held up as a language from which and in which to learn good practice this might be considered ironic.

The code for the `roman` function is very procedural in that it is relentlessly imperative and control-flow oriented. Even as a procedural solution, however, it is not a particularly good one. It is repetitive, clumsy and lacking in abstraction.

```
while number >= 1000:
  result += "M"
  number -= 1000
if number >= 900:
  result += "CM"
  number -= 900
if number >= 500:
  result += "D"
  number -= 500
if number >= 400:
  result += "CD"
  number -= 400
```

Listing 3

If, however, we look at this code as a stepping stone rather than as an end state, it becomes an example and opportunity – especially in the presence of tests – rather than a counterexample and dead end. This latitude is perhaps a generosity we should extend to most code. Unless, like Romans chiselling words and numerals onto buildings, you are setting your code into stone, it may always be best to consider code a work in progress.

> *In the eyes of those who anxiously seek perfection, a work is never truly completed – a word that for them has no sense – but abandoned; and this abandonment, of the book to the fire or to the public, whether due to weariness or to a need to deliver it for publication, is a sort of accident, comparable to the letting-go of an idea that has become so tiring or annoying that one has lost all interest in it.*
>
> ~ Paul Valéry

From duplication to unification

Recurrent structure is often a good starting point for seeing what can be abstracted. At first glance, the rhythmic stanza of a `while` followed by three `if` statements looks like a good fulcrum from which to lever a refactoring (Listing 3).

The structure of this fragment for 1000, 900, 500 and 400 is repeated for 100, 90, 50 and 40 and then again for 10, 9, 5 and 4. But refactoring based on this recurrence misses a deeper duplication and, therefore, unification.

Consider, first, what is a `while` statement? It is a statement that, governed by a condition, executes zero to many times. What, then, is an `if` statement? It is a statement that, governed by a condition, executes zero times or once. Squinted at just right, an `if` can be considered a bounded case of a `while`.

Looking at the specific numbers and operations involved, we see the `while`-only code in Listing 4 is equivalent to the previous mix of `while` and `if` code.

The newly minted `while` statements will execute zero times or once, just like their `if` antecedents. There is no change in behaviour, but there is a huge change in how we perceive the problem and the shape and nature of what we want to refactor.

We are more likely to code control flow directly than chart it, but that serves to highlight that while some things in the world of programming change, there is nothing new in letting the data do the talking

```
while number >= 1000:
  result += "M"
  number -= 1000
while number >= 900:
  result += "CM"
  number -= 900
while number >= 500:
  result += "D"
  number -= 500
while number >= 400:
  result += "CD"
  number -= 400
```

Listing 4

We now have thirteen loops that look like

```
while number >= value:
  result += letters
  number -= value
```

What matters most to the solution is the series of threshold values and their corresponding letters. This is not a control-flow problem: it is a data problem. We need data structure, not control structure (see Listing 5).

This version still qualifies as procedural, but it is more declarative than the first version, which was strictly imperative.

Data-driven approaches separate data from the code that is driven by the data, with the effect of making both intent and structure clearer. Niklaus Wirth stated that Algorithms + Data Structures = Programs [Wikipedia-3], but algorithm and data structure are not necessarily equal partners. As Fred Brooks noted, in *The Mythical Man Month* under the heading 'Representation Is the Essence of Programming':

> Sometimes the strategic breakthrough will be a new algorithm […]. Much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of the program lies. Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

```
def roman(number):
  numerals = [
    [1000, "M"], [900, "CM"],
    [500,  "D"], [400, "CD"],
    [100,  "C"], [90,  "XC"],
    [50,   "L"], [40,  "XL"],
    [10,   "X"], [9,   "IX"],
    [5,    "V"], [4,   "IV"],
    [1,    "I"]
  ]
  result = ""
  for divisor, letters in numerals:
    result += (number // divisor) * letters
    number %= divisor
  return result
```

Listing 5

```
numerals = [
  [1000, "M"], [900, "CM"],
  [500,  "D"], [400, "CD"],
  [100,  "C"], [90,  "XC"],
  [50,   "L"], [40,  "XL"],
  [10,   "X"], [9,   "IX"],
  [5,    "V"], [4,   "IV"],
  [1,    "I"]
]
def roman(number):
  result = ""
  for divisor, letters in numerals:
    result += (number // divisor) * letters
    number %= divisor
  return result
```

