overload 125

FEBRUARY 2015 £3

Making a Tool of Deception

C++14 provides many ways to write better code. We see how it is employed to write the modern, header-only mocking framework *Trompeloeil*.

Best Practices vs Witch Hunts

What happens when "good" practices turn "bad"

Modern C++ Testing

Comparing modern C++ testing frameworks

I Like Whitespace

Why the whitespace in your code matters

Faking C Function with fff.h

A faking function framework for C

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 125

February 2015

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 126 should be submitted by 1st March 2015 and those for Overload 127 by 1st May 2015.

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

The ACCU

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

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

FUD – Fear, uncertainty and doubt

Sometimes programming is neither an engineering discipline nor a science. Frances Buontempo considers when it slips into the numinous realm.

Previously, I considered how difficult it can be to allow creative ideas to flow in an attempt to encourage each of you to be brave and find potential articles to write up. You may be unsurprised to learn I have failed to heed my own advice and have in fact found yet another excuse for not writing an editorial. More than a lack of ideas or source of inspiration can freeze people up or make them behave in odd ways. This will require some context, so bear with me.

As I am sure you are aware, debates have raged for ages [e.g. Stack Exchange] concerning whether programming counts as a science or as an engineering discipline. It is possible to do a degree in either computer science, with an emphasis on data structures and algorithms, or in computer engineering, with an emphasis on the hardware – networks, multimedia, or circuit boards. Both may involve an element of programming, yet this is not the whole story. Some of the academic computing disciplines stray into philosophy along the road. Many early experiments (does that make it a science?) touched on the realm of Artificial Intelligence. They asked big questions like the Voight-Kampff machine, perhaps more correctly referred to as the Turing Test [Turing]. Having recently seen a review of *The Imitation Game* film about Turing where a reviewer remarked the Turing Test was like the Voight-Kampff test from the film Blade Runner, I need to be more discerning about which reviewers I listen to [thanks to this team for being better informed]. I digress. My bachelor's degree is in Mathematics and Philosophy, and is actually a Bachelor of Arts degree. My philosophy supervisor was heavily involved in research with the computing department, reputedly getting to play against the Deep Blue chess computer [Deep Blue], and I did a module in Pascal programming. Does this mean computing, or programming, can be regarded as philosophy? An article in *Scientific American* [Wartik] considers this question. First, we are reminded scientific methodology requires repeatable experiments, developing theories that match observation. "In science, a theory fails if it doesn't predict observation. In CS theory, there is no observation." [op cit] Wartik then explores similarities with mathematics, which is yet another viewpoint held by some, but concludes much computer science cannot be formalised enough to be a branch of mathematics. He remarks that to the ancient Greeks, mathematics and philosophy were almost indistinguishable, and while they have diverged to an extent now, computer people do tend to fall into different camps of 'philosophical view-points'. Even if you can't decide which is right you can think about what you are doing and why.

What impact does deciding if you are doing science, engineering, mathematics, or even philosophy have on your day to day coding? Certainly it can be sensible to step back and sketch out the complexity of an algorithm, or think for a moment about the right data structure. It can be clever to devise or engineer a system using a mixture of science, mathematics, APIs and frameworks. Though this is part of what we do, it doesn't seem to be the norm. Working with legacy code, or even looking back at code you wrote years ago can reveal the Dunning-Kruger effect in action [Dunning Kruger]. It seems that at times incompetent people lack awareness of their own incompetence and call into question the genuine ability in others. They would frequently be over-optimistic about their own performance in various tests, though could give more accurate assessment of their test performance after training, while not necessarily doing any better. Even armed with an approximately realistic idea of one's own ability, fear, uncertainty and doubt – FUD – can take hold. Though the term is a tactic used in sales and marketing, or even politics [FUD], the elements can grip a team or a lone coder. For example, if you try to introduce an open-source solution into a project, will you immediately be asked "Who will fix it if we find bugs?" If your government claims another has weapons of mass destruction, are you more likely to support a war effort? There are countless examples of the deliberate or unintended spread of misinformation bringing about an effect. Though many of the sales-focussed FUD has been deliberate, programmers find themselves being subject to it, and even propagating it. On many occasions, I have seen situations where running programs in a complicated, fragile environment leads to various instructions about what to restart in which order. On further enquiry it often seems the first person happened to do this, and it worked. Even if it makes no sense, for example to restart a service that isn't involved it is hard to resist the temptation to just do what you're told. Fear. What if it then gets even more broken? Uncertainty. This makes no sense, but perhaps they know better than me. Doubt. I doubt it's required, but I doubt my own powers of reasoning now.

This FUD can spread beyond attempting to get some software working, to how it is actually written. Without an understanding of an API or the language people do unnecessary things, including but not limited to calling `Dispose` inside a `using` block in C# or checking `this` isn't null. Such can be described as *Superstitious Code* [c2], and has various causes:

> Code written by some kind of Bad Programmer, or really exhausted adventurous programmer.
>
> - ignorance of the language, libraries, system in use
> - imaginary assumptions guiding the programming task
> - fear of hidden bugs or of doing something wrong
>
> yields Superstitious Code.

Other variants include cargo cult programming, or voodoo chicken programming. Many people get caught talking to their computer either as they code, or try to make something run, using geek shibboleths

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.

[Shibboleth] – words either only pronounceable by those in the IT crowd or used as a stream of now meaningless symbols that people say at specific points, the etymology now lost in time. In all cases an almost religious habit is driving the behaviour rather than knowledge. It is worth noting that the very word 'religion' possibly stems from the idea of regulation – going through something again and again, though could also be from *ligare*, meaning bind or connect. For the Romans, *relgio* was about knowledge of the rites relating to both private and public life, and had nothing to do with faith. Programming can involve a mixture of both. That said, regulations and habit can be good. Knowing some SOLID principles, using version control, or being able to spot a code smell early on and nip it in the bud isn't science or engineering. It may be peppered with some fear about what may happen if ignored, but not uncertainty or doubt. However, some cultish programming stances are foolish. You may be subject to strict coding standards, which were put in place with good reason, but find code reviewers are sticking to the letter, rather than the spirit of the law. Some superstitious practices are grounded in a modicum of sense. Avoid 'airport checkins' i.e. committing code to a shared coded based just before jetting off on a two week holiday. If it builds on the build machine, but not yours, consider rebuilding locally. I am sure you can think of other examples. People also consider environmental settings. I don't just mean checking how long your PATH variable has become in case something important falls off the end. It is wise to use a font size large enough that you can spot the difference between a colon and a semicolon. (This only wasted half an hour, nonetheless…) Sometimes the tests do only work with a network cable plugged in. (Shame, I hear you cry.) The environmental superstitions can go too far though. I have heard talk of the need to be wearing one's lucky underpants before going near certain code modules. As a woman, I'm never clear what to make of that, though I do claim to have a lucky hat. It's made entirely of tin foil.

Religion and superstition can only take you so far. Sensible heuristics can guide you in the right direction, or help you stay on the path, avoiding dragons and other monsters. At times, this is not enough. If confronted with a big red button, saying 'Do not press' those of a less superstitious bent will find it almost impossible not to take a scientific approach or some brand of contumacy and hit the button to find out what happens. This leads us to a realm of real magic of Hacker's Folklore [Real magic]. The tale tells us of a switch, connected by a single wire with two positions 'Magic' and 'More magic', defying the basic beliefs regarding electrical circuits. It was in the position 'More magic', and curiosity and a stubborn belief in sense drove people to switch it to just 'Magic', being certain it could not affect the computer. Which subsequently crashed. This story was not believed, but a repeat experiment led to the same outcome. More magic was clearly required.

Where does all this talk of FUD leave us? Perhaps we should embrace the unknown from time to time. If you are petrified by fear, consider walking into the dark with a friend, either by pair programming or finding a good code reviewer. Consider just making baby-steps and testing as you go. If you are unsettled by uncertainty, see what happens. If you can't be sure of how long something will take to complete, give bounds not an exact answer. If you don't know if the stock prices will go up or down, guestimate a probability of each and build a stochastic model. If you're not sure what happens if you delete some Boolean flags, try a scratch refactoring [Feathers]. Change the bits you don't like the look of and see what the compiler and tests say. Or in the absence of either, try it then roll back, just to get a better understanding of the code. Are you dithering over doubt? Just do it, and again, assuming you have version control, you can roll back. Admittedly xcopying to a prod server is a different matter, so perhaps talk to someone first. Ideally try to set up a safe environment where you can try things first. Again, if it's the config, which you then need to change completely to run on another environment, this might not work out. Trying things first and assuming the experiment is repeatable is almost like science, after all.

All told, it is easy to be persuaded into approaching a problem in a specific way. Always bounce all the services, 'Just In Case'. Never use exceptions because they always slow things down, allegedly. Always be 'rigidly agile'. Or never use open source. Or always write an in-house version. We all know to "Never say 'Never'" though. FUD may have made me dust off my rusty hat, and make further excuses for a lack of editorial, but it is worth considering what's driving you and why you are doing things in a certain way. Being un-sure is the start of many fruitful journeys to greater learning and lots of fun. There is nothing wrong with continuing to ask "Why?" no matter how old you are, and just because someone speaks with authority doesn't mean they know what they are talking about.

One of the painful things about our time is that those who feel certainty are stupid, and those with any imagination and understanding are filled with doubt and indecision.
~ Bertrand Russell, The Triumph of Stupidity

References

[c2] http://c2.com/cgi/wiki?SuperstitiousCode

[Deep Blue] http://www.theguardian.com/uk/the-northerner/2012/may/14/alan-turing-gary-kasparov-computer

[Dunning Kruger] http://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

[Feathers] *Working Effectively with Legacy Code* Michael Feather Prentice Hall 2004

[FUD] http://en.wikipedia.org/wiki/Fear,_uncertainty_and_doubt

[Real magic] http://www.catb.org/jargon/html/magic-story.html

[Shibboleth] http://en.wikipedia.org/wiki/Shibboleth

[Stack Exchange] http://programmers.stackexchange.com/questions/18886/what-discipline-does-computer-science-belong-to

[Turing] http://en.wikipedia.org/wiki/Turing_test

[Wartik] 'I'm not a real scientist, and that's okay.' Guest Blog, *Scientific American*, Nov 2010 http://blogs.scientificamerican.com/guest-blog/2010/11/12/im-not-a-real-scientist-and-thats-okay/

Best Practices vs Witch Hunts

Best practices can be a Good Thing. Sergey Ignatchenko considers when they can turn into Really Bad Things.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

In any field, software development included, there are lots of well- and less-known *best practices*. As a rule of thumb, *best practices* are very useful and in general should be followed. Unfortunately, way too often, developers are starting to take *best practices* as gospel, and/or become obsessed with them. This turns these *best practices* into outright *witch hunts*. In this article, we'll discuss what the mechanism behind witch hunts is, and why they're Really Bad Things.

Life cycle: from best practice to witch hunt

Let's see how best practice usually evolves (YMMV, but usually the picture is rather similar):

1. A few teams start (usually independently) using a certain practice within their projects.
2. The practice becomes more widespread.
3. Somebody publishes the practice as a 'rule of thumb', usually with rationale behind, and with certain cases of applicability exceptions.
4. The practice is officially named as 'best practice' by some authoritative source. The rationale is usually still present, but some of the applicability exceptions are often lost due to the lack of space.
5. The practice becomes pretty much universal. At the same time, both the rationale and the applicability exceptions are gradually forgotten.
6. Then some people ('zealots') emerge, who think that the ~~Earth will stop turning~~ project is hopelessly deficient if there is even one single case of the best practice being violated. Neither the rationale nor the applicability exceptions are taken into account.
7. When such a 'zealot' lays his hands on a real-world project, he starts to enforce the practice mercilessly.
8. At first, the 'zealot' normally starts with eliminating the least controversial of the best practice violations. Among other things, this means that at this stage he usually eliminates violations which are in line with the rationale and in conditions where the applicability exceptions do not apply. As a result, his efforts at this stage tend to benefit the project in general, which is usually more or less obvious to the whole team. This triggers the natural feeling that the 'zealot' is right in his fight for the best practice ideals.
9. As a natural consequence, only a few people dare to challenge the 'zealot' with his further fight for the best practice. Also, a substantial chunk of the team starts to believe that this specific best practice is the Absolute Good Thing. And this is the point where the best practice effectively becomes a witch hunt.
10. Applying this specific best practice is never questioned, and both rationale and applicability are completely ignored. In the name of this best practice, pretty much anything can be done within this team. In particular, it includes violating any other best practices (especially those lacking their own 'zealots' in the team). All kinds of weird things can and will happen in such teams (for practical examples, see below). As an additional benefit, if there are two best practices with 'zealots' behind them in the same team, any kind of conflict between these best practices can result in a 'holy war'.

