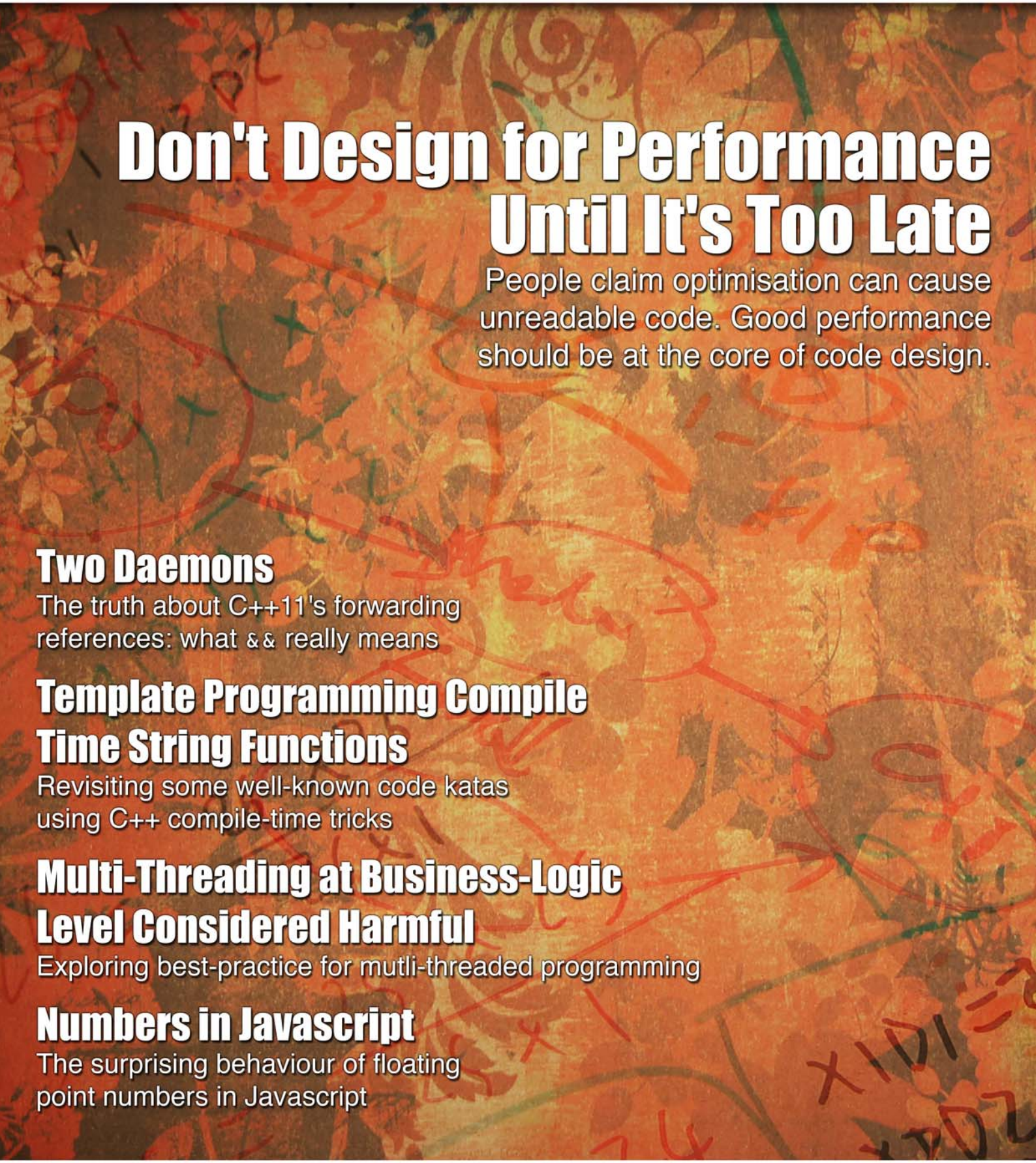
overload 128

AUGUST 2015 £3



Don't Design for Performance Until It's Too Late

People claim optimisation can cause unreadable code. Good performance should be at the core of code design.

Two Daemons

The truth about C++11's forwarding references: what && really means

Template Programming Compile Time String Functions

Revisiting some well-known code katas using C++ compile-time tricks

Multi-Threading at Business-Logic Level Considered Harmful

Exploring best-practice for mutli-threaded programming

Numbers in Javascript

The surprising behaviour of floating point numbers in Javascript

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 128

August 2015

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 129 should be submitted by 1st September 2015 and those for Overload 130 by 1st November 2015.

The ACCU

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

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

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

Copyrights and Trade Marks

Semi-automatic Weapons

Automating work can save time and avoid mistakes. But Frances Buontempo doesn't think you should script everything.

Having been distracted by an esteemed member's [@ChrisOldwood] tirade of gags and puns on twitter, I thought I might attempt to tell a joke myself.

> "Doctor, doctor, it hurts when I jab myself in the eye with a pencil."
> "Have you tried automating this?"

As you can see I am not very good at jokes. I did spend some time trying to think up others, and as ever this has put paid to any hope of writing an editorial for this issue. In all seriousness, I have been musing on various kinds of automation recently. My dream of automatically generating an editorial remains, but is still just a dream. Instead I have been observing other people trying to automate various activities with various degrees of success. Having also just finished reading *The Thrilling Adventures of Lovelace and Babbage: The (mostly) true story of the first computer* [Padua], I was struck this time by the motivation of the first computer – to replace human computers of logarithmic and other mathematical tables by an automatic calculating machine. Padua suggested that Babbage owned a mechanical ballerina – an automaton – of which he was extremely proud. Perhaps he was inspired by roller-skating Merlin's famous silver swan [Swan]. The steam-punk style mechanical swan gracefully appears to move in a stream and catch a fish. A young Charles Babbage is reported to have seen this and been mesmerised. Merlin showed him a mechanical dancer, which Babbage managed to purchase at auction many years later [Dancer] that he proudly restored and displayed on a glass pedestal in his office. It seems like his youthful enchantment with automata fed his later attempts to build the first computer. These tales also make a delightful steam-punk comic [Padua].

Though certain automata were purely for entertainment, many of Babbage's difference and analytical engines were designed to speed up laborious manual calculations, even though they may not have progressed much beyond the design stage. Other machines appear to be somewhere in between. In a century where people wondered if man could create life or if a woman's vision of Frankenstein creating a monster were a real possibility, one may discover the beginnings of attempts at artificial intelligence. Many years before the Deep Blue [DeepBlue] chess computer, one finds the mechanical Turk [Turk] – an apparent automaton which was really rather good at playing chess. It seems that despite the appearance of being a machine that could automatically play chess, it was in fact being driven manually by, well, a man, not the apparent automatic marvel first promised. Obviously, it actually needed to be driven by a very good chess player, since a machine playing chess poorly would have been somewhat less remarkable. We will return to this theme of something 'automatic' actually being semi-automatic or manual shortly.

While many of the mechanical marvels entertained the bourgeoisie or aristocracy, automation gradually crept into almost all realms of life. In England, we are taught about the rise of the machines and the clog-throwing saboteurs and Luddites. To be fair the saboteurs are possibly French [Saboteur]. The Luddites, sometimes described as machine-breakers, were attempting to bring about a change putting the workers in a better negotiating position with their employers. The word is frequently misused nowadays to mean someone who is afraid of technology. Technophobe would be a better term for such a person, though that could be argued to literally mean one who is afraid of skill – those of us who work with 'jobbing programmers' had better beware! One of the saboteur methods was the withdrawal of efficient working, specifically where workers 'avoid any actions that would hurt their own job prospects' [Method]. You may suspect that failing to do your job efficiently would harm your job prospects, but until we are able to measure efficiency this may be a moot point. Some measures, such as lines of code written have been used previously but tend to encourage people to work to the metrics rather than producing the software which our customers desire and furthermore discourage deleting code.

Getting back on track, it is important to notice that the machines which saboteurs were punished for breaking were not fully automated. They still required some workers to load up the threads for the looms and so on. So many times our automatic tools are simply semi-automatic. An automatic car does not drive itself, yet. Furthermore, many attempts at machine learning or artificial intelligence require a lot of human input. Cordelia Schmid remarked at the BCS' annual Karen Sparck Jones lecture this year that models using hand-tuned parameters are not examples of machines learning [Eigenfaces]. Babbage's difference engine required human-produced punched cards for the instructions. A continuous integration server will do nothing until a human commits some code, apart from time-based runs which still require code in the first place and a human to setup the job. Each automatic set-up requires a human in the loop [HITL].

Having observed that many automated processes are not fully automated, it might be worth stopping to think why we have tried to automate the process in the first place. If you have written a library to automatically generate boiler-plate code to save you hand crafting it, it might be worth pausing and considering why you need so much boiler-plate code. And besides, who hand-writes code these days anyway? Perhaps you are solving the wrong problem. Having to do something over and over again might indicate a deeper underlying problem that should be removed. If your code needs a special linker tool to find all the dependencies it might be better to re-architect your solutions so it doesn't need turtles, I mean dependencies, all the way down. If you use a dependency injection framework to automate composition of code, and end up having to constantly hand-tune miles and miles of xml, you should probably start

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 works at Bloomberg, 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.

wondering if you are going down the wrong path rather than writing a GUI to allow automatic generation of the required configuration once someone has clicked all the right buttons. Semi-automatic weapons might not be the best solution to a computer problem, though we all feel the computer deserves destruction from time to time. Use the right tool for the job, or consider if the job's worth doing.