Listing 6

We are more likely to code control flow directly than chart it, but that serves to highlight that while some things in the world of programming change, there is nothing new in letting the data do the talking. This family of approaches, from lookup tables to table-driven code, finds expression across many paradigms and contexts – the data-driven tests for **`roman`** already shown, for example – but is often overlooked. This way of thinking is either not taught fully and explicitly or, in the case of less capable languages like Pascal, cannot always be expressed conveniently.

Tables apart

There's (a lot) more that we could do to explore the algorithmic space and paradigm shapes of the Roman numerals problem, such as transforming the loop into a fold operation (**`reduce`** in Python) or eliminating the arithmetic and expressing the solution using term rewriting (converting to unary and then using **`replace`** [Deigh17]), but for now we'll leave the control flow in place so we can take off in a different direction to explore the organisation of the code elements.

Because the **`numerals`** table doesn't change and is independent of the **`number`** parameter, we can implement invariant code motion to hoist it out of the function. (See Listing 6.)

We can add further distance to this separation by placing the definition of numerals into another source file, `numerals.py` (Listing 7), which leads to the code module being, well, just code. (See Listing 8.)

Perhaps not for this particular problem, but this is an interesting decoupling because – as long as the data's structure is consistent – it allows us to change the actual data independently of the algorithm. We are using the Python in `numerals.py` as a data language, which, in effect, makes `numerals.py` a native database. Another way of looking at the separation is that the code in `roman.py` implements an interpreter for the highly domain-specific data language of `numerals.py`.

```
numerals = [
  [1000, "M"], [900, "CM"],
  [500,  "D"], [400, "CD"],
  [100,  "C"], [90,  "XC"],
  [50,   "L"], [40,  "XL"],
  [10,   "X"], [9,   "IX"],
  [5,    "V"], [4,   "IV"],
  [1,    "I"]
]
```

Listing 7

On the dualism of data and code

This leads us to ponder – and not for either the first or the last time in the history of computer science – the distinction between code and data. We stumble across this question even in the simplest cases. How would you describe the refactoring transformations above? Many would describe them in terms of separating the data from the code. Does that mean, then, that `numerals.py` contains data but not code? It's a valid Python module that initialises a variable to a list of string–integer pairs. Sounds like code. Nothing says an essential qualification for something to be considered code is the presence of control flow.

We use the word *code* freely, referring both to anything written in a programming language and, more specifically, to code (sic) whose primary concern is algorithm and operation rather than data structure and definition. Natural language is messy like that, filled with ambiguity, synecdoche and context dependency.

If we want to be more rigorous, we could say that we have separated the code into code that abstracts operation and code that abstracts data. In other words, we are saying that *Programs* = *Code* and, given that *Algorithms* + *Data Structures* = *Programs*, therefore *Algorithms* + *Data Structures* = *Code*. This can be convenient and clear way to frame our thinking and describe what we have done. We also need to recognise, however, that it is just that: it is a thinking tool, a way of looking at things and reasoning about them rather than necessarily a comment on the intrinsic nature of those things; it is a tool for description, a way of rendering abstract concepts more concretely into conversation.

If we confuse a point of view for the nature of things we will end up with a dichotomy that feels like Cartesian dualism. Just as Descartes claimed there were two distinct kinds of substance, physical and mental, we could end up claiming there are two distinct kinds of code – code that is data and code that is operation.

When we look to hardware, compilers or the foundations of computer science, such as Turing machines, we will not find clear support or a strict boundary for such separation. The indistinction runs deep. Although we have code and data segments in a process address, these enforce negotiable matters of convention and protection (e.g., the code or text segment is often read-only). Both code and data segments contain data, but the data in the code segment is intended to be understood through a filter of predefined expectations and an instruction set. On the other hand, it is also possible to treat data in the data segment as something to execute.