Wait, but it is still a best practice, isn't it?

The best practice is useful only if all considerations (including both rationale and applicability) are carefully considered. Without taking these considerations into account, applying the best practice often leads to conflicts with other best practices, which in turn can easily lead to grave consequences.

What can possibly go wrong when applying a best practice?

Naming something *best practice* doesn't really change its impact on the projects. As with pretty much anything out there and despite the name, *best practices* are not absolute virtues. Rather, they are 'rules of thumb', with all kinds of considerations which need to be taken into account. The very name 'best practice' is misleading – a more appropriate name would be 'usually a best practice'. Below are a few examples of 'best practices' which in application went wrong at some point.

Example 1. Mild example – magic numbers

Use of 'magic numbers' (using unnamed numerical constants) is generally frowned upon in the programming community. An associated *best practice* is to replace them with named constants. This *best practice* exists for two good reasons.

Rationale:

1. **Better readability**
2. **Better maintainability in case the constant changes**

So far, so good, but recently I've run into discussion where the author has asked if in the code

```
kbytes = bytes / 1024;
```

`1024` is a 'magic number' and should be replaced with the named `BYTES_PER_KBYTE`. Moreover, there were quite a few people saying that `1024` is indeed a 'magic number' which should be replaced. However, I contend (and most of the practical programmers I've spoken about it agree) that this is wrong, and that `1024` should be left as is (unless an exception applies as described below).

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

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

best practice is useful only if all considerations (including both rationale and applicability) are carefully considered

To realize the reason why this *best practice* is wrong, one should take a look at the very reasons behind it. Note that in our case, both reasons behind having this best practice do not apply. First, **`bytes / 1024`** is more readable than **`bytes / BYTES_PER_KBYTE`**, and second, the chances of **`BYTES_PER_KBYTE`** ever changing are infinitesimally small (ok, unless you're into nitpicking between kilobytes and kibibytes, and are planning to change the semantic meaning of the term **`KBYTE`** in the context of your project, which would be a Very Bad Thing *per se*).

This means that **`BYTES_PER_KBYTE`** is not a good thing to have, and the plain **`1024`** is generally preferable. However, as we are speaking about applicability, this observation is also not an absolute, and there are possible exceptions. One such exception arises if you need a consistent way of displaying your bytes to the end-user. In this case, a constant such as **`BYTES_TO_DISPLAY_KBYTES`** might make sense (and it may change later, which justifies its existence), but the generic **`BYTES_PER_KBYTE`** is still pretty much useless.

Ok, this was a very mild example of a *witch hunt*, with very mild negative implications.

Example 2. Stronger example – memory leaks

Those dealing with C/C++ are familiar with memory leaks[1]. Common *best practice* is to avoid them at all costs.

Rationale:

> **To avoid using unnecessary memory which can exhaust computer/process resources**

Applicability Exception:

> **All memory allocated via `malloc()` will be freed on program exit anyway, so calling `free()` right before program exit is not necessary.**

One of the common methods used for detecting memory leaks is using some kind of tool (such as Valgrind for Linux and the built-in debugger for Windows) which runs on program exit and lists all the blocks which were allocated via **`malloc()`** but were never deallocated via **`free()`**. It is indeed a very useful tool, and it is very good *best practice* to run this tool and eliminate as many of these memory leaks as possible. However, should we always aim to eliminate *all of them*?

More than once in my programming career, I have needed to allocate a once-per-program buffer (for example, for logging purposes) which was used by all the program threads. When I faced this task for the very first time 20 or so years ago, I tried to deallocate this buffer properly on program exit. However, in some weird scenarios[2] deallocating it caused some race conditions, which once-in-100-or-so exits have caused a program crash (thread writing to an already-deallocated buffer) for no real reason.

That time, I spent almost two weeks trying to fix this elusive problem (which kept reappearing after each fix in a new form). It continued as a kind of weird ding-dong battle until I asked myself: what would change if *I don't deallocate this buffer at all*? The whole rationale of fighting memory leaks doesn't really apply when we've already decided to exit the program, as in a few microseconds it will all be freed anyway (and in a much more efficient manner BTW). As soon as I realized this, the problem went away for good, and I was able to throw away a few dozen of rather weird synchronization lines of code which I wrote when trying to fix the problem by other means.

In this example, I myself was guilty of *witch hunting*, though I'm humbly asking the sentencing judge to consider my inexperience at that point as a mitigating circumstance.

Our next example is more severe, and would involve breaking the code as a result of a *witch hunt*.

Example 3. Breaking code while fighting -Wall

This one I've seen myself more than once. In some project, there is a perfectly valid C code, which involves some implicit casts (like unsigned-to-signed of the same size, or well-defined integer truncations/promotions). All these casts are perfectly well-defined in C, and lead to perfectly well-defined results both in theory and practice. The code is reasonably readable too. In short, there are no (zero, nada) problems with the code whatsoever.

Then, a 'zealot' comes in and adds **`-Wall`** to the compiler command line. Right away, the compiler starts to complain about these casts (why the compiler does it is beyond me, but this is not in the scope now). These warnings are overzealous, as the code is well-defined and standard-compliant.

Then, our 'zealot' starts to get rid of these overzealous warnings – not by disabling those overzealous warnings, and not even by thoughtfully adjusting types so there are less conflicts, but by merely inserting explicit casts as he sees fit. He needs to insert dozens of such casts, and he doesn't pay much attention to what the-code-he-changes really does (he's on the job of eliminating warnings, and he has much more to do, anything else is merely an obstacle on his way to achieving the Greater Good of **`-Wall`** without warnings).

As he makes a mistake when inserting casts, the previously perfectly valid code becomes broken. Even worse, the code might be broken in fringe cases which may go unnoticed for a while. Not to mention that code with tons of explicit casts becomes much less readable. I rest my case and ask the jury to sentence our 'zealot' to a life of abstinence from programming.

Now to our next, and final, example of *witch hunting*.

Example 4. The ultimate example – Debian RNG disaster

Once upon a time, there was a library named OpenSSL. As a part of it, they used a random number generator, which in turn, had two lines of code

1. While those writing in garbage-collected languages may not be familiar with memory leaks and even think that there can be no memory leaks in JVM, it is wrong at least for so-called semantic memory leaks [NoBugs12]
2. AFAIR, most of the trouble was about threads being not terminated for various reasons, and it does happen when dealing with network stuff and you don't want your user to wait forever for no apparent reason.

which were using uninitialized memory. It wasn't a real problem, as the whole idea was about gathering as much entropy as they could get their hands on, and uninitialized memory couldn't possibly hurt them.

Then, an overzealous Debian supporter, in a holy fight to eliminate all Valgrind warnings, commented out these two lines of code [Mason08] [Schneier08] [Kroll13]. Nothing changed, except that all the keys generated by all the Debian-based distributions (yes, these do include the all-popular Ubuntu), became easily guessable until this bug was fixed (which took over two years). Worse than that, when the problem was discovered, it meant that all the SSL exchanges between such Debian-based distributions, suddenly became retrospectively vulnerable (i.e. if somebody recorded them, he'd be able to decrypt them now based on the nature of this vulnerability).

The whole incident was at the time the worst security incident in the entire open source community, and it resulted precisely from an overzealous developer trusting tools and making changes to eliminate warnings without understanding the context or the potential implications.[3]

It is interesting to note that when analyzing the disaster, all kinds of explanations (read: excuses) were given for it happening, including the outright ridiculous, "The OpenSSL code was too clever by half" [Cox08]. Very few sources (if any) have mentioned the real culprit: an obsession with the enforcement of *best practices*, which unfortunately too easily becomes a witch hunt. BTW, I'm not trying to condemn the person who made the change – but rather the whole culture where such *witch hunts* (without taking into consideration related circumstances) are considered the Right Thing To Do.

I'm scared, what should I do?

There are at least two lessons to be learned from this article. The first one is:

> Whenever you're in doubt about applying a *best practice* – think about the rationale behind it. If the rationale doesn't apply in your case – forget about the *best practice*, it doesn't apply here.

3. The argument that he tried to ask OpenSSL team and asked in the wrong place, and who's responsible for it happening (which has resulted in lots of fingerpointing between Debian and OpenSSL at the time), is beyond the scope of this article.

The second lesson applies to much more dangerous *witch hunts* within existing code:

> Whenever there is the slightest risk that by enforcing a *best practice* on existing code, you'll break it – think more than twice before going ahead with your change. And no, the change being just commenting two lines and having no apparent downsides, is not guaranteed to be safe.

Merciless refactoring is good, merciless refactoring having no clue about what you're doing, is not. ■

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[Cox08] Russ Cox, 'Lessons from the Debian/OpenSSL Fiasco', http://research.swtch.com/openssl

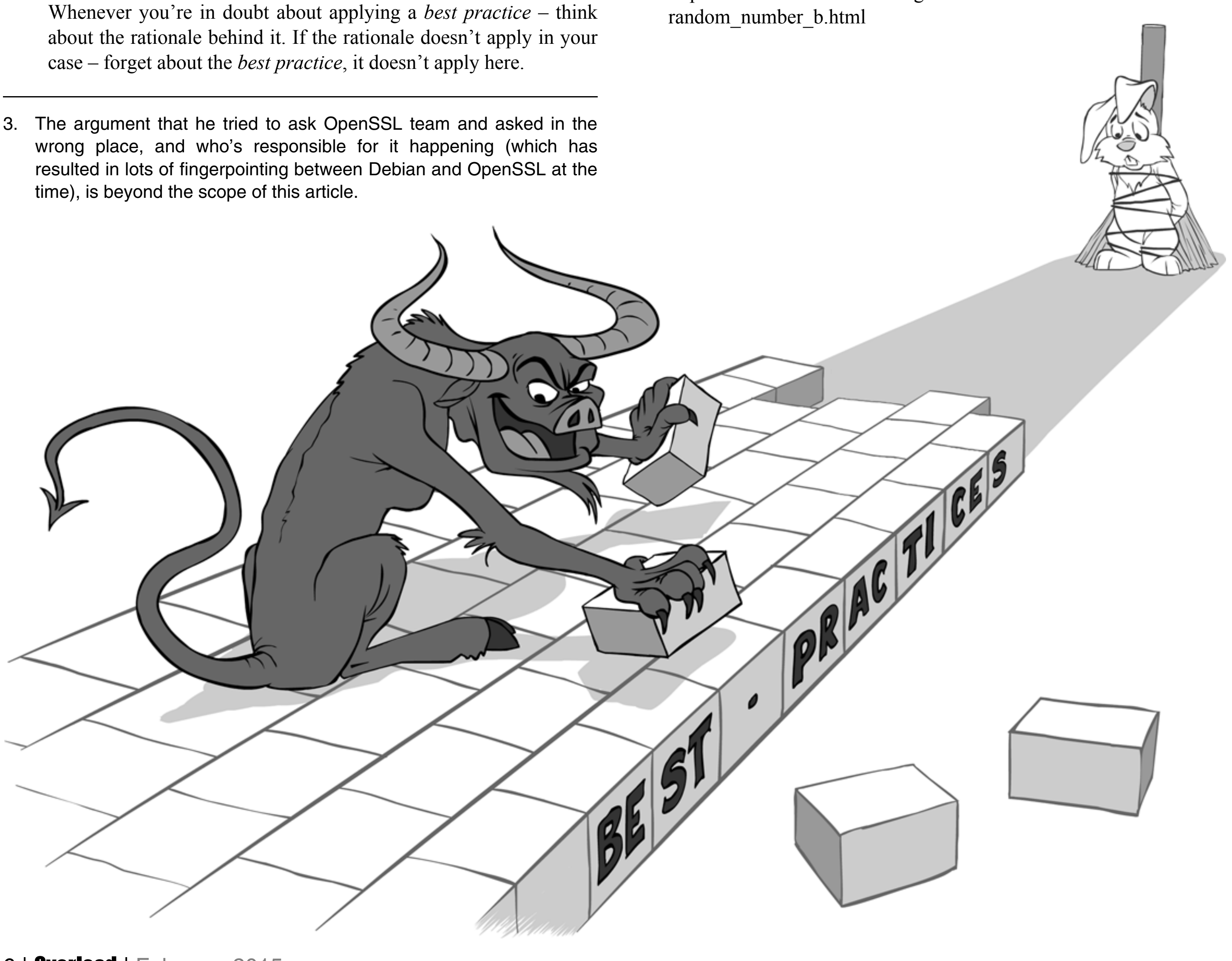
[Kroll13] Joshua Kroll, 'The Debian OpenSSL Bug: Backdoor or Security Accident?', https://freedom-to-tinker.com/blog/kroll/software-transparency-debian-openssl-bug/