This brings me neatly on to the 'F' word… I say *the* 'F' word, but there is more than one:

- Framework
- Factory, notice the Luddites at the ready again
- Farm, another aspect involved in the industrial revolution
- Fudge, or perhaps kludge, certainly not very tasty
- FAQs – often written before anyone tries to 'Read the Flipping Manual' so they are often not frequently or indeed ever asked
- Future-proofed – almost never involving any kind of proof
- Functional, just barely, and only on one machine
- Failover, again and again and again
- And others
  - Fortran, function, Fail...
  - Fuzz, futz, fiddle,
  - File, FAT, file-system, factor, foobar, frame, FIFO…

The list is rather long, so let us just focus on the first few. Though the word factory clearly has its origins in the Latin *facere* – to make or do, it also combines the sense of an office for agents or workers in a possibly foreign place or perhaps derives from the idea of an oil press or mill [Etymonline]. Perhaps you should start using IMill for your abstract factory builder patterns from now on. A factory is now used to mean a place where items are manufactured, and the irony of automatically manufacturing something is not lost on me. Notice that word, manufactured, means making something by hand. Is anything truly automated?

The ubiquitous Frameworks are a specific example of semi-automatic weapons. These APIs and libraries are designed to take some of the monotony out of writing code. Various third-party frameworks exist, ranging from Ajax to various middleware and so on. Many large companies end up writing their own as well, to suit their special needs, which will tend to add to the time required to learn how to use the framework, whereas with a third-party solution you have a chance to hire workers who already know how to drive the code. Clearly each approach has pros and cons. Part of the drive behind many frameworks is to provide software reuse, to save people time reinventing the wheel. If your framework is slowing you down, "You're doing it wrong". It is possible to take reuse too far. If you extracted every `for` loop and moved it to a framework, you would end-up tying together lots of different modules that had no reason to be bound together. It is important to step back and consider what problem you are trying to solve before speeding ahead and writing scripts or frameworks for every commonality you spot or imagine. If you do conclude that it is worth making a library, or framework, of reusable components keep Gödel's incompleteness laws in mind.

- in any consistent formal system F within which a certain amount of arithmetic can be carried out, there are statements of the language of F which can neither be proved nor disproved in F ['F' words again]
- such a formal system cannot prove that the system itself is consistent (assuming it is indeed consistent) [Gödel]

It would be worrying to end up with an inconsistent framework, though perhaps as programmers we are less concerned by its incompleteness – just keep churning out more code. If Gödel's theorems seem a little abstract, we could simply consider the entropy in our code. The lower the entropy, the more likely we are to be able to compress reusable parts down to a library, if not a framework [Veldhuizen]. Of course, it will be hard to know in advance how much entropy our code will contain, and yet many people will be tempted to automate something before they have tried it. This flies in the face of advice like 'YAGNI' – you ain't going to need it. It is hard to find the balance between experimentally writing a few scripts that might come in handy and waiting until you know what you really need. Veldhuizen [op cit] draws out an interesting principle:

> Principle 1 (Entropy maximization). Programmers develop domain-specific libraries that minimize the amount of frequently rewritten code for the problem domain. This tends to maximize the entropy of compiled programs that use libraries.

In other words, as we pull out commonality what we are left with has higher entropy or chaos. By trying to make our lives easier, we are causing chaos, which may be no bad thing, but on the face of it this seems remarkable. After some mathematics the author proves "the process of discovering new and useful library components is not a process that can be fully automated". It is not possible to automate everything. Humble human programmers will always be required. Our tools can only ever be semi-automatic, and are frequently hand-crafted. One blog on this paper draws the conclusion "The takeaway is that when trying to create reuse you can probably do it forever so one needs to temper this desire with practicality." [ElegantCoding]

There is nothing wrong with a framework per se, but it is very difficult to write one up front. Martin Fowler talks about 'Harvested Frameworks' [Fowler], whereby you grow or rather harvest a framework from a working application. That allows you to spot commonality after you have written it, instead of guessing upfront. In general, I tend to try to capture repetitive tasks at least as a script, once I had done them three times or so, allowing me to get a feel for what's common and what needs to be configurable. There is nothing wrong with a semi-automatic process. Even if our continuous integration server will run any tests after compiling the code, this won't stop me running the tests before I commit my changes. In our geek-driven search to automate everything, we need space for humans in the loop. If jabbing yourself in the eye hurts, stop it rather than automate it. As Bill Gates is reported to have said

> The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency. [Gates]

References

[Dancer] http://www.hrc.wmin.ac.uk/theory-babbagesdancer2.html

[DeepBlue] https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)

[HITL] (a) Cranor 'A Framework for Reasoning about the Human in the Loop' *Usability, Psychology and Security*, 2008

(b) Rothrock and Narayanan, 'Human-in-the-Loop Simulations. Methods and Practice' *Springer* 2011.

[Eigenfaces] http://buontempoconsulting.blogspot.co.uk/2015/05/eigenfaces-ftw-or-zebranon-zebra.html

[ElegantCoding] http://www.elegantcoding.com/2011/07/software-frameworks-resistance-isnt.html

[Etymonline] http://www.etymonline.com/index.php?term=factory

[Fowler] http://martinfowler.com/bliki/HarvestedFramework.html

[Gates] http://www.investinganswers.com/education/famous-investors/50-quotes-wealthiest-man-america-3088

[Gödel] http://plato.stanford.edu/entries/goedel-incompleteness/

[Method] https://en.wikipedia.org/wiki/Industrial_Workers_of_the_World_philosophy_and_tactics#Sabotage

[Padua] *The Thrilling Adventures of Lovelace and Babbage: The (mostly) true story of the first computer.* Sydney Padua 2015 Pantheon.

[Saboteur] https://en.wikipedia.org/wiki/Sabotage#Etymology

[Swan] http://www.atlasobscura.com/places/silver-swan

[Turk] https://en.wikipedia.org/wiki/The_Turk

[Veldhuizen] *Libraries and their Reuse: Entropy, Kolmogorov complexity, and Zipf's Law*. OOPSLA 2005

Multi-threading at Business-logic Level is Considered Harmful

Multi-threaded code promises potential speed-up. Sergey Ignatchenko considers how it often slows things down instead.

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

Assumption is a mother of all screw-ups
~ Honorable Mr. Eugene Lewis Fordsworthe

For quite a long time (since I first needed to deal with non-trivial multi-threading 15 years ago) I knew that mixing multi-threading (especially thread synchronization) with business logic is a Really Bad Idea and argued for avoiding it whenever possible (see, for example, [NoBugs10]). This notion became so deeply ingrained in my mind, that I've erroneously started to assume that everybody else shares this knowledge (or belief, depending on which side of the argument you are ☺).

As usually happens with most assumptions, Mother Nature has once again proved that I was wrong. Recently I wrote an article on networking for games [NoBugs15], where I took 'mixing multi-threading with business logic is a Bad Idea' as granted; I've had feedback that this is unclear and needs explaining. Ok, here goes the explanation (an outline for this explanation has already been provided in [Ignatchenko15], but this is a much more elaborate version with a few additional twists).

There are four Big Reasons for avoiding handling both business logic and non-trivial multi-threading within the same pieces of code. However, before going into reasons, we need to provide some definitions.

Definitions

In this field, a lot depends on how trivial your multi-threading is. For example, if you have multi-threading where all the synchronization is performed on one single mutex, we can call it 'trivial' (and, as shown below, you're likely to be able to get away with it[1]). However, the window for triviality is very narrow: for example, even going into two interrelated mutexes instead of one, can easily make multi-threading non-trivial (and, as discussed below, has potential to make your life a nightmare).

Another example is trivialized multi-threading (a close cousin of the trivial one); one good example of trivialized multi-threading is when all the inter-thread interactions are made only via queues. It doesn't mean that implementing queues is trivial, but that from the point of view of the developer-who-writes-business-logic, he doesn't need to care about queue implementation details. In other words, the problem is not about having multi-threading within your program, it is about mixing multi-threading synchronization with business logic in the same piece of code.

Now, we're all set to start discussing why you shouldn't intermix non-trivial multi-threading synchronization with business logic.

Reason 1: Cognitive limits of the human brain

In psychology, there is a well-known '7 ± 2' cognitive limit [Wikipedia]. This means that the number of objects an average human can hold in working memory is 7 ± 2.[2] When you go above this limit, you a get kind of 'swapping' (to 'swap out' some entities to free space in your working memory, only to 'swap them back in' when they're needed). And from our programming experience, we all know what swapping does to performance ('slowing down to a crawl' being a very mild description). A similar thing happens when a human being goes beyond his cognitive capacity – the process of solving the problem becomes so slow that often the problem cannot solved at all (unless it can be split into smaller problems, with each of these problems fitting into cognitive limits).

BTW, don't think that as you are not an average person[3], you will be able to process 70 objects or entities instead of the average 7 – you won't; the best you can realistically hope for is 10–15, and this difference won't change our analysis. And even if you have on your team one person with an exceptionally high cognitive limit, you can be sure that it is extremely uncommon, which means that relying on her abilities to maintain your program is a Really Bad Idea. The simple question, "What are we going to do when she leaves?" is enough to bury the idea of relying on One Single Developer (however much of a genius she is).

So, how does this 7 ± 2 limit apply to combining business logic with multi-threading? The answer is simple: for real-world programs, each of these things is already complicated enough and usually is already pushing this "7 ± 2" limit. Combining them together will very likely take you over the limit, which will likely lead to the problem of 'making the program work' becoming unsolvable. Exceeding the limit becomes even more obvious when we observe that when adding multi-threading to business logic, we're loading our brain with not only analysis of readily visible entities such as threads and mutexes, but also with less obvious entities such as how existing business objects will interact with this mutex? With these TWO mutexes? This brings the number of entities even higher, which in turn makes the cognitive overload even worse.