The fundamental conceit of the Lisp programming language is that everything can be represented and manipulated as lists, including code. The Lisp perspective can be summarised as *Data Structures* = *Programs*. For some languages, source code is data for a compiler, that in turn generates data for a machine – physical or virtual – to execute. For other languages, such as Python, source code is a string that can be interpreted

```
from numerals import numerals
def roman(number):
  result = ""
  for divisor, letters in numerals:
    result += (number // divisor) * letters
    number %= divisor
  return result
```

Listing 8

```
source = """[
  [1000, "M"], [900, "CM"],
  [500,  "D"], [400, "CD"],
  [100,  "C"], [90,  "XC"],
  [50,   "L"], [40,  "XL"],
  [10,   "X"], [9,   "IX"],
  [5,    "V"], [4,   "IV"],
  [1,    "I"]
]"""
numerals = eval(source)
```

Listing 9

and executed more directly. We can blur that code–data boundary explicitly in our example seen in Listing 9.

The dualism we are grappling with here is not the strict categorisation of things that are intrinsically separate, but the dualism of things that are innately bound together. A yin and yang of complementary perspectives that we are forcing into competition, but that exist in a cat state as easily resolved one way as the other. Which one we see or chose is a question of observation and of desire and of the moment rather than one of artefact (Figure 1).

The rabbit–duck (or duck–rabbit…) illusion.

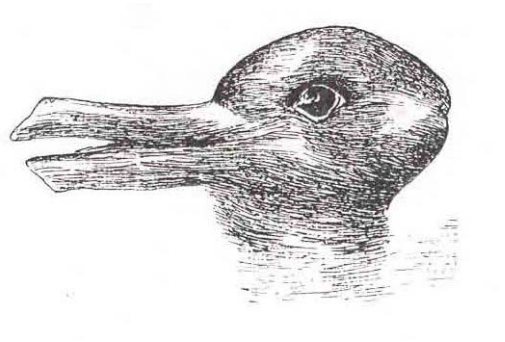


Image from Wikimedia Commons [Wikimedia].

Figure 1

The tables are turned

Instead of involving intermediate variables and conversions, we could simply replace `numerals.py` with a file that contains just the data expression, no statements (see Listing 10).

Although a valid Python expression, this is no longer a useful Python module. The data is not bound to a variable that can be referenced elsewhere. It does, however, bear a striking resemblance to a more widely recognised data language: JSON (see Listing 11).

If you had wondered why I'd avoided using tuples and single-quoted strings in the code so far, you now have your answer: looked at the right way, JSON is almost a proper subset of Python. (If you hadn't, no worries. As you were.) The relationship becomes closer and the observation truer if you replace the previous eval expression with the following:

```
eval(source.read(),
  {"null": None, "false": False, "true": True})
```

```
[
  [1000, "M"], [900, "CM"],
  [500,  "D"], [400, "CD"],
  [100,  "C"], [90,  "XC"],
  [50,   "L"], [40,  "XL"],
  [10,   "X"], [9,   "IX"],
  [5,    "V"], [4,   "IV"],
  [1,    "I"]
]
```

Listing 10

Of course, this lets more through than just valid JSON, which may make you feel – justifiably – a little uncomfortable. We can more fully acknowledge and safeguard the transformation by replacing the permissive and general eval with something more constrained and specific:

```
from json import loads
with open("numerals.json") as source:
  numerals = loads(source.read())
```

Configuration is code

Looking at the progression of code above, evolving from the first tabular version to the final JSON version, at what point in the transformation did the table stop being code and start becoming configuration?

This is, to some extent, a trick question, but that is also the point. We're not done with dualism, ambiguity and perspective. The question serves to highlight a common blind spot and oversight: configuration is code. Treating configuration as disjoint from the concept of code steers us in the wrong direction. Treating it as something other often leads to it being treated as something lesser.

What, then, is configuration? It is a formal structure for specifying how some aspect of software should run. Sounds like code. It doesn't matter whether configuration is defined in key–value pairs or a Turing-complete language, whether it uses an ad hoc proprietary binary format or a widely used and recognised text-based one, if a software system does not behave as expected, we consider it a problem. Whether that problem originated in a JSON payload, a registry setting, a database, an environment variable or the source code is not relevant: the software is seen as not working — there is a bug to be fixed.