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

[Mason08] Justin Mason, 'Serious Debian/Ubuntu openssl/openssh bug found', http://taint.org/2008/05/13/153959a.html

[NoBugs12] 'No Bugs' Bunny, 'Memory Leaks and Memory Leaks', http://accu.org/index.php/journals/1936

[Schneier08] Bruce Schneier, 'Random Number Bug in Debian Linux', https://www.schneier.com/blog/archives/2008/05/random_number_b.html



Making a Tool of Deception

Is it possible to use modern C++ to make mocking easy? Björn Fahller introduces Trompeloeil, a header-only mocking framework for C++14.

"I wonder if I can...?" are dangerous words. They often lead to disappointment, occasionally to commitment, and almost always to spending time.

In this particular case I wondered if I could use lambdas and other modern C++ features to make powerful mocking constructs easy to use. It turned out that this time I hit all of the three consequences of my question, and the result is the Trompeloeil mocking framework [trompeloeil].

My unit-testing experience is heavily coloured by google-mock [gmock]. While I have tinkered with other mocking frameworks, [gmock] is the one I have a working experience with. This has undoubtedly influenced my view on how mocks are used.

The issues I wanted to address may sound like [gmock] bashing, but that would be unfair – the extra expressive power that C++11/14 gives over C++98 makes a huge difference. My list of desired features are:

1. Match parameter values as boolean expressions inline in expectations.
2. Write side effects for matched expectations inline as any statement.
3. Express return values for matched expectations inline as expressions.
4. Allow wild cards for "don't care" values in expectations, even for overloaded functions.
5. Easily understood lifetime of objects used in expectations.
6. Control of lifetime of a mock object that may be destroyed by the test subject.
7. Implementation a in single header file.
8. Compilation errors from simple mistakes (like forgetting return in a non-void function.)
9. Rely less on the preprocessor than [gmock] does.
10. Shorter compilation times compared to [gmock].

This article tries to explain how [trompeloeil] is made, although all solutions listed here are simplified to save space and keep focus on the important bits.

Mock implementations

The syntax chosen for defining and placing expectations on mocks is similar to that of [gmock]. Listing 1 shows the definition of a class with two mocked functions, and Listing 2 shows how expectations are placed on an instance.

The first problem to solve is that the mock implementation of a member function must search for matching expectations, and this must also work when the signature types don't match perfectly. In Listing 2 `"cat"` is not a `std::string` for the first expectation, but it is equal-comparable to one, and the wild card in the second expectation must only match the `int` overload.

```
class Mock
{
public:
 MAKE_MOCK1(foo, void(std::string));
 MAKE_MOCK1(foo, bool(int));
};
```

Listing 1

```
Mock obj;
REQUIRE_CALL(obj, foo("cat"));
REQUIRE_CALL(obj, foo(ANY(int))
  .RETURN(_1 > 0);
```

Listing 2

The chosen implementation is that `MAKE_MOCKn()` adds a list of expectations as a member variable, and `REQUIRE_CALL()` creates an expectation object that is added to the list, which leaves the problem of knowing the type of the expectation object. A simplified, slightly pseudocoded, version of this logic implemented by the `MAKE_MOCKn()` macro, is shown in Listing 3, where `PARAMS(num, sig)` creates a parameter list of `num` parameters from the signature `sig`, and `LINEID(name)` appends the current line number to the name.

The idea is that `REQUIRE_CALL(obj, func(params))`, can use `decltype(obj.tag_func(params))` to get the `tag_type`, and from there call the static member function to create the matcher object. This works even if the type isn't a perfect match, like using a c-string literal for a `std::string` parameter, and for the wild card, which can convert to the desired type when finding the tag, and compares equal to any value of the desired type. This takes care of item 4 in the list of desired features.

The logic for finding the list of expectations that the matcher object adds itself to is similar.

Matches and actions

The expectation object created by `REQUIRE_CALL()` is an instance of the template `call_matcher<Sig, Value>`, and is basically a simple struct containing additional conditions, side effects and a return handler. Listing 4 shows a simplified version.

In `call_matcher<Sig, Value>`, `Sig` is the signature of the mocked member function, and `Value` is a tuple containing copies of all values given in the parameter list to the function in `REQUIRE_CALL()`.

In addition, `condition<Sig>` is `std::function<bool(const param_tuple&)>`, and `side_effect<Sig>` is `std::function<void (param_tuple&)>`, where `param_tuple` is a `std::tuple<>` with references to all parameters given in the call.

Björn Fahller is a systems analyst, technical team leader and software developer, with experience in embedded systems development since 1994. Smiles are brought through playful programming, aviating and socializing a no-longer-quite-feral cat. He can be reached at bjorn@fahller.se

The solution outlined works, but it is not very friendly to an error prone developer using it

```
template <typename sig, typename ... U>
auto make_call_matcher(U&& ... u)
{
  using std::forward;
  using std::make_tuple;
  using param_t =
    decltype(make_tuple(forward<U>(u)...));
  using matcher = call_matcher<sig, param_t>;
  return new matcher(forward<U>(u)...);
}
#define MAKE_MOCK_NUM(num, name, sig)           \
  struct LINEID(tag_type)                       \
  {                                             \
    template <typename ... U>                   \
    static auto name(U&& ... u)                 \
    {                                           \
      return make_matcher<sig>                  \
      (std::forward<U>(u)...);                  \
    }                                           \
  };                                            \
  LINEID(tag_type) tag_                         \
    ## name(PARAMS(num, sig));
#define MAKE_MOCK1(name, sig)                   \
MAKE_MOCK_NUM(1, name, sig)
```

Listing 3

The mock implementation of a member function creates a **param_tuple** instance, and searches the list of **call_matchers**, checking first if **value** matches, and if it does, if all **condition**s match. If no match is found, a violation is reported. If a match is found, all **action**s are called and finally the result of calling the **return_handler** is returned.

```
template <typename Sig, typename Value>
struct call_matcher
{
  template <typename ... U>
  call_matcher(U&& u) :
    value(std::forward<U>(u)...) {}
  template <typename R>
  call_matcher& set_return(R&& r) {
    return_handler = std::forward<R>(r);
    return *this;
  }
  std::list<condition<Sig>>      conditions;
  std::list<side_effect<Sig>>    actions;
  std::function<return_handler_sig<Sig>>
                                 return_handler;
  Value                          value;
}
```

Listing 4

If you look at the **RETURN()** in Listing 2, you see that the parameter is referred to as **_1**. **RETURN** is a macro, with a shortened implementation in Listing 5.

The lambda parameter **x** becomes a reference to the **param_tuple** instance mentioned above, and **mkarg<n>(x)** returns the reference held by the tuple if **n** is a legal index, or an instance of **illegal_parameter<n>** otherwise. The latter ensures compilation errors if you accidentally refer to something that doesn't exist. **ignore()** is a simple empty function that prevents compiler warnings for unused local variables. The use of **auto** in the parameter list for the lambda is the only construction in [trompeloeil] that requires C++14, in other places C++14 offers a convenience over C++11, but is not strictly needed. Extra conditions are handled similarly using a **WITH()** macro, and actions using a **SIDE_EFFECT()** macro. This construction takes care of items 1, 2, and 3 in the feature list. It also solves item 5, easily understood lifetimes of objects used. Any value given directly in the parameter list to **REQUIRE_CALL()** is copied and lives as long as the expectation object does. Any value in **RETURN()**, **SIDE_EFFECT()** and **WITH()** are copied/moved, and the lifetime ends when the expectation object is destroyed. There are also versions of the latter 3 macros, **LR_RETURN()**, **LR_SIDE_EFFECT()** and **LR_WITH()**, which use a reference capture for the lambda (LR for Local Reference, not an ideal name, but it works.)

If it will fail, fail immediately

The solution outlined above works, but it is not very friendly to an error prone developer using it. Forgetting a **RETURN()** in a non-void member function gives a run time error. A **RETURN()** with wrong type gives the all too familiar C++ template error vomit, and somehow squeezing in **RETURN()** several times uses only the last one added.

In order to provide better error pinpointing, **REQUIRE_CALL()** does not only instantiate a **call_matcher** template, it also instantiates a **call_modifier** template that operates on the **call_matcher**. A simplified **call_modifier** template is shown in Listing 6. The **call_modifier** template is instantiated with the type of the **call_matcher**, and **matcher_info** of the function signature. The helper **return_of_t<>**, is a simple template alias of the return type from a function signature.

This technique of using a template inhering stepwise modifications of known types as a trampoline for the actual work works very well for providing good error messages. **static_assert** is often messy because

```
#define RETURN(...)              \
 set_return([=](auto& x) {       \
 auto& _1 = mkarg<1>(x);         \
 auto& _2 = mkarg<2>(x);         \
 ignore(_1, _2);                 \
 return __VA_ARGS__;             \
 })
```

Listing 5

An amazingly cool feature would be if **parameters could be referenced** in expectations **by their names**, instead of positional identities.

```
template <typename Sig>
struct matcher_info
{
  using signature = Sig;
  using return_type = void;
};

template <typename RetType, typename Parent>
struct return_injector : Parent
{
  using return_type = RetType;
};

template <typename Matcher, typename Parent>
struct call_modifier : public Parent
{
  using typename Parent::signature;
  using typename Parent::return_type;

  template <typename H>
  auto set_return(H&& h)
  -> call_modifier<Matcher,
       return_injector<
       return_of_t<signature>,
       Parent
     >
  {
    using namespace std;
    using h_rt =
      decltype(h(declval<param_tuple>()));
    using rt = return_of_t<signature>;
    static_assert(
      is_constructible<rt, h_rt>::value
      || !is_same<rt, void>::value,
      "RETURN for void function");
    static_assert(
      is_constructible<rt, h_rt>::value
      || is_same<rt, void>::value,
      "RETURN wrong type for function");
    static_assert(
      is_same<return_type, void>::value,
      "Multiple RETURN");
    matcher.set_return(forward<H>(h));
    return {matcher}
  }
```

Listing 6

compilation doesn't stop at failure, and the intended message is lost in loads of other messages. This technique, however, limits the mess substantially and typically provides good feedback that is not hidden in a long list of irrelevant problems. The conditions for each **`static_assert`** are stricter than necessary to avoid tripping several of them.