For trivial (and trivialized) multi-threading, this effect, while present, can be seen as adding (very roughly) only one additional entity; while even one additional entity can also bring you over the cognitive limit, it is still much better than having dozens of additional entities in scope. Also, cognitive

'No Bugs' Hare 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 currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

1 Also with a single mutex you can easily lose most (or all) the multi-threading parallelism – depending on your specific task, but this is beyond the scope now.
2 There are arguments in scientific community about exact numbers (one popular number is four), but there is pretty much a consensus that whatever the numbers of entities are – they're single-digit.
3 Of course you aren't, I wouldn't dare to question it

if you have a bug in your multi-threading code, you won't really be able to jump to a certain point in the debugger and see what's going on

limits are not exactly hard limits as in "9 and you're fine, 10 and you're mine", and while one extra entity over the limit would clearly mean reduced overall performance of the developer, it isn't likely to cause 100% drop in performance (so it shouldn't go into the 'problem never solved' area). Therefore, given the very small typical numbers for cognitive limits, while adding even one entity will be noticeable (so is not desirable), it is not very likely to be fatal.

Reason 2: Non-determinism is bad enough, but inherently untestable programs are even worse

We don't know what we have until we lose it
~ proverb

Non-trivial multi-threaded code usually has one property – it is inherently non-deterministic.

By the very definition of pre-emptive multi-threading, context switches happen not when you expect them, but between any two assembler-level instructions (yes, we're not considering disabling interrupts within business logic). On one run of the program, a context switch may happen between lines A and B, and on the next run of the very same program, it may happen between lines B and C (on some runs it may happen even in the middle of a line of code, if it is compiled to more than one assembly instruction). It means that the multi-threaded program MAY become non-deterministic, i.e. it MAY behave differently from one run to another even if all the program inputs are exactly the same.

One may ask, "What is so bad about that?" Unfortunately, this potential non-determinism has several extremely unpleasant implications.

A: Untestability

As you have no way to control context switches, you cannot really test your program.

Your multi-threaded program can pass all of your tests for years, and then, after you've changed a line in one place, a bug in a completely unrelated place (which has existed for all these years, but was hidden) – starts to manifest itself. Why? Just because context switching patterns have shifted a bit, and instead of context switch between lines A and B, you've got a context switch between lines B and C.

In [Ignatchenko98] a multi-threading bug is described, which has manifested itself on a 20-line program which has been specially written to demonstrate the bug, and it took any time between 20ms to 20s (on the very same computer, just depending on the run) for the bug to manifest itself (!). On a larger scale – it was a bug no less than in the Microsoft C++ STL implementation shipped with MSVC (carrying a copyright by no less than P.J. Plauger), and while the bug was sitting there for years and has manifested itself in a real-world environment, the manifestation was usually like "our program hangs about once a month on a client machine with no apparent reason", which is virtually impossible to debug. Only careful analysis of the STL code found the bug (and the analysis wasn't related to any specific problem with any specific program, it was done out of curiosity).

Another example of untestability is as follows. Your program passes all the tests in your test environment, but when you deploy it to the client's computer, it starts to fail. I've observed this pattern quite a few times, and can tell that it is extremely unpleasant for the team involved. The reason for failure is the same – context switch patterns have shifted a bit due to different hardware or due to different load patterns on client's machine.

Bottom line: you cannot rely on testing for multi-threaded programs. Bummer.

B: Irreproducibility

Non-determinism implies that on every program run you get different patterns.

This means that if you have a bug in your multi-threading code, you won't really be able to jump to a certain point in the debugger and see what's going on (nor will you be able to print what happens there, unless you're printing everything in sight over the whole program, which by itself will shift patterns and may mask the bug). Ok, technically you are able to jump to any point of your program, but the variables you see may (and if you have a multi-threaded bug – will) differ every time you jump there.

This makes debugging multi-threaded issues a nightmare. When the bug manifests itself about every 50th run of the program, it is already bad enough for debugging, but when the pattern you see is a bit different every time when it happens – your task of debugging the program can easily become hopeless.

Many of you will say "Hey, I've debugged multi-threaded programs, it works perfectly". Indeed, much debugging works in a multi-threaded environment, and you can debug a multi-threaded program, you just cannot debug subtle multi-threaded issues within your non-trivial multi-threaded program.

To allow for multi-threaded debugging in one of many complicated multi-threaded projects, we went as far as creating our own fiber-based framework which simulated threads, with our own simulated scheduler and switching at the relevant points. Our simulated scheduler was run using a pseudo-random generator, so when seeding it with the same original seed, we've got determinism back, and were able to debug the program. For us, it was the only way to debug that program (!). There are similar tools out there (just Google for "deterministic framework to debug multi threaded program"), and they might help, but while helpful for debugging small primitives, such methods are inherently very time-consuming and most likely will be infeasible for ongoing debugging of your business logic.

C: Need for proofs of work (or exhaustive deterministic testing)

So, we've found (both from theory and illustrated by experience) that no kind of testing can serve as a reasonable assurance that your multi-threaded program will work, and that debugging is likely to be a real problem. Sounds Really Bad, doesn't it? More importantly, can we do something about it?

Any change in business logic is likely to affect non-trivial thread synchronization, which in turn is likely to lead to impossible-to-test and next-to-impossible-to-debug bugs

In practice, I tend to provide proofs of work for any non-trivial multi-threaded code. I've found from experience, that it is the only way to ensure that a multi-threaded program will work 100% of the time (opposed to working 99.99% of the time, which means failing here and there), and will work everywhere.

For small pieces of code (20–50 lines) it is perfectly feasible. The level of formality you need for your proofs is up to you, but it is important at least to convince yourself and somebody else, that with any pattern of switches the piece of code in question will work as expected. One good example of code where more or less formal proofs are feasible (and necessary) is an implementation of the queue for inter-thread communications.

Of course, for thousands-of-lines business logic, such proofs are not feasible (that is, unless you trivialize the interaction of business logic with multi-threading).

An alternative to proofs of work is to use one of those deterministic testing frameworks mentioned above, and to perform exhaustive testing, testing the program behavior for all the possible (or at least relevant, though the notion of 'relevant' requires very careful consideration) context switches. Our own framework (the one mentioned above) did allow such testing, but times for such exhaustive testing were growing at least exponentially as the size (more precisely – number of points of interest where the context switch might be relevant) of the program grew, so once again such exhaustive testing wasn't feasible for the programs with over 20–50 lines of code.

Reason 3: Code fragility

A logical consequence of untestability and the need for proofs of work is code fragility. If, whenever you need to change the program, you need to re-prove that it still works, this cannot be safely entwined with business logic (which, by definition, changes 5 times a day). If, whenever you're changing something, you're afraid that it might break something somewhere 50000 lines of code away, it won't work either.

More formally, a non-trivial mixture of business logic with thread synchronization is inherently fragile. Any change in business logic is likely to affect non-trivial thread synchronization, which in turn is likely to lead to impossible-to-test and next-to-impossible-to-debug bugs.

Reason 4. Context switching granularity

To be efficient, multi-threading programs SHOULD make sure that they don't cause too much context switching (i.e. multi-threading SHOULD be coarse-grained rather than fine-grained). The thing is that context switches are damn expensive (taking into account the cost of recovery from thread caches being flushed out by another thread, think of the order of 10,000 CPU clock ticks on x86/x64).

For example, if you want to move integer addition to another thread, you're likely to spend 20,000 CPU clock ticks for 2 context switches (to another thread and back, with roughly half of the work being in your original thread), and to save 0.75 CPU clocks on offloading the addition. Of course, this is an extreme example, but way too often multi-threading is used without understanding the implications of the cost of the context switches.

In this regard, separating business logic from threading helps to establish a well-defined interface which encourages (though doesn't guarantee) coarse-grained granularity. For example, when having queues for inter-thread communications, it is usually easier to write a coarse-grained program, which is (as a rule of thumb; as with anything else, there are exceptions) a Good Thing.

On the opposite side, code which intermixes business logic and thread synchronization tends to overlook the need to keep granularity in check; while in theory it is possible to handle it properly, in practice adding it to the equation is not feasible, not least because of adding yet another layer of entities, overloading (already overloaded) cognitive limits even further.

Hey, there are working multi-threaded programs out there!

One may say: "Hey, you're saying that writing multi-threaded programs is impossible, but everybody and his dog is writing multi-threaded programs these days!". You do have a point. However:

- Quite a few multi-threaded programs are using trivial multi-threading (for example, with a single mutex). Have you ever seen a multi-threaded program which is able to utilize only 1.2 cores? They're likely using single mutex. And BTW, I cannot blame them as soon as they provide adequate overall performance: if one marketing guy has said "we need to write 'support for multiple cores' on our website, because all the competition does it", a single mutex is one way to do what marketing wants without jeopardizing the whole project.
- Quite a few programs (think of video codecs) do really need to utilize multiple cores, but don't really have much business logic (depends on how you define 'business logic', but at least it doesn't change too often for codecs). They may get away with more or less complicated thread sync, but even for video codecs having per-frame (or per-large-part-of-frame) processing granularity (with clearly defined inter-thread interfaces such as queues) tends to work better than alternatives.
- Quite a few multi-threaded programs out there do have those difficult-to-find-and-debug bugs. This is especially true for those programs which don't have a multi-million install base [Wikipedia-2], but having a large install base certainly doesn't guarantee that the program is multi-threaded-bug-free. I would guesstimate that for those programs which are released (i.e. out of the development shop), at least 50% of crashes are related to multi-threading.
- And finally, there are programs out there which do follow the principles outlined in the next section, 'Divide and conquer'.