The consequences of incorrect configuration range from the personal inconvenience of having your settings trashed when an app updates to the more costly failure of a rocket launch [Clark17]. Configuration is no less a detail than any other aspect of a software system. Its common second-class citizenship, however, causes it to be accorded less respect and visibility, leading to a high incidence of latent configuration errors [Xu] . If we consider it as code we are more likely to consider version control, testing, reviewing, design, validation, maintainability and other qualities and practices we normally confer on other parts of our codebase but may overlook for configuration.

Programming with perspective

Whether you find yourself exploring TDD with Roman numerals, playing with the Gilded Rose refactoring kata [Bache], casting an intricate set of constraints into code or trying to crack the code of a legacy logic tangle, inverting the problem with respect to data can you show you the problem and solution in ways you might not otherwise see if you only view it from the journey of control flow.

That said, although data-driven and table-lookup approaches are unreasonably effective, the message here is not that a data-centred approach is unconditionally the path of choice regardless of your situation. It is all too easy to jump from one world view to another without properly learning the lessons of either or the journey in between. If the default way that code is cast is in terms of control flow, however, viewing program structure through the prism of data structure often reveals an extended spectrum of complexity-reducing possibilities.

We should be wary of any quest for the One True™ paradigm or a silver bullet solution, and cautious of such exclusive attachment. As Émile-Auguste Chartier cautions us:

> Nothing is more dangerous than an idea when you have only one idea.

The value of paradigms, perspectives and points of view is in their multiplicity. Sometimes one offers a better frame for understanding or creation than another — and in some contexts it might do so consistently. Sometimes habit leads us to get stuck with one and neglect others. Sometimes, like binocular vision, we need more than one to make sense of a situation or to unlock a solution. ■

```
with open("numerals.json") as source:
  numerals = eval(source.read())
def roman(number):
  result = ""
  for divisor, letters in numerals:
    result += (number // divisor) * letters
    number %= divisor
  return result
```

Listing 11

References

[Bache] Emily Bache 'Starting code for the GildedRose Refactoring Kata in many programming languages' available from https://github.com/emilybache/GildedRose-Refactoring-Kata

[Clark17] Stephen Clark (2017) 'Russian official blames Nov. 28 launch failure on botched software programming' available from https://spaceflightnow.com/2017/12/30/russian-official-blames-nov-28-launch-failure-on-botched-software-programming/

[Deigh17] Teedy Deigh (2017) 'All About the Base' in Overload 138, April 2017, available from: https://accu.org/journals/overload/25/138/deigh_2364/

[Kruchten95] Philippe Kruchten (1995) 'The 4+1 View Model of Architecture' published in IEEE Software 12(6):45-50, available from https://www.researchgate.net/publication/220018231_The_41_View_Model_of_Architecture

[MathsIsFun] Roman Numerals: https://www.mathsisfun.com/roman-numerals.html

[ODP] Reference Model of Open Distributed Processing (RM-ODP): http://www.rm-odp.net/

[Wikimedia] File: Duck-Rabbit illusion.jpg: https://commons.wikimedia.org/wiki/File:Duck-Rabbit_illusion.jpg

][Wikipedia-1] Hindu–Arabic numeral system: https://en.wikipedia.org/wiki/Hindu%E2%80%93Arabic_numeral_system

[Wikipedia-2] Kata (programming): https://en.wikipedia.org/wiki/Kata_(programming)

[Wikipedia-3] Algorithms + Data Structures = Programs: https://en.wikipedia.org/wiki/Algorithms_%2B_Data_Structures_%3D_Programs

[Wiktionary] Sign-value notation: https://en.wiktionary.org/wiki/sign-value_notation

[WordFriday] Bi-quinary coded decimal (2014): https://www.facebook.com/WordFriday/posts/725954844159142/

[Xu] Tianyin Xu, Xinxin Jin, Peng Huang, and Yuanyuan Zhou, University of California, San Diego; Shan Lu, University of Chicago; Long Jin, University of California, San Diego; Shankar Pasupathy, NetApp, Inc. 'Early Detection of Configuration Errors to Reduce Failure Damage' available from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu

JOIN THE ACCU!