This takes care of item 8 in the list of desired features, good compilation errors for simple mistakes, but it does so at a compile time cost.

Now and then

I'm rather pleased with where this has come so far. Mocking with [trompeloeil] is easy, with very readable test code due to inline expressions in the expectations, and it is easy to understand the lifetimes of objects. Most error messages are good, but more work can be done there. In this article I have not shown how lifetime expectations are controlled, nor how you can decide which expectations must be met in sequence, and which are unrelated to each other, but those too are easily expressed.

Compilation time and binary size are disappointing. Better than [gmock], but only by a narrow margin, and the the frivolous reliance on the preprocessor feels like a failure.

Going forward, I really want to address the compilation times. Faster than [gmock] means it's within what people accept, but I think it's too slow for good edit-build-run TDD cycles.

I would also like to have a better **`MAKE_MOCK()`** macro, which doesn't need the number of arguments explicitly, since that is a recurring source of unhelpful errors.

An amazingly cool feature would be if parameters could be referenced in expectations by their names, instead of positional identities.

If you have ideas for advancing [trompeloeil] further, please get in touch. ■

References

[gmock] https://code.google.com/p/googlemock/

[trompeloeil] https://github.com/rollbear/trompeloeil

Modern C++ Testing

Various C++ testing framework exist. Phil Nash compares CATCH with the competition.

As many readers may know, Catch is a test framework that I originally wrote (and still largely maintain) for C++ [Catch]. It's been growing steadily in popularity and has found its way to the dubious spotlight of HackerNews on more than one occasion.

The most recent of such events was late last August. I was holidaying with my family at the time and didn't get to follow the thread as it was unfolding so had to catch up with the comments when I got back. One of them [HackerNews] stuck out because it called me out on describing Catch as a 'Modern C++' framework (the commenter recommended another framework, Bandit, as being 'more modern').

I first released Catch back in 2010. At that time C++11 was still referred to as C++1x (or even C++0x!) and the final release date was uncertain. So Catch was written to target C++03. When I described it as being 'modern' it was in that context and I was emphasising that it was a break from the past. Most other C++ frameworks were just reimplementations of JUnit in C++ and did not really embrace the language as it was then. The use of expression templates for decomposing the expressions under test was also a factor.

Of course since then C++11 has not only been standardised but is fully, or nearly fully, implemented by many leading, mainstream, compilers. I think adoption is still not high enough, at this point, that I'd be willing to drop support for C++03 in Catch (there is even an actively maintained fork for VC6! [Moene12]). But it is enough that the baseline for what constitutes 'modern C++' has definitely moved on. And now C++14 is here too [Sutter14] – pushing it even further forward.

'Modern' is not what it used to be

What does it mean to be a 'Modern C++ Test Framework' these days anyway? Well the most obvious thing for the user is probably the use of lambdas. Along with a few other features, lambdas allow for a lot of what previously required macros to be done in pure C++. I'm usually the first to hold this up as A Good Thing. In a moment I'll get to why I don't think it's necessarily as good a step as you might think.

But before I get to that; one other thing: For me, as a framework author, the biggest difference C++11/14 would make to something like Catch would be in the internals. Large chunks of code could be removed, reduced or at least cleaned up. The 'no dependencies' policy means that Catch has complete implementations of things like shared pointers, optional types and function objects – as well as many things that must be done the long way round (such as iterating collections – I long for range for loops – or at least `BOOST_FOREACH`).

Phil Nash is easily fascinated. Along the way, outside of contract work, consulting, training and coaching he has authored open source projects such as Catch (a C++ & Objective-C test framework), Clara (a C++ command line parser) and several iOS apps. He can be contacted at accu@philnash.me

The competition

I've come across three frameworks that I'd say qualify as truly trying to be 'modern C++ test frameworks'. I'm sure there are others – and I've not really even used these ones extensively – but these are the ones I'll reference in this discussion. The three frameworks are:

- Lest – by Martin Moene, an active contributor to Catch – and partly based on some Catch ideas – re-imagined for a C++11 world [Moene].
- Bandit – this is the one mentioned in the Hacker News comment I kicked off with [Karlsson].
- Mettle – Seeing this mentioned in a tweet from @MeetingCpp is what kicked off the train of thought that led me to this article [Porter].

The case for test case macros

But why did I say that the use of lambdas is not such a good idea? Actually I didn't quite say that. I think lambdas are a very good idea – and in many ways they would certainly clean up at least the mechanics of defining and registering test cases and sections.

Before lambdas C++ had only one place you could write a block of imperative code: in a function (or method). That means that, in Catch, test cases are really just functions – which must have a function signature – including a name (which we hide – because in Catch the test name is a string). Those functions must be captured somehow. This is done by passing a pointer to the function to the constructor of a small class – whose sole purposes is to forward the function pointer onto a global registry. Later, when the tests are being run, the registry is iterated and the function pointers invoked.

So a test case like this:

```
TEST_CASE( "test name", "[tags]" )
{
    /* ... */
}
```

...written out in full (after macro expansion) looks something Listing 1.

(`generatedFunctionName` is generated by yet another macro, which combines root with the current line number. Because the function is declared static the identifier is only visible in the current translation unit (cpp file), so this should be unique enough)

So there's a lot of boilerplate here – you wouldn't want to write this all by hand every time you start a new test case!

With lambdas, though, blocks of code are now first class entities, and you can introduce them anonymously. So you could write them like:

```
Catch11::TestCase( "test name", "[tags]", []()
{
  /* ... */
} );
```

```
static void generatedFunctionName();
namespace{
  ::Catch::AutoReg generatedNameAutoRegistrar
    (   &generatedFunctionName,
        ::Catch::SourceLineInfo( __FILE__,
           static_cast<std::size_t>( __LINE__ )
),
        ::Catch::NameAndDesc( "test name",
          "[tags]") );
  }
  static void generatedFunctionName()
  {
    /* .... */
  }
```

Listing 1

This is clearly far better than the expanded macro. But it's still noisier than the version that uses the macro. Most of the C++11/14 test frameworks I've looked at tend to group tests together at a higher level. The individual tests are more like Catch's sections – but the pattern is still the same – you get noise from the lambda syntax in the form of the `[]()` or `[&]()` to introduce the lambda and an extra `);` at the end.

Is that really worth worrying about?

Personally I find it's enough extra noise that I think I'd prefer to continue to use a macro – even if it used lambdas under the hood. But it's also small enough that I can certainly see the case for going macro free here. Since the first version of this article was published on my blog the author of Lest commented that he now uses a macro for test case registration too. He also reported that, at least with present compilers, the lambda-based version has a significant compile-time overhead

Assert yourself

But that's just test cases (and sections). Assertions have traditionally been written using macros too. In this case the main reasons are twofold:

1. It allows the expression evaluation to be wrapped in an exception handler.
2. It allows us the capture the file and line number to report on.

(1) can arguably be handled in whatever is holding the current lambda (e.g. it or describe in Bandit, suite, subsuite or expect in Mettle). If these blocks are small enough we should get sufficient locality of exception handling – but it's not as tight as the per-expression handling with the macro approach.

(2) simply cannot be done without involving the preprocessor in some way (whether it's to pass `__FILE__` and `__LINE__` manually, or to encapsulate that with a macro). How much does that matter? Again it's a matter of taste but you get several benefits from having that information. Whether you use it to manually locate the failing assertion or if you're running the reporter in an IDE window that automatically allows you to double-click the failure message to take you to the line – it's really useful to be able to go straight to it. Do you want to give that up in order to go macro free? Perhaps. Perhaps not.

Interestingly Lest still uses a macro for assertions and (again, as the author commented on my blog) Mettle now uses a macro for `expect()` in order to capture that information.

Weighing up

So we've seen that a truly modern C++ test framework, using lambdas in particular, can allow you to write tests without the use of macros – but at a cost!

So the other side of the equation must be: what benefit do you get from eschewing the macros?

Personally I've always striven to minimise or eliminate the use of macros in C++. In the early days that was mostly about using `const`, `inline` and templates. Now lambdas allow us to address some of the remaining cases and I'm all for that.

But I also tend to associate a much higher 'cost' to macro usage when it generates imperative code. This is code that you're likely to find yourself needing to step through in a debugger at runtime – and macros really obfuscate this process. When I use macros it tends to be in declarative code. Code that generates purely declarative statements, or effectively declarative statements (such as the test case function registration code). It tends to always generate the exact same machinery – so should not be sensitive to its inputs in ways that will require debugging.

How do Catch's macros play out in that regard? Well the test case registration macros get a pass. Sections are a grey area – they are on the path of code that needs to be stepped over – and, worse, hide a conditional (a section is really just an `if` statement on a global variable!). So score a few points down there. Assertions are also very much runtime executable – and are frequently on the debugging path! In fact stepping into expressions being asserted on in Catch tests can be quite a pain as you end up stepping into some of the 'hidden' calls before you get to the expression you supplied (in Visual Studio, at least, this can be mitigated by excluding the Catch namespace using the StepOver registry key [Pennell04]).

Now, interestingly, the use of macros for the assertions was never really about C++03 vs C++11. It was about capturing extra information (file/line) and wrapping in a try-catch. So if you're willing to make that trade-off there's no reason you can't have non-macro assertions even in C++03! That said, the future may hold a non-macro solution even for that, in the form of proposal N4129 for a `source_context` type to be provided by the language [N4129].

Back to the future

One of my longer arcs of development on Catch (that I edge towards on each refactoring) is to decouple the assertion mechanism from the guts of the test runner. You should be able to provide your own assertions that work with Catch. Many other test frameworks work this way and it allows them to be much more flexible. In particular it will allow me to decouple the matcher framework (and maybe allow third-party matchers to work with Catch).

Of course this would also allow macro-less assertions to be used (as it happens the assertions in bandit and mettle are both matcher-like already).

So, while I think Catch is committed to supporting C++03 for some time yet, that doesn't mean there is no scope for modernising it and keeping it relevant. And, modern or not, I still believe it is the simplest C++ test framework to get up and running with, and the least noisy to work with. ■

References

[Catch] http://catch-lib.net

[HackerNews] https://news.ycombinator.com/item?id=8221135

[Karlsson] https://github.com/joakimkarlsson/bandit

[Moene] https://github.com/martinmoene/lest

[Moene12] http://martin-moene.blogspot.co.uk/2012/12/catch-c-test-framework-vc6-port.html

[N4129] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4129

[Pennell04] http://blogs.msdn.com/b/andypennell/archive/2004/02/06/69004.aspx

[Porter] https://github.com/jimporter/mettle

[Sutter14] http://isocpp.org/blog/2014/08/we-have-cpp14

I Like Whitespace

Programming style can cause endless arguments. Bob Schmidt shares why he thinks whitespace matters.

This is an opinion piece. The opinions expressed are my own, and nobody else's, and most likely don't reflect the opinions of Fran or the reviewers. Because they are my opinions, I think they are right. If I thought they were wrong I'd have other opinions. If I thought they were wrong and still subscribed to them it would be a serious case of cognitive disconnect, and we can't be having any of that.

This article contains sarcasm and (possibly failed) attempts at humor. My sarcasm font is broken, so you're just going to have to recognize those sections without any help from me.

I know that nothing attracts more heat than light than discussions about programming style. Where do the braces go; how many spaces to indent; in C++ where should the `const` keyword be placed. We all have opinions on all of these subjects, and more. Most aspects of programming style are matters of taste, and taste is subjective. My style is, of course, the only correct one.

I use a lot of whitespace when writing code. From reviewing and maintaining a lot of code written by others I know that this is not a common practice. I think it is an issue of weak thumbs and pinky fingers. Later I'll suggest exercises for strengthening these digits so that inserting additional whitespace (in the form of spaces, carriage returns and (temporary) tabs) will become easier.