Divide and conquer

The first step in solving a problem is to recognize that it does exist
~ Zig Ziglar

Despite the 'Divide and conquer' concept (originally Latin *Divide et impera*) coming from politics, it is still useful in many fields related to engineering, and is usually a Good Thing to use in the context of programming (not to be confused with programming team management!).

Jokes aside, if we can separate business logic from non-trivial multi-threading (trivializing multi-threading interaction from the point of view of business logic), we will be able to escape from (or at least heavily mitigate) all the problems described in this article. The number of entities to fit into cognitive limits will come back to reasonable numbers, business logic will become deterministic again (and while multi-threading synchronization will still require proofs of work, they are feasible for small and almost-never-changing pieces of code), code will be decoupled and will become much less fragile, and coarse-grained granularity will be encouraged.

The only teensy-weensy question remaining is "how to do it". There are several approaches to start answering this question, and I hope to describe one of them sooner rather than later ☺. For now, we need to recognize that we do have a problem, solving it is the next step. ■

References

[Ignatchenko15] Sergey Ignatchenko, 'Three Reasons to Avoid Intermixing Business Logic and Thread Synchronization', http://java.dzone.com/articles/three-reasons-avoid

[Ignatchenko98] Sergey Ignatchenko, 'STL Implementations and Thread Safety', *C++ Report*, July/Aug 1998

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

[NoBugs10] 'No Bugs' Hare, 'Single-Threading: Back to the Future?', http://accu.org/index.php/journals/1634

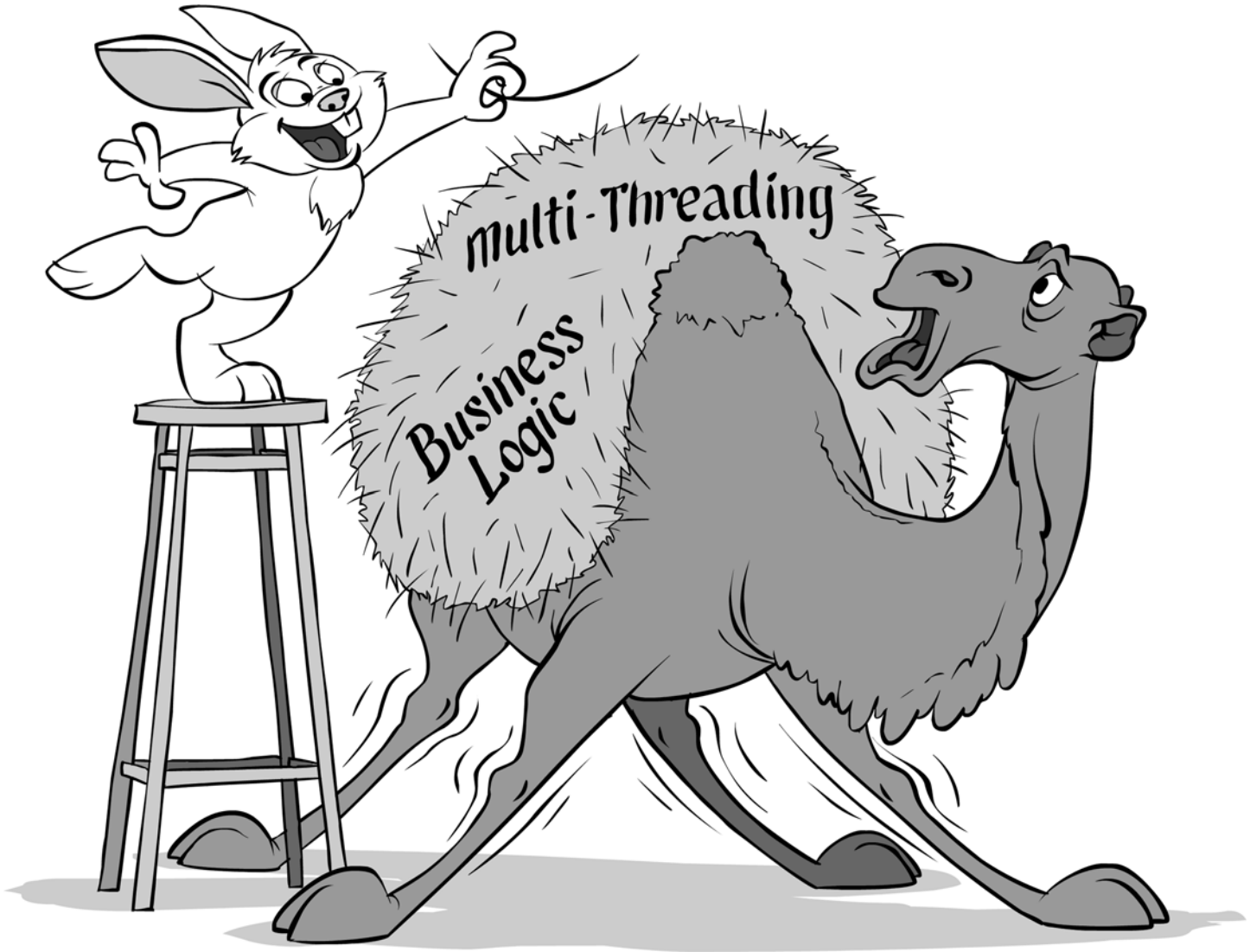
[NoBugs15] 'No Bugs' Hare, '64 Network DO's and DON'Ts for Game Engines. Part IIIa: Server-Side (Store-Process-and-Forward Architecture)', http://ithare.com/64-network-dos-and-donts-for-game-engines-part-iiia-server-side-store-process-and-forward-architecture/

[Wikipedia] https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

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

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



Two Daemons

Most people have come across C++11's forwarding references. Dietmar Kühl explains what && really means.

When playing Nethack [Nethack] using the traditional terminal-based interface you may be confronted by a monster taking the form of a `&` character: a daemon of some sort. Depending on your level of experience, facing one daemon is often OK but facing two tends to get you into way more trouble. Since the release of C++11 we are frequently faced with the two daemons `&&`!

Now, if you think that the two characters next to each other represent the two daemons, you'd be right in the Nethack sense. In C++ any instance of these two characters applied to a type actually represents one of two daemons. When you have got a `T&&` it is either a reference to an object which can be moved from or it is an entity whose type is deduced to match what the entity is initialized with. It uses exactly the same notation for two entirely different things! To relieve you from being haunted by these daemons and rather be prepared to fight them, I'll describe below how these entities differ. In addition, I'll show how you may use these insights to support movable entities with pre-C++11 compilers to some extent.

How to know when to move

First let's make a quick detour. In C++ a lot of temporary objects are floating around. When functions or expressions return their results as values you get a 'temporary'. If the temporary isn't of a simple data type, it may be quite involved and expensive to copy. On the other hand, since they are temporaries, there isn't much use in them as they are about to be destroyed! Instead of copying a temporary to get the value elsewhere and destroying the temporary, it makes much more sense to move the temporary's content if it is expensive.

It would be great if we could indicate whether an object shall be moved or copied! The C++ approach to indicate different processing is to use different types. Since lvalue references are already used to indicate copying and we don't want to move lvalues by accident, we could reasonably use a different type, let's call it `movable_ref<X>`, to indicate that an object can be moved (bear with me – I know about `X&&` and I'll get to this).

If a type is used to indicate that an object can be moved, the compiler can choose an appropriate overload of a function which may move from the object. For example, a class supporting a move construction could be declared like Listing 1.

It is even possible to implement `movable_ref<X>` using a pre-C++11 compiler! All it takes is a class which internally holds a reference to an object indicating that the object won't be used again and to provide suitable access to that reference. For example, a corresponding class template and a suitable factory function could look something like Listing 2 (see the 'Further reading' for a pointer to a complete implementation).

Dietmar Kühl Dietmar is a senior software developer at Bloomberg L.P. working on the data distribution environment used both internally and by clients. In the past, he has done mainly consulting for software projects in the finance area. He is a regular attendee of the ANSI/ISO C++ standards committee and a moderator of the newsgroup comp.lang.c++.moderated.

```
class foo {
  // ...
public:
  foo(foo const& other);
      // well-known copy
  foo(movable_ref<foo> other);
      // move the referenced object
  // ...
};
```

Listing 1

```
template <typename T>
class movable_ref {
  // suitable friend declaration goes here
  T* pointer;
  explicit movable_ref(T& object):
    pointer(&object) {}
public:
  operator T&() const { return *this->pointer; }
};
template <typename T>
movable_ref<T> move(T& object) {
  return movable_ref<T>(object);
}
```

Listing 2

With roughly the declaration in Listing 2, it is possible to use a simple expression like `move(x)` to create an object which indicates that the content of `x` can be transferred! This works like a charm for lvalues. For temporaries things become a little bit harder: a temporary cannot be bound to a non-`const` reference but it would still be desirable to move them. Carrying on just a little bit more with the class template above, it would be possible to create a `move()` member function for types which need to be moved. For example, the `move()` operation of a `std::string` could look like this:

```
movable_ref<std::string> std::string::move() {
  return ::move(*this); }
```

Note that temporaries are non-`const`. They just can't be bound to non-const lvalue reference. It is entirely possible to call a non-`const` member function like `std::string::move()` on them (assuming such a function exists)! So creating the above function would just work and allow temporaries to be moved using a notation like this:

```
std::string("temporary on the").move()
```

Note that so far there is no use of C++11 at all! The class template `movable_ref<X>` can be used with pre-C++11 compilers to move objects. There is the small caveat that it doesn't allow moving objects in return statements but with reasonable compilers there isn't much need for that as they'll use copy-elision with carefully written functions anyway.