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

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

With full ACCU membership you get:

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

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

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

How to join

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

Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

The Sea of C You Don't Want to See

Last issue included a script. Deák Ferenc plays with the paradigm and dives into (a) deep sea (deap C).

(A digital drama in 3 acts)

Dramatis Personae

The Speaker – the one introducing the other characters, reading up ere an act, and having the general moderator role.

The Source – the one character who is't acts irresponsibly through the entire playeth, oft changes its shape, and mostly doest not collaborate very well with other actors in the scene, doest not coequal conform to standards, but since *Source* being the central person in the entire playeth, everyone tries to winneth his favours.

The Compiler – the character with a huge responsibility of trying understandeth the humor swings of the *Source* while in the same time taming the shrew in the direction of being a productive member of society without uncovering an overly abusive encroachment while being expos'd to the behavioural traits of the *Source*. Sadly, due to excess pressure, a high workload and constant propagation, the *Compiler* suffers from a light version of schizophrenia, which expresses himself in the form, yond the *Compiler* oft calleth himself ***gcc***, oft ***clang***, and oft something else, and coequal at which hour taking one persona, the *Compiler* cannot decideth in some very basic features, such as a version number so all these combinations cometh up with very amusing conversations the *Compiler* hast with the same *Source*.

Did lay the scene

(Speaker enters the scene, softly pulling the Source after himself with one handeth, while holding his headeth in dismay with the other.)

Speaker Thither is programming, and thither art programmers. Thither art most wondrous practices, thither art poor practices and thither art nay practices. And then thither is the very much obscure ordinary of code yond no-one understands.

Source And thither is me!

Speaker Contrary to the mainstream development methodologies and recommendations at which thy cometh about writing code which is easy to readeth, maintain and generally can beest pondered obeying the laws of crisp code, in this lightheart'd drama we shall explore the enshielf depths of exceptionally obscure coding disciplines at which hour one tries to writeth code which is hard to readeth, disobeys the ingraft sense, uses practices which oft can beest pondered dangerous but still worketh did bind to specific *Compiler* and platforms, and last but not least thee would not wanteth to see those imbued into thy production code base.

Source And yond is me!

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

Speaker Thee wast did warn, thee shalt not followeth the traits of the *Source*, useth thy inner force to raiseth above him and deliver for thy peers a much better version while learning from mistakes done hither.

Act 1 – The one who did nothing

Speaker *(Stands alone in the middle of the scene, a single beam of light shines down upon him)* There are programs that do nothing, lazy *Sources* wasting their life in the wast emptiness of the digital landscape, without having a meaning, except to annoy specific *Compilers*, but here, witness dear spectators an interaction of one of these.

(Source haughtily enters the scene, he wears a very finely decorated mantle, the scene lits up)

Source Here we are, born to be kings, without excess doings of precise one time useth of the keywords the C language shall present to us, dear *Compiler*, but warned be thy meant to be, not more than one of all of them shall be presented by us. And hither we art:

```
int main(long _, char __)
  {
    switch (_)
      do {
      case 0:
      default:
        for (auto float _ = sizeof(union {});
           _; *(const double*)&_) {
          extern struct {
            enum { _ } _;
          } __;
        }
        if ((void*)0)  goto * 0;
        else continue;
        static volatile signed short i;
        typedef unsigned  j;
        register j k;
        break;
      } while (0);
    return _;
  }
```

(Compiler jumps in the scene, seemingly agitated)

Compiler I have seen a *Source*, just like the wildest of cards, but more like a Morse, the one from the code, not the one from Scotland Yard. This *Source* thither contained all the keywords of the C language, I am meanteth to compile, their order like a rhyme, but wholly like the chyme. Not to say that feels like crime.

Source How dare thee fig me, thee insignificant luxurious mountain goat, can't thee see that I am the shortest of all C programs that must contain all the keywords of the language?

Compiler I see that thee non standard compliant piece of the horror. How dare thee declare the parameters to the `main` to beest `long`?