Motivation

Back in the mid-1980s I had the misfortune of working on a system that used RATFOR, a language created by Brian Kernighan that is a pre-compiler for FORTRAN 66 [Kernighan76]. RATFOR provides modern programming structure syntax to FORTRAN 66, ostensibly making it easier to write good code than straight FORTRAN.

A code beautifier was part of the RATFOR software tool set. Running a source file through the beautifier generated a hard-copy listing of the source, with indentations based on the source syntax. It was an OK tool for the time, although it didn't always represent the true structure of the code correctly.

Unfortunately, the presence of the beautifier gave some of the original programmers on the project 'permission' to not structure their source code. Open a source file on a programmer's terminal, and every line started in the far left-hand column. (See Listing 1.)

This was perhaps the ugliest code I have ever had the misfortune to work on. I would completely format the entire source file prior to trying to fix or enhance the code. (I have not yet recovered from the trauma, in spite of all the software conference 'therapy' I've attended in the intervening years.)

Bob Schmidt is president of Sandia Control Systems, Inc. in Albuquerque, New Mexico. In the software business for 33 years, he specializes in software for the process control and access control industries, and dabbles in the hardware side of the business whenever he has the chance. He can be contacted at bob@sandiacontrolsystems.com.

```
 IF(A<B) THEN
[
IF(B<C) THEN
[
CALL X(A)
]
ELSE
[
CALL Y(B)
]
ENDIF
]
ELSE
[
CALL Z(C)
]
ENDIF
```

Listing 1

It was working on this ugly RATFOR code that first got me really thinking about coding style. When I first started programming, back in the dark ages of 1981, any resemblance my code had to a style was strictly an accident. I like to think my coding style has evolved a lot since then. Today I prefer a style that showcases the code and its structure, so I don't have to go looking for it.

Spaces

Iwouldn'treadalineoftextinanarticleifitwaswrittenlikethis. So why should I have to read a line of code formatted like this:

```
for(int i=0;i<someMaxValue;++i)
```

I believe it is so much easier to read if it is written more like this:

```
for ( int i = 0;   i < some_max_value;   ++ i )
```

Note the extra spaces: on either side of the opening parenthesis; on both sides of the assignment and less than operators; and multiple spaces after the semicolons. The multiple spaces after the semicolons make each of the sections of the for statement stand out.

For more complex cases, separate the three sections of a `for` statement to separate lines (see Listing 2).

Long iterator type names can lead to long `for` statements; the multi-line version of the `for` statement is particularly useful there. (Note that the new `auto` keyword should help alleviate some of those problems.)

I see a lot of code where an array index is itself an index into an array – sometimes to a third level. (See first line in Listing 3.)

I find this difficult to parse mentally. When writing array subscripts, I put a space after the opening bracket and one before the closing bracket (see second line in Listing 3).

Aligning types, names, equal signs, and initial values makes each section of a definition stand out

```
for ( int i = 0, j = 0;
      i < some_max_value   &&   j < some_other_max_value;
      ++ i, ++ j )
```

Listing 2

```
array1[array2[array3[index3].index2].index1]

x = array1[ array2[ array3[ index3 ].index2 ].index1 ]
```

Listing 3

Admittedly, it is clearer to pull the nested indices out into their own variables, but I still like to put spaces around the index:

```
index2 = array3[ index3 ].index2;
index1 = array2[ index2 ].index1;
x      = array1[ index1 ];
```

For complex Boolean expressions I tend to use extra spaces to separate sub-expressions:

```
if ( ( x < 0 ) || ( y > 0 ) )
```

At times I'll use extra spaces instead of parenthesis:

```
if ( x < 0   ||   y > 0 )
```

I find either of the previous versions easier to parse than either of these options:

```
if((x<0)||(y>0))
```

or

```
if(x<0||y>0)
```

If you continue reading you will notice that I also put a space after a function name and before the opening parenthesis of the parameter list. This is a holdover from my FORTRAN days (which weren't that long ago) when I used that space to differentiate between a function name (with the space) and an array name (without the space).

```
X = SINE ( ANGLE )
Y = ARRAY( ELEMENT )
```

Blank lines

Like spaces, I use blank lines liberally in my code. I use a blank line to separate variable declarations from other statements in a function, particularly in C where the declarations typically occur at the beginning of a function, or the beginning of a block. I also use a blank line to visually separate different steps in the code, such as the setup of a function call and the call itself.

```
message.header = TYPE;
message.body   = contents;

status = send_message ( &message );
```

A line with nothing but a curly brace is a blank line for this purpose.

```
if ( something )
{
    do_something_else ();
}
```

Putting the brace on the line after the **if** statement provides visual separation for the then clause.

Alignment

I find it easier to read code when certain aspects of the code are aligned. Consider variable definitions:

```
int x=0;
short longer_name=1;
char short_name[2]={0,0};
```

I find this style easier to follow:

```
int   x               = 0;
short longer_name     = 1;
char  short_name[ 2 ] = { 0, 0 };
```

Aligning types, names, equal signs, and initial values makes each section of a definition stand out. The example here is trivial, but recently I've been working on some legacy C code with functions that have 20 or more definitions at the beginning of each function. My eyes started to water when I had to find a particular entry in the list. (It's fixed, now.)

When making multiple assignments, I like to align the equals sign (see Listing 4).

```
structure.element_the_first = 1;
structure.second_element    = 2;
structure.third_element     = function_that_returns_the_needed_value ();
```

Listing 4

iWouldn'tReadTextIfItWasWrittenLikeThis, so why should I have to read variable and function names written this way?

Complex Boolean expressions are a place where spacing and alignment can come into play at the same time.

Who can look at this mess and easily tell what's going on?

```
if(((x==0)&&(x==1))||((y==3)&&((z==5)||(z==6))))
```

Using a combination of spacing and alignment makes it a little easier to parse mentally:

```
if ( ( ( x == 0 ) && ( x == 1 )      ) ||
     ( ( y == 3                 ) &&
       ( ( z == 5 ) || z == 6 ) )    )    )
```

Camel case

I don't like writing or reading names in camel case. iWouldn'tReadTextIfItWasWrittenLikeThis, so why should I have to read variable and function names written this way? We can't embed spaces in names (unless you're still writing in FORTRAN, which deletes whitespace prior to parsing), so the best we can do is use the underscore. `some_max_value` is easier to read than `SomeMaxValue`, and the longer and more descriptive the name the easier it is.

Braces and indentation

I used to think that three-space indentation was just about perfect. One or two spaces aren't enough visually, and four or more pushed code way too far to the right, particularly on old 80-character wide TTY terminals.

```
if ( x == 0 )
   {
   process_x_when_0 ();
   }
else
   {
   process_x_when_not_0 ();
   }
```

Note that using three space indentation, and indenting the braces, causes the `if` and `else` blocks to line up under the opening parenthesis of the `if` statement (assuming you put a space between the `if` and the opening parenthesis). I found this visually appealing, and I wrote a lot of code that way.

I've since adopted a four space indentation, with the braces lined up with the `if` and `else` keywords, for one major reason – it is the common indentation used by my major customer (they have no standards), and it is easier in this case to adapt than be the odd man out. Plus, taking the advice of Bob Martin, I don't write functions with large, heavily indented `if-then-else` trees anymore [Martin09]. (But note that Bob is against the aligning rules I use, so he's not right about *everything*.)

```
if ( x == 0 )
{
    process_x_when_0 ();
}
else
{
    process_x_when_not_0 ();
}
```

I really don't like this style:

```
if ( x == 0 ) {
    process_x_when_0 ();
}
else {
    process_x_when_not_0 ();
}
```

I like putting my opening and closing braces on separate lines, for two reasons: they are always easy to find, and the line with nothing on it but the brace provides a line of whitespace around the statements that make up the `then` and `else` clauses.

MISRA standards [MISRA13] require that braces be used around `if` and `else` clauses even if there is only one statement in the clause (as in each of the three previous examples). I try not to write code that looks like this:

```
if ( x == 0 )
    process_x_when_0 ();
else
    process_x_when_not_0 ();
```

The standard is meant to eliminate problems introduced in maintenance:

```
if ( x == 0 )
    process_x_when_0 ();
else
    process_x_when_not_0 ();
    process_something_else (); // ALWAYS EXECUTED
```

Or with nested `if` statements (I have seen an error of this type just recently):

```
if ( x == 0 )
    if ( y == 0 )
        process_x_and_y_are_0 ();
else
    process_x_is_not_0 ();
```

This only works the way you want it to with properly applied braces:

```
if ( x == 0 )
{
    if ( y == 0 )
    {
        process_x_and_y_are_0 ();
    }
}
else
{
    process_x_is_not_0 ();
}
```

Dan explained to me that I wasn't the person I should be writing it for, and I've used his style ever since

I admit that I do, sometimes, omit the braces, based on the context of the code. I'm working on forcing myself to always include them.

Tabs

If you don't want to over-work your thumbs hitting the space bar, by all means use the tab key, but please, set your editor to replace tabs with spaces. Two things frustrate me when it comes to the tab character remaining in code. I have to figure out what tab setting was used to begin with (seconds of my life I will never get back), and if I'm editing a line and delete a tab I now have to re-add spaces I may not have wanted to delete.

const

Where, exactly, should **const** be placed in a definition or declaration?

The most common placement of **const** is (was?) as in Listing 5.

Dan Saks is an advocate of placing **const** in a way that makes it easy to read the declaration, when read from right to left [Saks88] (see Listing 6).

In a conversation with Dan, I mentioned that I was using the other style, because in general I didn't find it difficult to parse mentally. Dan explained to me that I wasn't the person I should be writing it for, and I've used his style ever since. (I can be persuaded.)

So what does this have to do with whitespace? As you can see, in either style I like to line up the **const** keywords in a block of declarations. I think it's an important part of a declaration, and the alignment makes the **const** (or lack of it) pop out. (And line up the comments, too; it makes them easier to read.) I treat the **volatile** keyword the same way.

```
const int        i;    // i is a const int
      int* const p1;   // p1 is a const ptr to int
const int*       p2;   // p2 is a ptr to a const int
const int* const p3;   // p3 is a const ptr to a const int
```

Listing 5

```
int  const        i;   // i is a const int
int* const        p1;  // p1 is a const ptr to int
int  const*       p2;  // p2 is a ptr to a const int
int  const* const p3;  // p3 is a const ptr to a const int
```

Listing 6

```
int result = function_name ( first_parameter, second_parameter );
```

Listing 7

```
int  not_aligned_like_this ( int     first_parameter,
                             String  second_parameter )

int  new_function_name ( int   first_parameter,
                              short second_parameter )
```

Listing 8

Function parameters

I write function prototypes like this:

```
int function_name ( int        first_parameter,
                    double     second_paramter,
                    CLASS_NAME third_parameter );
```

When calling a function I will format it in a similar way. Having each parameter on a separate line makes them easy to distinquish from one another, and it is easy to add a comment to a parameter (which I find particularly useful when calling one of Microsoft's multi-parameter SDK functions):

```
int result = function_name ( first_parameter,
                             second_parameter,
                             class_parameter );
```

There are some exceptions. For functions with only two parameters, I tend to put them on the same line as the function name; two parameters are not hard to pick out on one line (Listing 7).

I do the same thing for standard library and STL functions and member functions, regardless of the number of parameters, because the parameter lists for these functions tend to be well known.

In his presentation 'Seven Ineffective Coding Habits of Many Programmers', Kevlin Henney [Henney14] talks about "unsustainable spacing" (approximately minute 24), and rejects the idea of certain styles of aligning code as being unmaintainable. He makes the point that maintaining certain styles (mine being one) is "doomed to failure" unless the code never changes. He uses as an example changing the name of a function.

Changing the name of the function breaks the alignment (see Listing 8).

Kevlin suggests the style in Listing 9, or something similar, because performing a refactoring such as changing the name doesn't break the alignment.

I see Kevlin's point, but I'm not … wait, hold the presses.