Of course, we wouldn't like this approach as it is a bit clumsy. Especially for temporaries it requires too much work! Also, the compiler can actually

if the object wasn't given a name, it can be moved as there is no way for the object to be accessed otherwise

automatically tell whether an object can be moved or needs to be copied in many cases: if the object wasn't given a name, it can be moved as there is no way for the object to be accessed otherwise (well, the object could register itself somewhere but a class supporting moving is well-advised not to do so). Apart from unnamed objects there are also a few other cases when the compiler knows that the object can be safely moved. For example, when a named variable is returned from a function it can be moved:

```
T f() {
  T value(...);
  // ...
  return value;
}
```

Note that the C++ standard considers moving `value` for the above function but will use copying if the return statement is written like this:

```
return (value);
```

In cases where it is known that an object won't be used the compiler could generate the call to `move()` implicitly and yield a `movable_ref<X>`. It turns out this is nearly what happens! Instead of using the type `movable_ref<X>` the compile uses a type which matches the declaration `X&&` where `X` is not a deduced type (I'm using `X` instead of `T` to indicate that `X` is not a deduced type; `T` tends to be used for template parameters and is often deduced), i.e., it uses rvalue references. Before diving into more details of rvalue references note that the notation `movable_ref<X>` can be used interchangeably for `X&&` assuming the following using alias is visible:

```
template <typename T>
using movable_ref =
  std::add_rvalue_reference_t<T>;
```

The interesting aspect of this `using` alias is that it allows code written in terms of `movable_ref<X>` to be used both with a C++03 and a C++11 (or later) compiler. With a few support functions the identical code can be used to move objects! This yields a neat migration path for libraries which need to compile with both new and old compilers. The main flaw of this idea is that it is presented in 2015 and not, at least, 5 years ago. However, better late than never: it may still help some C++ users who haven't migrated off C++03 compilers.

Using `std::add_rvalue_reference_t<X>` in the definition of `movable_ref` instead of `X&&` has the advantage of preventing deduction of `X`. This nearly makes the using alias useful in general in C++11 rather than just for libraries which need to compile with both C++03 and later versions. The main danger is that using `std::movable_ref<X>& x` is legal and looks as if `x` would be a reference to an object which can be moved from but it isn't. Instead the references are collapsed and the notation simply yields a `X&`.

rvalue references

Now let's have a look at rvalue references. When you have got a `X&&` for some non-reference type `X`, e.g. `std::string&&`, you have got a reference which can only be bound to something which can be moved from. Either there are rules allowing the compiler to implicitly consider the object movable or the user has used a cast to have an lvalue look like an rvalue reference, e.g.:

```
std::string     s("lvalue");
std::string&& r = static_cast<std::string&&>(s);
```

Using `std::move(s)` is just another way to write the cast above: `std::move()` is specified to deduce the argument type and do the cast.

A declaration even when using an rvalue reference, e.g., `r` in the above code, introduces an entity with a name. Since it has a name using `r` yields an lvalue! That is, the compiler won't allow moving from the referenced object. The name *rvalue reference* derives from the kind of entities which can be bound to this sort of reference: rvalue references only allow binding objects which are about to go away or which are made to look as if that is the case with a cast. The compiler won't allow binding an lvalue to an rvalue reference without a cast.

The net effect of these rules is that using `X&& x` as part of a function signature indicates that the function was called with an object which can be moved from and it is OK to change the state of `x`. The typical change is to move the content, i.e., to transfer resources from `x` to another object.

According to the core language this transfer can leave the object in some arbitrary state as long as it is safe to call the destructor on the object. However, the standard library mandates that the respective class invariants are retained for the moved-from object when using move construction or move assignment.

Forwarding references

Sadly, when you see a name declared as `T&& t` the object referenced by `T` cannot necessarily be moved! More specifically, if `T` is used to deduce the type the meaning of `T&&` is entirely different! For example, assume you have the class template in Listing 3.

The type for `x` is not deduced. That is, `x` is a reference to a movable object of the template argument `X` specified to instantiate the class template `example`. On the other hand, the type `T` for `t` can be deduced based on the arguments given to `f()`. Listing 4 contains a few examples on how `f()` could be called and what the type `T` becomes.

When replacing `T` in the function template `f()` with `std::string&` i.e., when instantiating `f()` for the type `std::string` the second argument type becomes `std::string& &&`. This odd-looking type doesn't exist as the references are *collapsed*: if there are multiple reference qualifiers on a type, they get collapsed so there is only one reference qualifier:

- `& &&` becomes `&`
- `&& &` becomes `&`
- `&& &&` becomes `&&`

That is, depending on how `f()` is called, `t` may be an lvalue reference or it may be an rvalue reference. Just because the argument is spelled as `T&&` and looks like an rvalue reference, it isn't necessarily one. Unconditionally trying to move from `t` would, thus, likely yield incorrect behavior.

This yields a neat migration path for libraries which need to compile with both new and old compilers

```
template <typename X>
struct example {
  template <typename T>
  static void f(X&& x, T&& t) {
    // ...
  }
};
```

Listing 3

```
std::string       s("mutable");
std::string const c("immutable");
example<int>::f(int(1), s);
  // T == std::string&
example<int>::f(int(2), c);
  // T == std::string const&
example<int>::f(int(2), std::string("tmp"));
  // T == std::string
```

Listing 4

Independent of how `f()` is called, `t` will be considered to be an lvalue. It always has a name and the compiler will not move from a named object implicitly unless the name is about to go away. If you want to potentially move an argument whose type was deduced, you'd use `std::forward<T>(t)`: depending on the explicit template argument type `T` the function argument `t` will be cast to a suitable reference type: if `T` is not an lvalue reference type, the function yields a `T&&`. Otherwise the function yields the type `T`.

Since it is a bit awkward to talk about a `T&&` where the `T` is deduced, there needs to be a name. Scott Meyers refers to these references as *universal references*. The C++ standard defines the term *forwarding reference* for the same entities. I think this term describes what these entities are doing better.

auto and &&

The rules for the type deduced when using `auto` are identical to the rules used with deduced arguments for function templates. Thus, you'll get the following:

- `auto v = expr;` declares `v` to have the value type of the expression `expr` and the result of `expr` will be copied into `v` (of course, the copy can possibly be elided).
- `auto& r = expr;` declares `r` as an lvalue reference to the result of `expr` which assumes that `expr` indeed yields an lvalue.
- `auto const& c = expr;` declares `c` to a const lvalue reference to the result of `expr`. If `expr` doesn't yield an lvalue but a temporary, the life-time of the temporary gets expanded until `c` goes out of scope.
- `auto&& s = expr;` declares `s` to be some reference to the result of `expr`. If `expr` yields a value `s` will be an rvalue reference to the temporary whose life-time gets expanded until `s` goes out of scope. Otherwise `s` will be a reference to the reference yielded by `expr`.

Similar to the use in function templates whether `s` can be moved from depends on how `s` actually happens to be declared despite the use of `&&`. That is, with names declared using `auto&&` you wouldn't use `std::move()`. Instead you would use `std::forward()` with a somewhat ugly-looking type:

```
std::forward<decltype(s)>(s)
```

Of course, spelling out the actual type may be a lot more ugly than `decltype(s)`.

Since we are on the topic of `auto` declarations: do **not** use range-based `for` loops using a variable declared as a plain `auto` unless you know that the value type of the iterator is a type which can be efficiently copied and you don't need to mutate the elements. Otherwise you are much better off to use `auto` with a reference qualifier. As a default it is reasonable to use `auto&&`:

```
for (auto&& v: container) { ... }
```

When the elements are only read you may want to explicitly indicate that the elements are not mutated in which case `auto const&` is the way to go. Similarly, if you know you are going to mutate the elements you may want to explicitly indicate that using `auto&`.

Summary

C++ uses the notation `T&&` for two entirely unrelated things:

- If `T` is deduced `T&&` just means that an object is referenced and the type of `T` indicates whether the referenced object can be moved from.
- If `T` is not deduced `T&&` indicates that the referenced objects can be moved from unconditionally.

To support move semantics using pre-C++11 compilers a class template can be used to indicate that an entity is movable. With a corresponding using alias and a few access functions the same notation can be used for rvalue references in C+11 providing a migration path. ■

Reference and further reading

[Nethack] http://www.nethack.org/

For further reading on this topic see one of these:

Thomas Becker's articles: http://thbecker.net/articles/rvalue_references/section_01.html

Effective Modern C++, Scott Meyers, Item 24

Github: https://github.com/bloomberg/bde/blob/master/groups/bsl/bslmf/bslmf_movableref.h

Scott Meyers. 'Universal references in C++11' *Overload*, 20(111):8:12, October 2012.

Not quite the reaction you were expecting to the latest release?

Clearly stated...

It may not be the software. How clear are the release notes? What about the product manual, online help, training materials, ...?

Changes may meet a business need, but if what worked yesterday doesn't work today, people may resent them. And when today's way involves extra steps, people work around them. After all, their priority is getting the job done.

Result: those shiny new features remain unused, and your application appears not to live up to its promise.

If you would like some help in turning nervous cats into contented ones, get in touch.

T 0115 8492271

E info@clearly-stated.co.uk

W www.clearly-stated.co.uk

We share your belief in professionalism and are members of the Institute of Scientific and Technical Communicators, the UK professional body for technical authors and related professions (visit www.istc.org.uk)

Don't Design for Performance Until It's Too Late

People claim optimisation can cause unreadable code. Andy Balaam argues good performance should be at the core of code design.

There is a piece of ancient wisdom which states:

> Premature optimisation is the root of all evil

This ancient wisdom [Knuth74] is, like all ancient wisdom, correct. However.