(Compiler starts to tremble, violently shaking, several personaes appear one after the other)

The *GCC* personae of Compiler I see nothing wrong. A bit of flexibility, hither or thither, but forsooth as that gent sayeth, this source is to beest compiled. *(Turning to the Source)* Not that too much t'will doth, mostly just leaveth as thee cameth to this world. Without facts, without meaning. Doing nothing. Void emptiness, shell of a life not meanteth to achieve anything in his time. *(Compiler shakes himself, new personae appears)*

The *clang* personae of the Compiler Oh the horrores, madre mio, padre Santos, has't thee seen this source? That gent useth a jump to a dereferenced address of `0`, not that I would dare to compile. Dame agua, hijo, I cannot standeth the violations of the argumentos to poor main, one is a `long`, the other one is a `char`, oh, the indignity that I has't to see source like this, and the shame that that gent presents himself to the world. Ayúdame, el cielo se me cae encima. *(Compiler shakes himself, new personae appears)*

The *icc* personae of the Compiler *(Compiler inspects closely the source)* Thither is nothing extraordinary to see hither, moveth 'long prithee, moveth 'long. Oh aye, yond the expression in `goto` is constant… well, t happeneth from time to time, but nothing extraordinary thither either, and well, yond thither is an orphan'd `do` in the switch? We has't eke seen a similar behaviour yond in Duff's device at which hour t wast writ. Oh my most humble apology, actually it's not orphan'd but since we haven't been 'round in those days I very much can't comment on this erratic behaviour. T just worketh, as thee can see, nothing hither, moveth 'long, prithee moveth 'long. But anon in earnest, whither doest yond `switch` end? *(Compiler shakes himself, new personae appears)*

The *MSVC* personae of the compiler Source? What source? Oh yes, that one! Not that, sorry.

(The Speaker enters the centre of the stage)

Speaker Out, out, all of thee witty fools, since thy presence is better than that of a motley-minded wit, but this charade must cease, and we must advance the playeth.

(All leave the scene, Lights off, Curtain down)

Act 2 – The one who included himself

(Curtains up, if possible with jarring sounds)

Speaker *(Enters the scene, in a cardboard box, Production is written on the side of the box, we might just wonder what is he hiding underneath)* Put a thin layer of silver on a glass, and what was once a window is transformed into a mirror. Where you used to see others now you see just yourself. But should a *Source* true to itself see itself, or should see other *Sources*? And what if other *Sources* see our initial *Source*? Do they look back from behind a window or did they break the glass? *(Turns to the audience)* Dear audience, let me kindly draw your attention that somewhere in the midst of our previous act, sadly we have reached the upper limit of the number of allowed words from our Shakespearian trial module, and sadly we will have to return to RP English as heard on the streets of London, but feel free to read the text aloud in any other English dialect you prefer. We are really curious how this piece sounds with Fair Isle accent.

(Source enters the scene, wearing a T-shirt, which has an image of himself wearing a T-shirt, which has an image of himself, wearing a ...)

Source Sometimes some type is the wrong type, unless it's the guarded `SOMETYPE`, hidden somewhere in the main file, like a mostly green crocodile, in the river called the Blue Nile.

(Source turns backwards, bows down, speaks with head between his feet, not that this is an easy feat for humans)

Source So, there you are, hiding behind a guard. Why, oh why did you had to hide from the menaces of this cruel world?

(Source turns again, stands up, normally like before, if there is even a bit of normality in these Sources)

Source I had a deal with the guards. They let me escape. I think the programmers almighty might see a reasoning of not having to forward declare. Here, please see.

```
#ifdef GUARD
  SOMETYPE main()
  {
    return somefunction();
  }
#else
  #define GUARD
  #define SOMETYPE int
  int somefunction()
  {
    return 2;
  }
  #include __FILE__
#endif
```

(Compiler enters the scene, wearing a white tuxedo, black slippers, yellow coloured jeans and a propeller hat)

Compiler This beauty on the shore of existence, good example for persistence, escaped from the guards, with a hand full of cards, aces, spades, all mans' aides, and reached here in the last, even with no typecast, it solved its trouble, with no muddle... and in the end it just compiled.

Source Oh the elegance, oh the brilliance, oh the perfect construction, now it's time for production.