I have had a rough time with this section on function parameter placement. As I said in my opening note, this is nothing but an opinion piece, and my goal was to document the way I do things, give an explanation as to why I do it that way, and leave it up to you, dear reader, to ignore what I have

It is not easier to write code in this style ... the payoff occurs when the code is read

```
int  original_function_name (
    int   first_parameter,
    short second_parameter )

int  new_function_name (
    int   first_parameter,
    short second_parameter )
```

Listing 9

to say and do it the way you want. That's fine; I don't really expect to sway many opinions. (Maybe one person? Anybody? <crickets>)

The problem occurred when I tried to pick a rhetorical fight with Kevlin with regard to placement and alignment of function parameters. I really don't like the style Kevlin advocates in his presentation. There is something about starting the parameter list on the line after the function call I find aesthetically unpleasing. I tried five or six times to justify using my style, and not his, but I kept running into this wall.

> With all due respect to Kevlin, just how often is a function name changed over a large code base? I don't work on any code base that easily allows that type of change. Changing a widely used function name isn't something to be taken lightly, and if you do it you should be prepared to do it correctly. (OK, Bob, but what are you going to do when you start working on a code base that *does* allow those types of changes, and actively encourages them?)
>
> I see Kevlin's point, but using the shortcomings of our current tool set to advocate use of such an ugly style is wrong. (Compromise isn't a dirty word, Bob.)

And on it went.

In his presentation Kevlin asks a rhetorical question, "Why do you choose an approach that is difficult to maintain?" I do it because I feel it is the right thing to do. I truly believe if we're not actively improving our code we are passively making it worse. I consider it part of doing a good job, and nobody said it would be easy.

Then Kevlin followed up with "Do your colleagues also do what you do?" Sadly, the answer is no. And that is where I kept getting tripped up in my counter-argument.

So, after all of the false starts I realized that I do see Kevlin's point, and most of my justifications boiled down to "this is the way I have always done it, I like it that way, and I don't want to change". Not exactly the end to the rhetorical fight I envisioned when I started this section.

Now the question is, will I actually change the way I write function prototypes and calls? The answer is, I don't know. I will try to come up with a style that satisfies my need for readability and consistency, and Kevlin's need for sustainability. It's the aesthetic issue that is going to be the problem. (Cue cognitive disconnect in five, four, three…)

Conclusion

Most of us, if not all of us, spend a lot more time reading code than writing it. My chosen style isn't more efficient on the writing end; on the contrary, I would say that it takes a bit more time. It is not easier to write code in this style – I would say that it requires a bit more discipline. I believe the payoff occurs when the code is read, by myself or by others.

Thirty years ago, when hard drives were expensive, programmers were cheap, and programmer's terminals had 23 lines and 80 columns, it might have made sense to use the fewest number of characters possible to implement code. Here we are in 2015, where terabyte drives are almost trivially cheap, programmers are expensive, and we are no longer using TTYs. Making use of whitespace to make code easier to read for ourselves and our colleagues makes sense – to me, at least. ■

Acknowledgements

As always, my thanks to Fran and the reviewers. You read this and published it anyway. And a special thank you to Kevlin Henney, who didn't volunteer for the rhetorical fight, but easily won it, anyway.

References

[Henney14] 'Seven Ineffective Coding Habits of Many Programmers', Henney, Kevlin, NDC 2014 http://vimeo.com/97329157

[Kernighan76] *Software Tools,* Kernighan, Brian and Plauger, P.J., Addison-Wesley Professional, 1976

[Martin09] *Clean Code,* Martin, Robert C., Prentice Hall, 2009

[MISRA13] 'Guidelines for the use of the C language in critical systems', MISRA, http://www.misra-c.com/

[Saks88] 'Placing const in Declarations', Saks, Dan, *Embedded Systems Programming*, June 1988

Exercises

'Pinky Finger Workout', typingweb, https://www.typingweb.com/tutor/lesson/index/id/357/

'Thumb Exercises: Active Motion', Northwestern Memorial Hospital, http://www.nmh.org/ccurl/275/700/thumb-exercises-active.pdf

Faking C Function with fff.h

Faking functions for testing in C can ease testing. Mike Long overviews a micro-framework for mocking.

I have a little micro-framework called fff.h for generating fake functions (sometimes these types of functions are called mocks) in C. The basic premise is that testing a C source file is difficult in idiomatic C because of all the external function calls that are hardwired into the production code. The way fff.h helps is to make it a one-liner to create fake implementations of these for the purposes of testing.

Let me give an example. In my last C project, the basic formula for testing a C module was like Listing 1.

```
extern "C"
{
  #include "driver.h"
  #include "registers.h"
}
#include "../../fff.h"
#include <gtest/gtest.h>

extern "C"
{
  static uint8_t readVal;
  static int readCalled;
  static uint32_t readRegister;
  uint8_t IO_MEM_RD8(uint32_t reg)
  {
    readRegister = reg;
    readCalled++;
    return readVal;
  }
  static uint32_t writeRegister;
  static uint8_t writeVal;
  static int writeCalled;
  void IO_MEM_WR8(uint32_t reg, uint8_t val)
  {
    writeRegister = reg;
    writeVal = val;
    writeCalled++;
  }
}

TEST(Driver,
When_writing_Then_writes_data_to_DRIVER_OUTPUT_RE
GISTER)
{
  driver_write(0x34);
  ASSERT_EQ(1u, writeCalled);
  ASSERT_EQ(0x34u, writeVal);
  ASSERT_EQ(DRIVER_OUTPUT_REGISTER,
writeRegister);
}
```

Listing 1

```
TEST(Driver,
When_reading_data_Then_reads_from_DRIVER_INPUT_RE
GISTER)
{
  readVal = 0x55;
  uint8_t returnedValue = driver_read();
  ASSERT_EQ(1u, readCalled);
  ASSERT_EQ(0x55u, returnedValue);
  ASSERT_EQ(readRegister, DRIVER_INPUT_REGISTER);
}
```

Listing 1 (cont'd)

Now, this is a simple example but it illustrates the method. As you can see, most of the test code is taken up writing the fake functions and their associated data members. When modules have many dependencies, I would find myself having to write hundreds of lines of code to get anything compiled and ready to test (did I forget to mention this was legacy code :-)).

After a while of doing this, I started to see a pattern appear. Nearly every fake followed the same pattern, and they all needed to capture the same information. Around this time, Jon Jagger had written some interesting blog posts on using the C preprocessor to do fun things like count. It got me wondering, could I generate the fake code I wanted with the preprocessor?

I played around with different approaches, but soon I had something basic working - I could write a macro that would generate a fake function. After a bit of extra help from Tore Martin Hagen and Jon Jagger I had generalized it to be able to count the number of arguments it would require, and generate the correct code at compile time. A few more iterations, and Listing 2 is the updated code.

Now, you might think that there is not much difference between the two options, and you are correct. By creating the Fake Function Framework I can only save 20% less code, big deal. But that misses a few points:

- Every fake is defined in a standard way
- Fakes can be defined in C or C++ file with correct extern wrappers
- More complex dependencies save more coding

And beyond that, there are a bunch of additional features you get when you define your fakes using `fff.h`:

Function call history with arguments

Say you want to test that a function calls **`functionA`**, then **`functionB`**, then **`functionA`** again, how would you do that? Well `fff.h` maintains

Mike Long is an independent software consultant based in Oslo, Norway. He specialises coaching and mentoring teams adopting modern technical practices in hostile embedded and legacy environments. He is a regular speaker at international conferences in Europe and Asia, and founded the Beijing Software Craftmanship Meetup. You can contact him on twitter at @meekrosoft

```
extern "C"{
  #include "driver.h"
  #include "registers.h"
}
#include "../../fff.h"
#include <gtest/gtest.h>

DEFINE_FFF_GLOBALS;

FAKE_VOID_FUNC(IO_MEM_WR8, uint32_t, uint8_t);
FAKE_VALUE_FUNC(uint8_t, IO_MEM_RD8, uint32_t);

class DriverTestFFF : public testing::Test
{
public:
  void SetUp()
  {
    RESET_FAKE(IO_MEM_WR8);
    RESET_FAKE(IO_MEM_RD8);
    FFF_RESET_HISTORY();
  }
};
TEST_F(DriverTestFFF,
When_writing_Then_writes_data_to_DRIVER_OUTPUT_RE
GISTER)
{
  driver_write(0x34);
  ASSERT_EQ(1u, IO_MEM_WR8_fake.call_count);
  ASSERT_EQ(0x34u, IO_MEM_WR8_fake.arg1_val);
  ASSERT_EQ(DRIVER_OUTPUT_REGISTER,
    IO_MEM_WR8_fake.arg0_val);
}
TEST_F(DriverTestFFF,
When_reading_data_Then_reads_from_DRIVER_INPUT_RE
GISTER)
{
  IO_MEM_RD8_fake.return_val = 0x55;
  uint8_t returnedValue = driver_read();
  ASSERT_EQ(1u, IO_MEM_RD8_fake.call_count);
  ASSERT_EQ(0x55u, returnedValue);
  ASSERT_EQ(IO_MEM_RD8_fake.arg0_val,
    DRIVER_INPUT_REGISTER);
}
```

Listing 2

a call history and also stores the history of function arguments so that it is easy to assert these expectations.

Listing 3 shows how it works.

Of course, if you wish to control how many calls to capture for argument history you can override the default by defining it before include the fff.h like this:

```
  // Want to keep the argument history for 13 calls
  #define FFF_ARG_HISTORY_LEN 13
  // Want to keep the call sequence history for
  // 17 function calls
  #define FFF_CALL_HISTORY_LEN 17

  #include "../fff.h"
```

Function return value sequences

Often in testing we would like to test the behaviour of sequence of function call events. One way to do this with fff is to specify a sequence of return values with for the fake function. It is probably easier to describe with an example (see Listing 4).

By specifying a return value sequence using the `SET_RETURN_SEQ` macro, the fake will return the values given in the parameter array in sequence. When the end of the sequence is reached the fake will continue to return the last value in the sequence indefinitely.

```
TEST_F(DriverTestFFF,
Given_revisionB_device_When_initialize_Then_enabl
e_peripheral_before_initial
izing_it)
{
  // Given
  IO_MEM_RD8_fake.return_val = HARDWARE_REV_B;
  // When
  driver_init_device();

  // Then
  // Gets the hardware revision
  ASSERT_EQ((void*) IO_MEM_RD8,
    fff.call_history[0]);
  ASSERT_EQ(HARDWARE_VERSION_REGISTER,
    IO_MEM_RD8_fake.arg0_history[0]);
  // Enables Peripheral
  ASSERT_EQ((void*) IO_MEM_WR8,
    fff.call_history[1]);
  ASSERT_EQ(DRIVER_PERIPHERAL_ENABLE_REG,
    IO_MEM_WR8_fake.arg0_history[0]);
  ASSERT_EQ(1, IO_MEM_WR8_fake.arg1_history[0]);
  // Initializes Peripheral
  ASSERT_EQ((void*) IO_MEM_WR8,
    fff.call_history[2]);
  ASSERT_EQ(DRIVER_PERIPHERAL_INITIALIZE_REG,
    IO_MEM_WR8_fake.arg0_history[1]);
  ASSERT_EQ(1, IO_MEM_WR8_fake.arg1_history[1]);
}
```

Listing 3

Custom return value delegate

You can specify your own function to provide the return value for the fake. This is done by setting the `custom_fake` member of the fake. Listing 5 is an example.

Under the hood

So how does this all work under the hood? Let's take a look at an example:

```
  // faking "long longfunc(long argument);"
  FAKE_VALUE_FUNC(long, longfunc0, long);
```

This expands to create a function declaration with its associated capture variables, and a function definition (see Listing 6).

These macros can be used separately if you want to put the declarations in a header file and definitions in a sharable module.

In the declaration macro, we declare a struct and a function with C linkage (see Listing 7).