It appears to have been reinterpreted as essentially meaning:

> Don't design for performance until it's too late

which is clearly, and very importantly, very wrong.

Performance is a feature

Before I begin I want us all to agree that performance is a feature. [Atwood09]

I work on a real-life 'enterprise' [Vinh07] application. Its features are entirely driven by the need for immediate cash, not by developers following pipe dreams. And yet, for the last 6–12 months the majority of my time has been spent trying to retrofit performance into this application. Believe me, this is not because we have users who are obsessive about wasting valuable seconds – it's because our performance sucks so hard it's deeply embarrassing.

What is your favourite program? How well does it perform? What is your least favourite? Why?

For me, and many other people, their answers to those questions demonstrate the importance of performance. Firefox was launched to improve the performance of Mozilla. People love git because of how fast it is. Lotus Notes is hated so much partly because of its performance. My main complaints about programs I use involve performance (e.g. Thunderbird is too slow for IMAP email).

A fast response to the user is one of those crucial inches [Spolsky07] on the journey to software that makes people happy. Making people happy gives you the kind of scary fanboyism that surrounds git. Wouldn't you like that for your product?

What is optimisation?

When my hero said that premature optimisation was the root of all evil, he was talking in the days when you had to hand-optimise your C in assembly language. Or, more likely in his case, you had to hand-optimise your assembly language into faster assembly language. Optimisation like that very often obfuscates your code.

These days, 99% of the time, your compiler does all of this work for you, so you can have relatively comprehensible code in whatever trendy language you like, and still have the fastest possible local implementation of that in machine code.

Andy Balaam Andy is happy as long as he has a programming language and a problem. He finds over time he has more and more of each.

You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

Meanwhile, Knuth knew that 99% of your code is simply not performance-critical – it only runs a few times, or it's just so much faster than some other bit that it doesn't matter. The lesson we all learn eventually is that the slow bit is never quite what you thought, and you have to measure to find out where to concentrate your effort.

So, if optimisation is obfuscation, and 99% of your code isn't the bit you need to make faster, it becomes clear that premature optimisation is the root of much evil.

But optimisation is not designing for performance.

Design for performance

Fundamentally, to get good performance, you are going to need to measure the time spent in various parts of your code (I suggest Very Sleepy [VerySleepy] if you're on Windows) and make the slow bits faster (or happen less often). However, there are still some principles you can follow that will mean you need to spend less time doing this (which is a pity, because I really love doing it).

If you don't design for performance you are almost certainly going to need to restructure large parts of your program later, which is very difficult and time-consuming.

There are two aspects to designing for performance: writing good local code, and creating good global structure.

Write good local code

Before you write an algorithm, think for a few minutes about how to make it work efficiently. e.g. if you're writing C++, consider whether a `deque` or a `list` would be better than a `vector` for how you're going to use it.

Think about what is idiomatic for your language and why. Think about what the computer really has to do to produce the results you are asking for. Are there going to be a lot of objects about? Maybe you can avoid copying them too many times. Are you creating and deleting a lot of objects? Can you reuse some instead? (Exercise caution with that one, though – if you start obfuscating you come into conflict with the ancient wisdom.)

Often, if you think through what you are doing, and the most efficient way to do it, you will end up with a faster and more memory-efficient algorithm, that expresses your intention better than if you'd written the first thing that came into your head. There is no downside to that.

Try to minimise the number of times you need to ask the operating system for a chunk of memory: this is surprisingly slow. E.g. in C++, prefer creating by-value data members instead of pointers to objects allocated with their own call to new.

By the way, don't worry if this sounds intimidating. The way to learn this stuff is to measure what you have done and then work out why it is slow. Next time you'll jump straight to the fast solution without the detour to the slow one.

Of course, none of this will matter if you don't have good global structure.

if different parts use different string classes you are going to spend most of your time copying from one to the other

Always consider using vector in C++

C++'s standard containers are specified to fulfil certain 'performance' criteria, and also to give certain guarantees (e.g. about how long iterators stay valid) that are influenced by the algorithms that are expected to be used. The 'performance' criteria specify algorithm complexity, which means they are important when the number of elements in a container is very large, but real-world performance when numbers are smaller may be very different. Bjarne Stroustrup (among others) has observed [Stroustrup12] that even with thousands of elements, vectors can comprehensively outperform the other containers, even for operations that sound like the kinds of operations more suited to lists or maps.

Stroustrup provides a rule of thumb he calls 'Use compact data', which reflects the fact that often the most performance-critical aspect of code is the locality of the data being operated on. The enormous differences in memory access speed between different levels of cache and the system memory mean that the effect of keeping data close together (as a vector does) can swamp any effects of algorithmic complexity. [Wicht12]

When choosing a C++ container, normally the most important consideration will be simplicity and comprehensibility of the code, but when performance is the primary consideration, you should measure, and you may be surprised how often vector is the right choice. Many argue vector should be your default choice.

Create good global structure

The hardest and most important work you need to do to have good performance is to have good structure in the ways the different parts of your program interact.

This means thinking about how classes and components communicate with and control each other.

It may be helpful to use a streaming style of communication – can you send little chunks of information to be processed one by one instead of a huge great blob?

Try to make sure your components to use a common infrastructure: if different parts use different string classes you are going to spend most of your time copying from one to the other.

The hardest and deepest mystery in getting good performance (and in programming generally) is choosing the right fundamental data structures. I'll never forget the lesson I learnt when a friend of mine had a conversation with me about a toy project I was doing (that was particularly focussed on trying to be fast) and then went away and produced code that was orders of magnitude faster, simply because he had chosen the right data structure. The lesson I learnt was that I am not as good as I think I am.

To be honest this section is a little shorter than I'd like because I know I don't have a lot of answers about how to do this well. I do know, though, that if you don't think about it now you will have the pain of restructuring your program later, when it's full of bug fixes that are going to get re-broken by the restructuring.

Of course, if you do think about it now you're still pretty likely to need to change it later…

Ancient wisdom

Ancient wisdom is usually right, but misinterpreting it and using it as a license to write bad code is a bad idea. Carry on. ■

References

[Atwood09] Atwood, Jeff 'A Few Speed Improvements' http://blog.stackoverflow.com/2009/08/a-few-speed-improvements/

[Knuth74] Knuth, Donald '1974 Turing Award Lecture' from *Communications of the ACM* 17 (12), (December 1974), p 671

[Spolsky07] Spolsky, Joel 'A game of inches' http://www.joelonsoftware.com/items/2007/06/07.html

[Stroustrup12] Stroustrup, Bjarne 'Why you should avoid Linked Lists' from GoingNative 2012: https://www.youtube.com/watch?v=YQs6IC-vgmo

[VerySleepy] 'Very Sleepy' http://www.codersnotes.com/sleepy

[Vinh07] Vinh, Khoi 'If It Looks Like a Cow, Swims Like a Dolphin and Quacks Like a Duck, It Must Be Enterprise Software' http://www.subtraction.com/2007/10/19/if-it-looks-/

[Wicht12] Wicht, Baptiste 'C++ benchmark – std::vector VS std::list' https://dzone.com/articles/c-benchmark-%E2%80%93-stdvector-vs

We're always looking for new articles and new authors. If you are interested in writing for us but are looking for inspiration, take a look at our current wish-list of topics...

Overload wish-list

* crypo-currencies
* C++17
* property-based testing
* how to structure large python projects
* Qt
* Boost
* SQL
* NoSQL
* programming paradigms in general - especially functional
* Gcc architecture or internals.
* Anything GNU/Linux tool chain related
* History of computing
* Maintaining old systems
* Computational geometry using rounded arithmetic
* Puzzles

then read the 'Guide for Contributors' on the website (http://accu.org/index.php/journals/1414) and discuss your idea with the *Overload* editor.

Template Programming Compile Time String Functions

Practising old exercises in new ways can keep you sharp. Nick Weatherhead demonstrates some well-known code katas using C++ compile time tricks.

Many of us will recall some of the first programming exercises we had to solve. Possibly they involved converting strings into different representations and detecting some properties. When learning a new programming language or technique, it's not unusual to revisit and apply some of these. I will look at three examples using C++ templates for compile time constant expressions; an integer to string conversion, then the reverse by generating an integer from a Roman numeral, and finally detecting whether a string is a palindrome.

As a string is a sequence of adjacent characters it is common to physically represent it as an array. However, we soon discover that templates do not permit the use of string literals as parameters. There are a couple of workarounds – they can be defined with external linkage indicating that they are a single entity across translation units or, alternatively, they can be defined as a class member and passed in as a type. However, their use in compile time expressions is limited because pointer arithmetic isn't allowed, and further, whilst integral types can be evaluated at compile time with a propagated constant expression, there isn't a similar mechanism that recursively expresses the sequence of elements required in an array initialisation. Let's consider some alternatives.

```
#include <iostream>
using namespace std;
template< size_t I > struct itoa {
  friend ostream& operator<<(
    ostream& os, const itoa& ) {
    return os << itoa< I / 10 >( )
              << itoa< I % 10 >( );
    }
};
#define DIGITS X(0) X(1) X(2) X(3) X(4)        \
               X(5) X(6) X(7) X(8) X(9)
#define X( I )                                 \
template< > struct itoa< I > {                 \
  operator char( ) const { return 0x3##I; }    \
};                                             \
DIGITS
#undef X
int main( ) {
  cout << "\n3210 = " << itoa< 3210 >( )
       << "\n26   = " << itoa< 26   >( )
       << "\n9    = " << itoa< 9    >( );
}
```

Listing 1

Nick Weatherhead Nick's first encounter with programming was copying lines of code from magazines into the now venerable family BBC B. His teacher persuaded him to take computer science during his first term of A-Levels. This led to many hours of puzzle solving and programming, a relevant degree and finally gainful employment within London's financial sector. You can contact Nick at weatherhead.nick@gmail.com

A basic integer to ASCII conversion can be defined as in Listing 1. The arithmetic can be determined at compile time, but a sequence of calls is required to place each individual character into the output stream. It also shows how 'X' Macros are used in generating repeated code when multiple template specialisations are required. Instead of individually outputting each character, each class could be composed of a single character (1 byte). Typically, a compiler will align characters on a one byte boundary such that, when these are constructed together, they form a contiguous block of characters that can be cast as a string. Note that this technique isn't guaranteed to work; whilst C and C++ compilers retain the order in which members are declared in a structure they can add padding between them. Adapting the above gives Listing 2.