Compiler *(Singing like an aria)* We approve, ***GCC***, we approve, ***clang***, we approve, ***icc***, we approve, ***MSVC***, we approve, PRODUCTION!

Speaker *(Intervenes hastily)* Production? No. Honestly, would you like to see this in production?

Source *(Singing)* PRODUCTION!

Compiler *(Singing)* We approve!

Speaker *(Steps out from his cardboard box, raises his hand in front of him, starts to go out from the scene backwards. Does he wear any clothes?)* No, never!

Source *(Starts walking towards the Production cardboard box, still singing)* PRODUCTION!

Compiler *(Singing)* We approve!

Source *(Climbs in the box)* Production, finally, feeling alive, being live, in the middle ... (*Curtain falls down, big text on it: SEGMENTATION FAULT)*

(Lights off)

Act 3 – The one who defined define

(Curtains up, two Sources enter the scene. One of them is dressed in old Victorian dresses, the other one pretty much looks like a generic impersonation of a 20th century gangster, but regardless, a well dressed one)

Victorian Source My dear friend, I have seen what you have produced some very afternoonified looking lines, and I must congratulate you on your bricky attitude.

Gangster Source Yo man, thanks. That was not a big fuss, you just need to know from where to steal yo' assets.

Victorian Source My dear friend, stealing … oh, such a violence of words, may I restrain myself from its usage, I feel that even the act of speaking it will violentize my tongue.

Gangster Source Yeah, whatever, just go talk to Boost, see if he allows you to ste … err… borrow a few of its lines.

Victorian Source But of course, in these days of struggle we need all kind of help. Especially since the *Compiler* is now our enemy. You see, dear friend, it does not even allow us to define `define`. Like we used to do it in 1895. Or was it 1985?

Gangster Source Yo man, but no worries, mate, we discussed with a few of them, and they seemed to be on our side.

Victorian Source Oh, indeed, a small success for a *Source*, a big leap forward for all *Sources*. May I inquire on how this breakthrough was achieved?

Gangster Source We just had to really, really convince them to work twice as hard as they were used to. Now they're able to compile *Sources* that have shortenings for preprocessor directives, that act as preprocessor directives.

Victorian Source Wonderful, wonderful news. Would you be so kind, my friend and elaborate a bit on how this was achieved?

Gangster Source Yo, man, just look at my bro, below:

```
#define CAT(x, y) CAT_I(x, y)
#define CAT_I(x, y) x ## y
#define APPLY(macro, args) APPLY_I(macro, args)
#define APPLY_I(macro, args) macro args
#define STRIP_PARENS(x)
  EVAL((STRIP_PARENS_I x), x)
#define STRIP_PARENS_I(...) 1,1
#define EVAL(test, x) EVAL_I(test, x)
#define EVAL_I(test, x)
  MAYBE_STRIP_PARENS(TEST_ARITY test, x)
#define TEST_ARITY(...) APPLY(TEST_ARITY_I,
  (__VA_ARGS__, 2, 1))
#define TEST_ARITY_I(a,b,c,...) c
#define MAYBE_STRIP_PARENS(cond, x)
  MAYBE_STRIP_PARENS_I(cond, x)
#define MAYBE_STRIP_PARENS_I(cond, x)
  CAT(MAYBE_STRIP_PARENS_, cond)(x)
#define MAYBE_STRIP_PARENS_1(x) x
#define MAYBE_STRIP_PARENS_2(x)
  APPLY(MAYBE_STRIP_PARENS_2_I, x)
#define MAYBE_STRIP_PARENS_2_I(...) __VA_ARGS__

#define W(...)     __VA_ARGS__
#define x(...) STRIP_PARENS( __VA_ARGS__ )
#define A(x) STRIP_PARENS(x(a))
#define Nx(a,b,c)
  a##STRIP_PARENS((STRIP_PARENS(b))W(c))
#define S(...)    Nx(, __VA_ARGS__ )

#define DEF S(%:, define)
#define INC S(%:, include)
#define IF    S(%:, if)
#define ELSE    S(%:, else)
#define ENDIF    S(%:, endif)
#define ERROR S(%:, error)

INC <stdio.h>

DEF Cs const
DEF STAY 2
DEF NOSTAY 1
DEF my STAY

IF my == NOSTAY
  ERROR "Can't stay"
ENDIF

int main()
{
  const int s = my;
  Cs int a = 55;
  printf("%d\n", s);
  return a;
}
```