The implementation of the fake function is defined with the macro in Listing 8.

```
// faking "long longfunc();"
FAKE_VALUE_FUNC(long, longfunc0);

TEST_F(FFFTestSuite,
  return_value_sequences_exhausted)
{
  long myReturnVals[3] = { 3, 7, 9 };
  SET_RETURN_SEQ(longfunc0, myReturnVals, 3);
  ASSERT_EQ(myReturnVals[0], longfunc0());
  ASSERT_EQ(myReturnVals[1], longfunc0());
  ASSERT_EQ(myReturnVals[2], longfunc0());
  ASSERT_EQ(myReturnVals[2], longfunc0());
  ASSERT_EQ(myReturnVals[2], longfunc0());
}
```

Listing 4

```
#define MEANING_OF_LIFE 42
long my_custom_value_fake(void)
{
  return MEANING_OF_LIFE;
}
TEST_F(FFFTestSuite,
when_value_custom_fake_called_THEN_it_returns_cus
tom_return_value)
{
  longfunc0_fake.custom_fake =
    my_custom_value_fake;
  long retval = longfunc0();
  ASSERT_EQ(MEANING_OF_LIFE, retval);
}
```

Listing 5

```
#define FAKE_VALUE_FUNC1(RETURN_TYPE, \
                         FUNCNAME, ARG0_TYPE) \
DECLARE_FAKE_VALUE_FUNC1(RETURN_TYPE, \
                         FUNCNAME, ARG0_TYPE) \
DEFINE_FAKE_VALUE_FUNC1(RETURN_TYPE, \
                        FUNCNAME, ARG0_TYPE) \
```

Listing 6

Counting with the preprocessor

The curious among you might be wondering about how the preprocessor does counting. Well, the macros are in Listing 9.

You can learn more about this technique in the resources section at the end of this article.

Summary

The goal of the Fake Function Framework is:

- to make it easy to create fake functions for testing C code
- to be simple – you just download a header file and include include it in your project, there are no fancy build requirements or dependencies of any kind
- to work seamlessly in both C and C++ test environments.

Acknowledgements

The fake function framework would not exist as it does today without the support of key people. Tore Martin Hagen (and his whiteboard), my partner-in-crime in Oslo, was instrumental during the genesis of fff. Jon Jagger, who during ACCU 2011 helped me teach the preprocessor to count. James Grenning, who convinced me the value of global fakes, sent me a prototype Implementation, and showed me how expressive a DSL can be. Micha Hoiting helped me to add support for const arguments. Thanks to you all! ■

```
#define DECLARE_FAKE_VALUE_FUNC1(RETURN_TYPE, \
           FUNCNAME, ARG0_TYPE) \
  EXTERN_C \
  typedef struct FUNCNAME##_Fake { \
    DECLARE_ARG(ARG0_TYPE, 0, FUNCNAME) \
    DECLARE_ALL_FUNC_COMMON \
    DECLARE_VALUE_FUNCTION_VARIABLES \
      (RETURN_TYPE) \
    RETURN_TYPE(*custom_fake)(ARG0_TYPE arg0); \
  } FUNCNAME##_Fake;\
  extern FUNCNAME##_Fake FUNCNAME##_fake;\
  void FUNCNAME##_reset(); \
END_EXTERN_C \
```

Listing 7

```
#define DEFINE_FAKE_VALUE_FUNC1(RETURN_TYPE, \
   FUNCNAME, ARG0_TYPE) \
  EXTERN_C \
    FUNCNAME##_Fake FUNCNAME##_fake;\
    RETURN_TYPE FUNCNAME(ARG0_TYPE arg0){ \
      SAVE_ARG(FUNCNAME, 0); \
      if(ROOM_FOR_MORE_HISTORY(FUNCNAME)){\
        SAVE_ARG_HISTORY(FUNCNAME, 0); \
      }\
      else{\
        HISTORY_DROPPED(FUNCNAME);\
      }\
      INCREMENT_CALL_COUNT(FUNCNAME); \
      REGISTER_CALL(FUNCNAME); \
      if (FUNCNAME##_fake.custom_fake) return \
        FUNCNAME##_fake.custom_fake(arg0); \
      RETURN_FAKE_RESULT(FUNCNAME) \
    } \
    DEFINE_RESET_FUNCTION(FUNCNAME) \
  END_EXTERN_C \
```

Listing 8

Resources

If you have any questions, drop me a line on twitter @meekrosoft.

To learn more you might want to check out some of these resources:

- The project has lots of example code and documentation on Github – https://github.com/meekrosoft/fff
- James Grenning explains how to fake an RTOS with fff.h – http://www.renaissancesoftware.net/blog/archives/303
- Strongminds blog introduction – http://blog.strongminds.dk/post/2012/08/17/Faked-Function-Framework.aspx
- C macro magic - PP_NARG - http://jonjagger.blogspot.co.uk/2010/11/c-macromagic-ppnarg.html

```
#define PP_NARG_MINUS2(...) PP_NARG_MINUS2_(__VA_ARGS__, PP_RSEQ_N_MINUS2())

#define PP_NARG_MINUS2_(...) PP_ARG_MINUS2_N(__VA_ARGS__)

#define PP_ARG_MINUS2_N(returnVal, _0, _1, _2, _3, _4, _5, _6, _7, _8, _9, _10, _11, _12, _13, _14, _15,
_16, _17, _18, _19, N, ...) N

#define PP_RSEQ_N_MINUS2() 19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

#define FAKE_VALUE_FUNC(...) FUNC_VALUE_(PP_NARG_MINUS2(__VA_ARGS__), __VA_ARGS__)

#define FUNC_VALUE_(N,...) FUNC_VALUE_N(N,__VA_ARGS__)

#define FUNC_VALUE_N(N,...) FAKE_VALUE_FUNC ## N(__VA_ARGS__)
```

Listing 9

How to Write an Article

Submitting an article for publication might seem difficult. Frances Buontempo explains how easy it is.

Who writes articles in *Overload*? Taking names from the list of *Overload* authors on the ACCU website gives around 250 authors, without breaking down joint authorship. The top three authors have been Alan Griffiths, with 57, Kevlin Henney with 56 and Francis Glassborow with 49 articles. Figure 1 shows a histogram of the top few authors. A consistent pattern emerges of a few people who contribute again and again – most regular *Overload* readers have never written an article. Some readers may have a blog, or join in a discussion on accu-general, or discuss something technical over a beer with colleagues one evening, but never get as far was writing it up for the ACCU. The majority of the articles published here are from ACCU members, but from time to time other people submit articles. Indeed, as a peer reviewed journal we are open to submissions from anyone. One of the benefits of such a journal is the feedback process. The article can be improved before being read by the public at large whereas a blog gets the feedback *after* it is published. Don't forget being published in a peer review journal counts as a few extra kudos points.

Idea

How do you decide what to write about? Bear in mind writing is usually a learning process. Even if you think you are the world's leading expert on a particular field, writing it up clearly will find gaps in your knowledge or spark off new ideas. It can be worthwhile to simply write a summary style article, with the latest thinking on a subject you are interested in, perhaps delving way back to its beginnings years ago or simply introducing a new language feature. This will get you, and your readers, up to speed with a way of doing something. Some articles have started life with a question on a discussion group, like accu general. In fact, my first *Overload* article, 'Floating point fun and frolics' [Overload 91], started there. It can make life easier to just have a trail of comments and ideas to summarise if you don't feel you have enough ideas yourself. Alternatively, you can present a new technique, library or even language you have developed yourself. Either way, the audience may expect to see a few references, so it is possible to do further background reading on a subject. If you're not sure whether to submit to *Overload* or *CVu*, try both and see what happens. The main points to bear in mind are

- *Overload* is freely available, so anyone may read it. You may want to just try something in *CVu* first time, but this is not a requirement.
- *Overload* tries to take a more academic tone, so might tend towards more technical content, with more references. Something like 'My first 'Hello World' program in JavaScript' might be more suited to CVu, while something like 'Advanced C++17' might be more suited to *Overload*.

If you do not have a full blown article, but just a sketch of an idea, it is ok to get in touch for some early feedback. We might be able to give a few pointers of further things to research or other ways of doing things. If you do want to submit a full-blown article, try to give it some kind of structure. Simply having an introduction, main work and conclusion is far better than sending in a list of bullet points. Spare a little thought for your target audience. How much background might need fully explaining? What can be covered by a reference and leave them to go read up if required. We are always open to other ideas, including but not limited to letters to the editor, for example if a previous article has set you thinking.

Submission

How do you submit an article to *Overload*? The best approach is to use email: Overload@ACCU.org. An easy to copy format, like Open Office, Word, or just plain text is best, though other formats are acceptable. Any diagrams should be attached as separate scalable graphics, so they can be positioned and sized easily for the final layout. If you are demonstrating code you have written, it might not need listing in its entirety. Enough listings to get the main point across often work well. It is sensible to add a link to the whole codebase, for example a github repository, if relevant, though. We also like to have a short biography, with a contact email. Your readers may get in touch and say 'Thanks'.

Some journals offer a template, expecting submissions in a specific font, with a specific size, number of columns and so on. We don't mind – the formatting and layout will happen later. We won't fuss too much about length either. Approximately one thousand words fill a page. Anything from one page upwards is ok. If your article is a 20 page epic, it might make more sense to split in into two or more mini-articles. This will depend on how many other pages have already been taken up, and where we can find a natural place in which to insert a break. More details on the format and structure can be found in 'Guidelines for contributors' [Overload 80].

Feedback

If your article looks like a plausible candidate, it will get sent round the review team, and you will be emailed back comments. We try to make sure we get a mix of positive encouraging comments, nit-picks over typos and grammar, and suggestions of unclear parts that may need rewording. On top of this basic style feedback, be prepared for the reviewers to point out something you may have missed, for example newer language techniques, more succinct ways of doing things, pre-existing libraries you can use off the shelf. You are allowed to argue back, of course, but this process can make the articles more thorough and you may learn even more during the process. Once in a while, the only feedback simply says, "This is great." Not often, but it can happen. On very few occasions the potential author decides not to take on board the feedback, and the idea is taken no further.

Once all the articles have been reviewed they are sent to the production editor, and you will then receive a proof first-draft, showing the actual layout. At this stage everyone needs to keep their eye open for omissions,

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.