For the specialisation of one digit there is an `out` function and in the general case to concatenate digits there is a nested class named `out` too; both can be called with the same `out( )` in order to return a null terminated string. The nested class can work on a completed definition of the outer template object, thus becoming a functor.

The constructors are specified with `constexpr`. This indicates that they can be evaluated and their resulting objects initialised at compile time. If

```
#include <iostream>
using namespace std;
template< size_t I > struct itos
: itos< I / 10 > {
  const char _;
  constexpr itos( ) : _( 0x30 + I % 10 ) {
    static_assert(
        sizeof( itos )
      == sizeof( itos< I / 10 > ) + 1
    ,
      "unwanted padding"
    );
  }
  struct out {
    const itos< I > _; const char nil;
    constexpr out( ) : nil( '\0' ) {
      static_assert(
          sizeof( out )
        == sizeof( _ ) + 1
      ,
        "unwanted padding"
      );
    }
    constexpr operator const char* const( )
    const {
      return _;
    }
  };
};
```

Listing 2

Expressions are not evaluated until the last character is read and the stack unwinds; thus the numerals are evaluated as if read in reverse

```
#define DIGITS X(0) X(1) X(2) X(3) X(4)         \
               X(5) X(6) X(7) X(8) X(9)
#define X( I )                                  \
template< > struct itos< I > {                  \
  const char _;                                 \
  constexpr itos( ) : _( 0x3##I ) { }           \
  constexpr static const char* const out( ) {   \
    return #I;                                  \
  }                                             \
  constexpr operator const char* const( )       \
  const {                                       \
    return &_;                                  \
  }                                             \
};                                              \

DIGITS

#undef X

int main( ) {
    cout << "\n3210 = " << itos< 3210 >::out( )
         << "\n26   = " << itos< 26   >::out( )
         << "\n9    = " << itos< 9    >::out( );
}
```

Listing 2 (cont'd)

`constexpr` isn't present calls to each constructor will be generated instead. Whether the constant expression branch is taken can be determined using the `noexcept` operator, for example:

```
noexcept( itos< 10 >::out( ) )
```

will evaluate to `true` in the example in Listing 2, but if `constexpr` is removed from either of the constructors it will return `false`. Static assertions are included in the constructors to verify that the characters are contiguous.

The code in Listing 3 uses variadic templates to interpret Roman numerals with the combinations of **I**, **V** and **X**, from **I** up to and including **XXXIX** i.e. from 1 to 39. However, it does so without validating the numerals. Roman numerals are written from left to right in descending denominations with the exception of subtractive cases e.g. **IV** and **IX**; they should also be represented using the fewest characters necessary.

In Listing 4, an underlying structure `n_` is defined to cater for a running total of the decimal value `d`, and counts for consecutive `I`s, `V`s and `X`s (`i`, `x`, `v`) which are then encapsulated in a class `n`. As each class is called a character is pushed onto the stack and these properties are transitively passed from one nested template to the next in the chain. Expressions are not evaluated until the last character is read and the stack unwinds; thus the numerals are evaluated as if read in reverse. Specialisations for `n` are defined for `I`, `V` and `X`; each adds its respective denomination of 1 (in the subtractive case of an `I` appearing before a `V` or an `X` this is negated), 5 or 10 to the running decimal and increases its character count.

```
#include <iostream>
using namespace std;

template< char N, char... NS > struct ns {
  static const int d =
     ns< N     >::d
  +  ns< NS... >::d;
};

template< > struct ns< 'I' > {
  static const int d = 1;
};

template< > struct ns< 'V' > {
  static const int d = 5;
};

template< > struct ns< 'X' > {
  static const int d = 10;
};

template< > struct ns< 'I', 'V' > {
  static const int d = 4;
};

template< > struct ns< 'I', 'X' > {
  static const int d = 9;
};

int main( ) {
  static const int
    v       = ns<'V'>::d
  , xix     = ns<'X','I','X'>::d
  , xxiv    = ns<'X','X','I','V'>::d
  , xxxviii = ns<'X','X','X','V','I','I','I' >::d;
  cout << "\nV       = " << v
       << "\nXIX     = " << xix
       << "\nXXIV    = " << xxiv
       << "\nXXXVIII = " << xxxviii;
}
```

Listing 3

Further consideration is given to glyphs that are greater in value to the one preceding them. Where this occurs any active counts for higher denomination characters are floored to one; this allows the subtractive cases to be dealt with and the likes of `VX` and `IIX` to be eliminated. Counters for denominations smaller than the one being examined are already accounted for so are zeroed out. In order to ensure a numeral is represented using the fewest characters any occurrence of more than three consecutive `I`s or `X`s or more than one `V` are identified. To prevent a combination a class definition is supplied which matches the count of the characters but has no body – this causes compilation to fail if any result.

The strategy to detect a palindrome is to step from the outermost to the innermost characters, comparing the beginning and end letters to see if they match

```
#include <iostream>
using namespace std;

template< int D, int I, int V, int X >
struct n_ {
  static const int d = D, i = I, v = V, x = X;
};

template< int D,        int V, int X >
struct n_<    D,     4,     V,     X >;

template< int D, int I,        int X >
struct n_<    D,     I,     2,     X >;

template< int D,               int X >
struct n_<    D,     2,     1,     X >;

template< int D, int I               >
struct n_<    D,     I,     1,     1 >;

template< int D, int I, int V        >
struct n_<    D,     I,     V,     4 >;

template< int D,        int V        >
struct n_<    D,     2,     V,     1 >;

template< char C, typename N = n_< 0, 0, 0, 0 > >
struct n { };

template< typename N > struct n< 'I', N >
: n_<
    N::d + ( N::v || N::x ? -1 : 1 )
  , N::i + 1
  , N::v ? 1 : 0
  , N::x ? 1 : 0
  >

{ };

template< typename N > struct n< 'V', N >
: n_<
    N::d + 5
  , 0
  , N::v + 1
  , N::x ? 1 : 0
  >

{ };
```

Listing 4

To aid legibility the nested templates are wrapped by a variadic template class (see Listing 5).

```
template< typename N > struct n< 'X', N >
: n_<
    N::d + 10
  , 0
  , 0
  , N::x + 1
  >

{ };

template< char N, char... NS > struct ns
: n< N, ns< NS... > >

{ };

template< char N > struct ns< N >
: n< N >

{ };

int main( ) {
//  static const int
//    iiv  = ns<'I','I','V'>::d
//  , vv   = ns<'V','V'>::d
//  , vxx  = ns<'V','X','X'>::d
//  , xxxx = ns<'X','X','X','X'>::d;

  static const int
    v       = ns<'V'>::d
  , xix     = ns<'X','I','X'>::d
  , xxiv    = ns<'X','X','I','V'>::d
  , xxxviii = ns<'X','X','X','V','I','I','I'
                >::d;
  cout << "\nV       = " << v
       << "\nXIX     = " << xix
       << "\nXXIV    = " << xxiv
       << "\nXXXVIII = " << xxxviii;
}
```

Listing 4 (cont'd)

In Listing 5, the outer class `l` defines a concatenation of characters and the nested template class `p` test to see if these represent a palindrome (reading the same backwards as forwards). The strategy to detect a palindrome is to step from the outermost to the innermost characters, comparing the beginning and end letters to see if they match until either they don't, in which case it isn't a palindrome, or the middle is met, in which case it is. For a single character, the specialisation `l` with a `nil` continuation, this is trivially true; however, for a concatenated list indexing and iteration of elements is required. This is achieved by indexing each element, as each is added to the list, with a propagated count `i`. When calling `p` the outer class will reference the last character `C` at the index `i` (being the length of the list minus one). Template `p` takes a parameter `I` which is the index of

... the interpretation and pattern matching capabilities of templates and their ability to prevent as well as action state transitions

```
#include <iostream>
using namespace std;

struct nil;

template< char C, typename L = nil > struct l {
  static const size_t i = L::i + 1;
  template < size_t I = 0 > struct p {
    static const char c =
      I < i ? L::template p< I >::c : C;
    static const bool is =
    (
      I >= i
    ||
        c == C
      &&
        L::template p< I + 1 >::is
    );
  };
};
template< char C > struct l< C, nil > {
  static const size_t i = 0;
  template< size_t I = 0 > struct p {
    static const char c     =    C;
    static const bool is    =    true;
  };
};
template< char C, char... CS > struct ls
:   l< C, ls< CS... > >
{ };
template< char C > struct ls< C >
:   l< C >
{ };

int main( ) {
  static const bool
    arora  = ls<'a','r','o','r','a'>
                ::p< >::is
  , hannah = ls<'h','a','n','n','a','h'>
                ::p< >::is
  , ania   = ls<'a','n','i','a'>
                ::p< >::is;
  cout << "\narora   = "
       << ( arora  ? "y" : "n" )
       << "\nhannah  = "
       << ( hannah ? "y" : "n" )
       << "\nania    = "
       << ( ania   ? "y" : "n" );
}
```

Listing 5

the corresponding character to make a comparison with; in the outermost case this is the first at index zero. The recursive function `is` checks to see if the index requested indicates that the middle has been met or, in the case of even length strings, crossed. If it hasn't then the characters are compared with one another and, if they match, the link `L` is followed to the previous character and `p` is called again with the next index. In this way the elements are being evaluated in reverse order to which the chain is defined. When finding a corresponding character `c` the link to previous elements is recursively followed until the index `I` is found.