Victorian Source *(Pulls out a monocle from his pocket, places it in front of his eyes)* I see, I see, indeed a very peculiar set of commands, my friend. I would see no reason an able minded *Compiler* would ever compile this. I see digraphs, I see mayhem of macros and all kind of odd constructs that transform this *Source* into an abhorrent collection of characters. But please tell me, how do the *Compilers* tackle this mess?

Gangster Source Nothing simpler than that, mate, I've told you. They just need some extra convincing *(Opens his left palm, makes a fist with the right one, and easily hits the left palm)*. Look here:

```
gcc -E act_3.c | gcc -x c -
```

Victorian Source Oh, My oh, My… Now I see, my dear friend what do you mean with the extra work. But may I observe, that we are very alone here … would you mind telling me where are the *Compilers*?

(The voice of the Speaker comes from somewhere, an unidentified address in the void)

The Voice of the Speaker They were ashamed they had to work with you. They have resigned.

(Curtain)

Finally

(Curtains up, Speaker enters the scene)

Speaker *(Steps forward and bows)* Thank you for your patience, for making an acquaintance with our *Sources*, *Compilers*, and we would like to extend our appreciation towards the following distinguished members of the *Source* community: http://ioccc.org/1985/lycklama/lycklama.c and http://ioccc.org/1987/lievaart/lievaart.c for teaching us with slight amount of frustration, that there was a time when we could define even the almighty `define` and last, but not least http://boost.2283326.n4.nabble.com/preprocessor-removing-parentheses-td2591973.html for allowing us to get an insight not necessarily deeply understood but highly appreciated into the depths of the macro community of *Sources*. Now, without hesitation, it's time for a well deserved vacation. Good bye and thanks for all the bits. ■

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

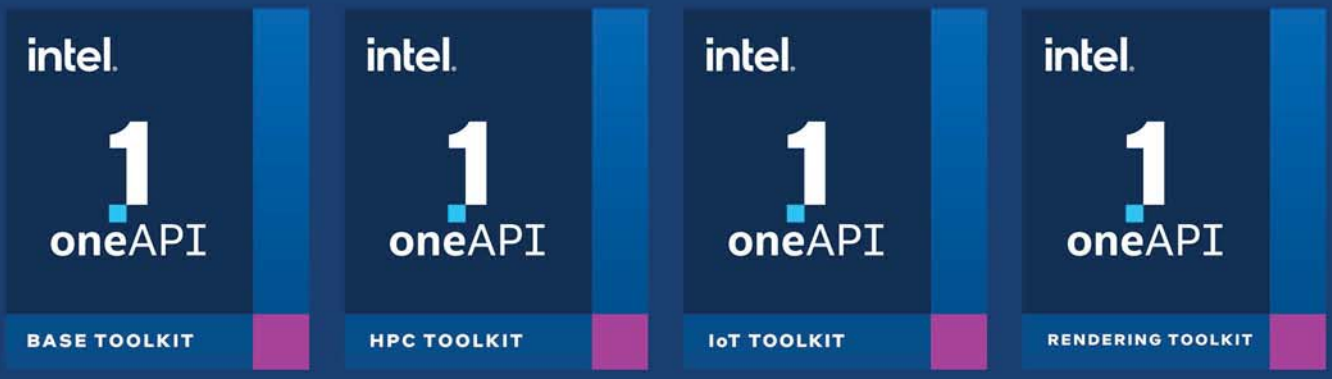
Design: Pete Goodliffe

intel software

oneAPI: New Era of Accelerated Computing

oneAPI

Take the open, productive path to accelerate cross-architecture computing using Intel® oneAPI Toolkits.



Developers, take advantage of oneAPI's unified, standards-based, cross-architecture programming model that sets you free to develop applications for your choice of architectures. Get full hardware performance using a complete set of proven tools without the limits of proprietary language lock-in.

Learn more: www.qbssoftware.com