If you do not have a full blown article, but just a sketch of an idea, it is ok to get in touch for some early feedback

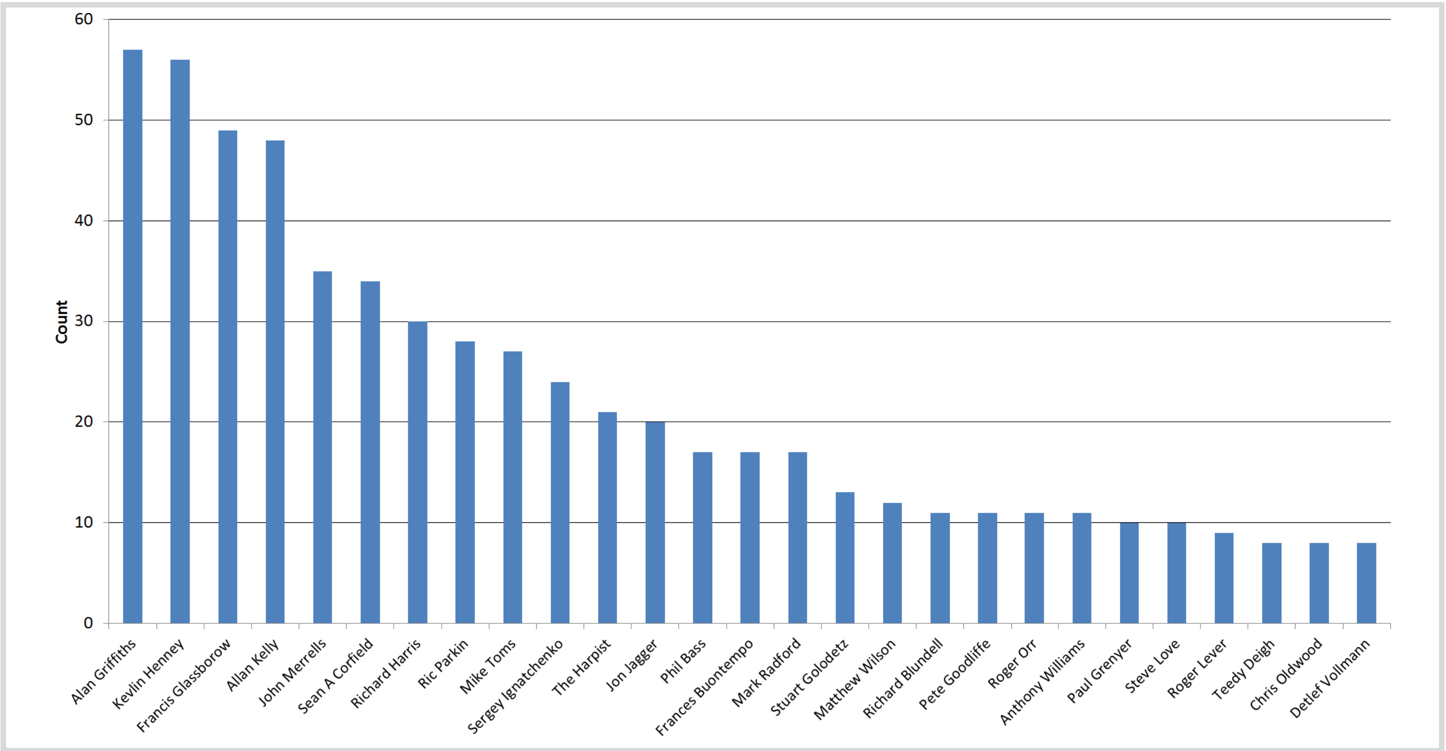


Figure 1

like second author's names, missing diagrams, copy and paste errors and other typos that have slipped under the net. Not matter how hard we try there always seem to be at least one in the final printed version.

Fame

Shortly after the drafts, the whole magazine will be pieced together. You are likely to see an announcement on accu-general, the accu.org webpage and possibly Twitter. Obviously, if you are a member and have paid for it you will get a paper copy through your door at some point and can leave it lying around open on your desk to show off to all your friends and colleagues. If you aren't a member, you can ask for a printed copy – yours to show off and share with others. It might even persuade someone to join. Almost nothing beats the sight of your name in print – try it. ■

References

[Overload 80] http://www.accu.org/index.php/journals/1414

[Overload 91] http://accu.org/index.php/journals/1558

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

Letter to the Editor

I recently received an email regarding an article from an Overload published way back in 2001. It is wonderful to hear people are still looking back through items people have written, no matter how long ago. The email drew my attention to C++11's `std::numeric_limits<T>::lowest()`, which I hadn't come across so I thought others might be in the same situation. Lion has kindly taken the time to formalise his initial observations into the following letter, and I have included a response from the original author, Thaddaeus Frogley.

Dear Editor,

I have just read the *Overload* article 'An introduction to C++ Traits' [Frogley01] and noticed that the initialisation of **`largest`** in the first code fragment (the one adapted from [Veldhuizen]):

```
T largest = std::numeric_limits< T >::min();
```

is done using **`std::numeric_limits<T>::min()`** where it could now be C++11's **`std::numeric_limits<T>::lowest()`** for cases where the type **`T`** is float or double and all array members are below or equal to zero.

Although I couldn't find an explicit reference, this seems to originate from C where **`INT_MIN`** and **`FLT_MIN`** or **`DBL_MIN`** show the same semantic difference; some people guess that it is because the libraries for rational and for integral numbers have been developed by different groups. Fernando Cacciola explains it very nicely in the N1880 proposal to the C++ standard [N1880]:

> numeric_limits::min() (18.2.1.2) is defined with a meaning which is inconsistent across integer and floating-point types. Specifically, for integer types, it is the minimum finite value whereas for floating point types it is the minimum positive normalized value. The inconsistency here lies in the interpretation of minimum: in the case of integer types, it signifies lowest, while for floating point types, it signifies smallest non-zero.

N2348 adds some history [N2348]:

> At Mont Tremblant, Pete Becker noted that the wording was flawed, because not all floating point representations have the feature that the most negative representable finite value is the negative of the most positive representable finite value.

An example (Listing 1) shows the different behaviour.

Because **`std::lowest()`** is not available in C++98, an alternative might be needed. I created a second example which uses the traits technique (see Listing 2).

Please note that it could be possible to distinguish at compile time between pre and post C++11 so that newer compilers could be routed to **`lowest()`** directly, but I couldn't find a short and clean way to do that, although **`#if (__cplusplus > 199711L)`** would come pretty close for many compilers.

Regards,

Lion Gutjahr

Lion Gutjahr wrote his first piece of code on a C64 he got for his 8th birthday, and he hasn't stopped programming since. He has 20 years of professional experience in the industry and service sectors and loves to express himself in C++ during his spare time. Lion can be contacted at lion.gutjahr@gmx.net

```
#include <limits>
#include <iostream>

template< class T >
T findMax(const T * data, const size_t numItems)
{
  // Obtain the minimum value for type T
  T largest = std::numeric_limits< T >::min();
  for (unsigned int i = 0; i<numItems; ++i)
    if (data[i] > largest)
      largest = data[i];
  return largest;
}

template< class T >
T findMaxNew(const T * data, const size_t
numItems) {
  // Obtain the minimum value for type T
  T largest = std::numeric_limits< T >::lowest();
  for (unsigned int i = 0; i<numItems; ++i)
    if (data[i] > largest)
      largest = data[i];
  return largest;
}
template< class T >
void printAry(const T * data, const size_t
numItems) {
  for (unsigned int i = 0; i<numItems; ++i)
    std::cout << data[i] << " ";
  std::cout << std::endl;
}
void test(float* ary, size_t nof) {
  printAry(ary, nof);
  float maxFloat = findMax<float>(ary, nof);
  float newMaxFloat =
    findMaxNew<float>(ary, nof);
  std::cout << "max in only negative floats is "
            << maxFloat;
  if (std::abs(maxFloat - newMaxFloat) >
    std::numeric_limits<float>::epsilon())
      std::cout << " but should have been "
                << newMaxFloat;
  std::cout << std::endl << std::endl;
}

int main() {
  float onlyNegativeFloats[] = { -2, -1, -3 };
  std::cout << "array of only negative floats: ";
  test(onlyNegativeFloats, 3);
  float positiveAndNegativeFloats[] = {
    2, -1, -3 };
  std::cout << "array of positive and negative
    floats: ";
  test(positiveAndNegativeFloats, 3);
  return 0;
}
```

Listing 1

```
#include <cstdlib>
#include <cmath>
#include <limits>
#include <iostream>
#include <string>

template< class T >
T findMax(const T * data, const size_t numItems)
{
  // Obtain the minimum value for type T
  T largest =
    std::numeric_limits< T >::min();
  for (unsigned int i = 0; i<numItems; ++i)
    if (data[i] > largest)
      largest = data[i];
  return largest;
}

namespace detail {
  template< class T, bool isFloat >
  struct cpp98_numeric_limits_lowest {};
  template< class T >
  struct cpp98_numeric_limits_lowest<T, true> {
    static T value() {
      return -std::numeric_limits<T>::max(); }
  };
  template< class T >
  struct cpp98_numeric_limits_lowest<T, false> {
    static T value() {
      return std::numeric_limits<T>::min(); }
  };
} // end namespace detail

template< class T >
T cpp98_numeric_limits_lowest() {
  return detail::cpp98_numeric_limits_lowest< T,
    std::numeric_limits<T>::is_specialized &&
    !std::numeric_limits<T>::is_integer>
        ::value();
}

template< class T >
T findMaxNew(const T * data,
  const size_t numItems) {
  // Obtain the minimum value for type T
  T largest = cpp98_numeric_limits_lowest<T>();
  for (unsigned int i = 0; i<numItems; ++i)
    if (data[i] > largest)
      largest = data[i];
  return largest;
}

template< class T >
void printAry(const T * data,
  const size_t numItems) {
  for (unsigned int i = 0; i<numItems; ++i)
    std::cout << data[i] << " ";
  std::cout << std::endl;
}

namespace detail {
  template< typename T, bool isFloat >
  struct areEqual {};
  template< typename T >
  struct areEqual<T, true > {
    static bool value(const T& a, const T& b) {
      return std::abs(a - b) <=
      std::numeric_limits<T>::epsilon(); }
  };
```

Listing 2

```
  template< class T >
  struct areEqual<T, false > {
    static bool value(const T& a, const T& b) {
      return a == b; }
  };
} // end namespace detail

template< class T >
T areEqual(const T& a, const T& b) {
  return detail::areEqual< T,
    std::numeric_limits<T>::is_specialized &&
    !std::numeric_limits<T>::is_integer>
        ::value(a, b);
}

template< class T >
void test(const std::string& desc,
   const T * data, const size_t numItems) {
  std::cout << "array of " << desc << ": ";
  printAry(data, numItems);

  T maxval = findMax<T>(data, numItems);
  T newMaxval = findMaxNew<T>(data, numItems);

  std::cout << "max in " << desc << " is "
    << maxval;
  if (!areEqual<T>(maxval, newMaxval))
    std::cout << " but should have been "
    << newMaxval;
  std::cout << std::endl << std::endl;
}
int main()
{
  {float vals[] = { -2, -1, -3 };
  test<float>("only negative floats", vals, 3); }

  {float vals[] = { 2, -1, -3 };
  test<float>("positive and negative floats",
    vals, 3); }

  {double vals[] = { -2, -1, -3 };
  test<double>("only negative doubles",
    vals, 3); }

  {double vals[] = { 2, -1, -3 };
  test<double>("positive and negative doubles",
    vals, 3); }

  {long double vals[] = { -2, -1, -3 };
  test<long double>("only negative long doubles",
    vals, 3); }

  {long double vals[] = { 2, -1, -3 };
  test<long double>("positive and negative long
    doubles", vals, 3); }

  {int vals[] = { -2, -1, -3 };
  test<int>("only negative ints", vals, 3); }

  {int vals[] = { 2, -1, -3 };
  test<int>("positive and negative ints", vals,
    3); }

  {unsigned short vals[] = { 2, 3, 1 };
  test<unsigned short>("unsigned shorts", vals,
    3); }

  return 0;
}
```

Listing 2 (cont'd)

References

[Frogley01] Frogley, Thaddaeus 'An introduction to C++ Traits', *Overload* Journal #43 (June 2001)

[N1880] N1880 proposal to the C++ standard http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1880.htm

[N2348] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2348.pdf

Response from Thaddaeus

If anything, it highlights the need to introduce lowest, if there is (was) no clear and straightforward way to achieve the same goal in the C++ of 2001, noting that **-max** doesn't work for unsigned integers!

My preferred solution these days would to write something like this (and this is also a more language agnostic approach):

```
template< class T >
T findMax(const T* data, int numItems)
{
    assert(numItems>0);
    // Obtain the minimum value for type T
    int i=0;
    T largest = data[i++];
    for (;i < numItems; ++i)
        if (data[i] > largest)
            largest = data[i];
    return largest;
}
```

which avoids the problem altogether, provided **numItems != 0**.

If the current system is yours, that's a lost customer

Good product information is about helping your customers reach their goals

Clearly stated...

We can help you with:

- product manuals/user guides
- online help
- training materials (including e-learning)
- bids and proposals...

For help turning frustrated customers into product champions, get in touch.

T 0115 8492271 E info@clearly-stated.co.uk W www.clearly-stated.co.uk

We are members of the Institute of Scientific and Technical Communicators, the UK professional body for technical authors. For information about the ISTC, visit www.istc.org.uk