In summary, 'integer to string' looked at alternative ways in which a string can be emitted with templates. It also demonstrated how nested classes can be used to define functions on the template definition of the outer class, the use of `X` macros to generate code for repeating specialisations, and introduced the use of the C++11 `constexpr` to evaluate constructors at compile time. 'Roman numeral' highlighted the interpretation and pattern matching capabilities of templates and their ability to prevent as well as action state transitions. Finally 'palindrome' built on the use of nested class functors to iterate over a sequence of characters. ■

Acknowledgements

I'd like to thank the *Overload* review team for their advice, particularly for suggesting that I explore the use of variadic templates, commenting that static assertions can be used to verify whether character alignment within a structure is contiguous, and also for cautioning against the use of `reinterpret_cast`.

Further reading

constexpr specifier (since C++11), May 2015.
http://en.cppreference.com/w/cpp/language/constexpr

C++11FAQ : constexpr – generalized and guaranteed constant expressions, September 2014
http://www.stroustrup.com/C++11FAQ.html

C++ Template: The Complete Guide, David Vandevoorde and Nicolai M. Josuttis, Addison Wesley, 2002, pp.40-41, 209-210.

Data structure alignment, June 2015,
https://en.wikipedia.org/wiki/Data_structure_alignment

JOIN THE ACCU!

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

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

With full ACCU membership you get:

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

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

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

How to join

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

Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Numbers in JavaScript

JavaScript floating point numbers can confuse C++ programmers. Anthony Williams draws our attention to some surprising behaviour.

I've been playing around with JavaScript (strictly, ECMAScript) in my spare time recently, and one thing that I've noticed is that numbers are handled slightly strangely. I'm sure that many experienced JavaScript programmers will just nod sagely and say "Everyone knows that", but I've been using JavaScript for a while and not encountered this strangeness before as I've not done extensive numerical processing, so I figured it was worth writing down.

Numbers are floating point

For the most part, JavaScript numbers are floating point numbers. In particular, they are standard IEEE 754 64-bit double-precision numbers. Even though the IEEE spec allows for multiple NaN (Not-a-number) values, JavaScript has exactly one NaN value, which can be referenced in code as `NaN`.

This has immediate consequences: there are upper and lower limits to the stored value, and numbers can only have a certain precision.

For example, 10000000000000001 cannot be represented in JavaScript. It is the same value as 10000000000000000.

```
var x=10000000000000000;
if(x==(x+1))
  alert("Oops");
```

This itself isn't particularly strange: one of the first things you learn about JavaScript is that it has floating-point numbers. However, it's something that you need to bear in mind when trying to do any calculations involving very big numbers (larger than 9007199254740992 in magnitude) or where more than 53 bits of precision is needed (since IEEE 754 numbers have binary exponents and mantissas).

You might think that you don't need the precision, but you quickly hit problems when using decimal fractions:

```
var x=0.2*0.3-0.01;
if(x!=0.05)
  alert("Oops");
```

The rounding errors in the representations of the decimal fractions here mean that the value of `x` in this example is 0.049999999999999996, not 0.05 as you would hope.

Again, this isn't particularly strange, it's just an inherent property of the numbers being represented as floating point. However, what I found strange is that sometimes the numbers aren't treated as floating point.

Numbers aren't always floating point

Yes, that's right: JavaScript numbers are sometimes not floating point numbers. Sometimes they are **32-bit signed integers**, and very occasionally 32-bit unsigned integers.

The first place this happens is with the bitwise operators (`&`, `|`, `^`): if you use one of these then both operands are first converted to a 32-bit signed integer. This can have surprising consequences.

Look at the following snippet of code:

```
var x=0x100000000; // 2^32
console.log(x);
console.log(x|0);
```

What do you expect it to do? Surely `x|0` is `x`? You might be excused for thinking so, but no. Now, `x` is too large for a 32-bit integer, so `x|0` forces it to be taken modulo 2^{32} before converting to a signed integer. The low 32-bits are all zero, so now `x|0` is just 0.

OK, what about this case:

```
var x=0x80000000; // 2^31
console.log(x);
console.log(x|0);
```

What do you expect now? We're under 2^{32}, so there's no dropping of higher order bits, so surely `x|0` is `x` now? Again, no. `x|0` in this case is `-x`, because `x` is first converted to a signed 32-bit integer with 2s complement representation, which means the most-significant bit is the sign bit, so the number is negative.

I have to confess, that even with the truncation to 32-bits, the use of signed integers for bitwise operations just seems odd. Doing bitwise operations on a signed number is a very unusual case, and is just asking for trouble, especially when the result is just a 'number', so you can't rely on doing further operations and having them give you the result you would expect on a 32-bit integer value.

For example, you might want to mask off some bits from a value. With normal 2s complement integers, `x-(x&mask)` is the same as `x&~mask`: in both cases, you're left with the bits set in `x` that were not set in mask. With JavaScript, this doesn't work if `x` has bit 31 set.

```
var x=0xabcdef12;
var mask=0xff;
console.log(x-(x&mask));
console.log(x&~mask);
```

If you truncate back to 32-bits with `x|0` then the values are indeed the same, but it's easy to forget.

Shifting bits

In languages such as C and C++, `x<<y` is exactly the same as `x*(1<<y)` if `x` is an integer. Not so in JavaScript. If you do a bitshift operation (`<<`, `>>`, or `>>>`) then JavaScript again converts your value to a signed integer before and after the operation. This can have surprising results.

Anthony Williams Anthony is the author of *C++ Concurrency in Action*. As well as working on multi-threading libraries, he develops custom software for clients, and does training and consultancy. Despite frequent forays into other languages, he keeps returning to C++. He is a keen practitioner of TDD, and likes solving tricky problems. Contact him at anthony@justsoftwaresolutions.co.uk

JavaScript numbers are double-precision floating point, so need to be treated the same as you would floating point numbers in any other language

```
var x=0xaa;
console.log(x);
console.log(x<<24);
console.log(x*(1<<24));
```

`x<<24` converts `x` to a signed 32-bit integer, bit-shifts the value as a signed 32-bit integer, and then converts that result back to a `Number`. In this case, `x<<24` has the bit pattern 0xaa000000, which has the highest bit set when treated as 32-bit, so is now a negative number with value -1442840576. On the other hand, `1<<24` does not have the high bit set, so is still positive, so `x*(1<<24)` is a positive number, with the same value as 0xaa000000.

Of course, if the result of shifting would have more than 32 bits then the top bits are lost: `0xaa<<25` would be truncated to 0x54000000, so has the value 1409286144, rather than the 5704253440 that you get from `0xaa*(1<<25)`.

Going right

For right-shifts, there are two operators: `>>` and `>>>`. Why two? Because the operands are converted to *signed* numbers, and the two operators have different semantics for negative operands.

What is 0x80000000 shifted right one bit? That depends. As an unsigned number, right shift is just a divide-by-two operation, so the answer is 0x40000000, and that's what you get with the `>>>` operator. The `>>>` operator shifts in zeroes. On the other hand, if you think of this as a negative number (since it has bit 31 set), then you might want the answer to stay negative. This is what the `>>` operator does: it shifts in a 1 into the new bit 31, so negative numbers remain negative.

As ever, this can have odd consequences if the initial number is larger than 32 bits.

```
var x=0x280000000;
console.log(x);
console.log(x>>1);
console.log(x>>>1);
```

0x280000000 is a large positive number, but it's greater than 32-bits long, so is first truncated to 32-bits, and *converted to a signed number*. `0x280000000>>1` is thus not 0x140000000 as you might naively expect, but -1073741824, since the high bits are dropped, giving 0x80000000, which is a negative number, and `>>` preserves the sign bit, so we have 0xc0000000, which is -1073741824.

Using `>>>` just does the truncation, so it essentially treats the operand as an *unsigned* 32-bit number. `0x280000000>>>1` is thus 0x40000000.

If right shifts are so odd, why not just use division?

Divide and conquer?

If you need to preserve all the bits, then you might think that doing a division instead of a shift is the answer: after all, right shifting is simply dividing by 2^n. The problem here is that JavaScript doesn't have integer division. 3/2 is 1.5, not 1. You're therefore looking at two floating-point operations instead of one integer operation, as you have to discard the fractional part either by removing the remainder beforehand, or by truncating it afterwards.

```
var x=3;
console.log(x);
console.log(x/2);
console.log((x-(x%2))/2);
console.log(Math.floor(x/2));
```

Summary

For the most part, JavaScript numbers are double-precision floating point, so need to be treated the same as you would floating point numbers in any other language.

However, JavaScript also provides bitwise and shift operations, which first convert the operands to 32-bit signed 2s-complement values. This can have surprising consequences when either the input or result has a magnitude of more than 2^{31}.

This strikes me as a really strange choice for the language designers to make: doing bitwise operations on signed values is a really niche feature, whereas many people will want to do bitwise operations on unsigned values.

As browser JavaScript processors get faster, and with the rise of things like Node.js for running JavaScript outside a browser, JavaScript is getting used for far more than just simple web-page effects. If you're planning on using it for anything involving numerical work or bitwise operations, then you need to be aware of this behaviour. ■