the magazine of the accu

www.accu.org

{cvu}

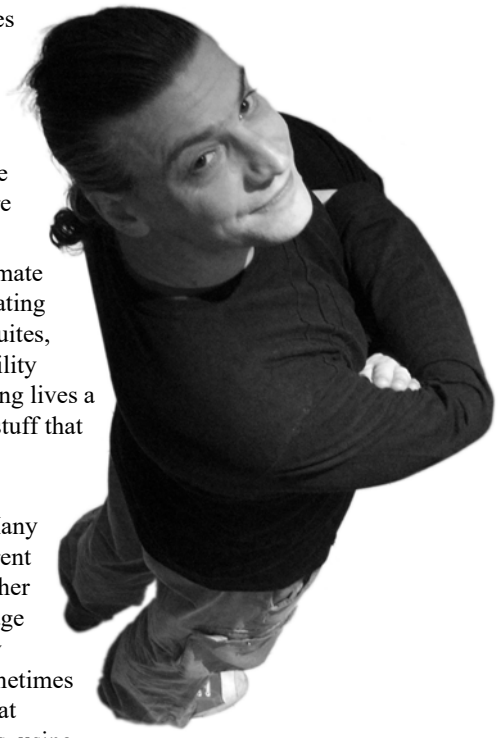
Volume 28 • Issue 4 • September 2016 • £3

Features

Regulars

{cvu}
Volume 28 Issue 4
September 2016
ISSN 1354-3164
www.accu.org

The integrated developer



The activity we call 'software development' is a many-faceted thing, and we have many titles for ourselves in its pursuit. Whether you're a Software Engineer, Architect, Programmer or (just) Coder, whether you think of it as science, engineering or craft, there's a great deal we do that's not actually writing code. And some of the code we do write isn't necessarily the code we're *paid* to write.

There are scripts to be written and tested to automate the mundane tasks like building the code, generating documentation, creating test data, running test suites, parsing log files and the like. There are small utility programs and IDE plug-ins that make our working lives a little easier. On top of this, there's all the other stuff that isn't programming at all.

We need to understand the intricacies of source control systems, some ancient, some modern. Many of us will need to get to grips with several different ones, and try not to get our shoe-laces tied together too much, so to speak. A vast array of data storage and retrieval facilities, from file systems to fully Enterprise class databases is at our disposal, sometimes all at the same time. We may need to be skilled at writing documentation in any number of formats, using any number of different authoring tools. We need to be able to operate the very latest tools alongside the most archaic, often at short notice. We are Continuously Integrating, Unit Testing, Bug Tracking, Web Hosting, Machine Learning, Mobile, Responsive, Agile and On-Line. Oh, I almost forgot Secure...

We may even have to be skilled diplomats in order to prevent our customers, managers, or fellow programmers from rioting or revolting.

Even for actually writing code, we may have to be familiar with a colony of data formats, an army of programming languages and, um, a *murder* of different versions of platforms, compilers, operating systems, libraries, and so on. Fun, isn't it?

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

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

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 28.5 1st October 2016
C Vu 28.6: 1st December 2016

Overload 136: 1st November 2016
Overload 137: 1st January 2017

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

Home-Grown Tools

Chris Oldwood turns to custom tools when off the shelf won't do.

Up until now this column has largely talked about tooling from the perspective of using 3rd party tools to 'get stuff done' but there are occasions when the one perfect tool we really need doesn't exist. It's often possible that we can cobble something together using the standard tools like cat, sed, grep, awk, etc. and solve our problem with a little composition, but if it's not that sort of problem then perhaps it's time to write our own.

Custom tooling spectrum

In the third instalment of this column [1], I covered what is probably the most lightweight approach to tooling, which is to wrap one or more other tools inside a simple script. These are very easy to write and are often an enabler for automating some kind of task. The investment is small and it's an easy win to reduce some friction in the development process and remove another source of human error from the loop. Despite the plethora of extensions and plug-ins available, there still seems to be an endless supply of small analysis and integration jobs that need doing to create a free-flowing development process.

At the opposite end of the spectrum are the kinds of constraints that lead to a serious investment in your own tooling. When you look at companies like Netflix and Google and see the tools that they put back into the development community you realise they are dealing with problems at a scale which many of us will never have to deal with. The rise of open source software has meant that companies who have invested in custom tooling are perhaps much more prominent than they once were as they often choose to release the fruits of their labour to the wider community. Clearly companies have had to do this in the past, but historically they may have seen this diversion as a technical advantage to be leveraged rather than a problem to be shared.

One of the earliest examples I know of where a company decided to rely on custom tooling was the Microsoft Excel team, which built its own C compiler [2]. Instead of having to use separate compilers for each platform they hedged their bets and created their own which would compile C code to a platform-independent bytecode called 'p-code'. Although the anticipated plethora of platforms never materialised, keeping the compiler in-house still brought them a number of benefits, such as stability and consistency, which in turn allowed them to deliver a better product faster.

Striking a balance

This latter example, along with Google creating its own version control system or Dropbox deciding to build its own cloud, is well outside the kinds of bounds I've ever experienced. I'm quite certain that if I told my clients that I needed to spend 6 months writing a continuous delivery system or web server framework, I'd be given my marching orders right away. Like all engineering trade-offs there is a balance between trying to bend and twist a general purpose tool into shape versus building exactly what you (think you) need. For the kinds of programming I do, the mainstream general-purpose programming languages available easily provide enough features, especially when coupled with the dizzying array of libraries that you now have easy access to.

Back at the very start of my professional career I joined a small software house which was slowly feeling the pinch from ever tighter margins through competition. I suspect they didn't have the kind of budget you needed for an enterprise scale internet mail gateway but they managed to adapt Phil Karn's DOS-based KA9Q suite [3] to send and receive mail to and from their PMail / Netware based setup. Given the free availability of the original source they naturally published both the binaries and source code for their customised version on Cix, Compuserve and Demon.

This was my first real foray into the world of writing and sharing tools. Whilst I had been a consumer of plenty of free software at University, it felt good to finally be on the other end – giving rather than receiving. Luckily the company was already licensed to use the NetWare SDK for its own licensing library and so I had the opportunity to fill the gaps left by Novell's own tools and produce some little utilities on the side. These were mostly to help with managing the printers but I also wrote a little tool to map out the network and remotely query the machines to help diagnose network problems. These, along with my DOS-based text-mode graphics library that emulated the Netware tools look-and-feel, were all published in source and binary form on Cix too.

Whilst my day job was supposedly writing desktop graphics software, I presume my employer (I was permanent back then) was happy to indulge these minor distractions because I was learning relevant skills and still contributing something of business value too. Knowing the people there, I'm sure the second-order effects, such as my resulting extra happiness at being given at little latitude, was factored in too. Building tools for pedagogical purposes has remained a theme ever since; however, this has largely been restricted to my own personal time since I switched from permanent to freelance status for the reasons cited earlier.

Sadly my desire to build tools to solve reoccurring problems was not always met with such gratitude. Whilst contracting at a large financial institute, I found myself being berated for building a simple tool that would allow me to extract subsets of data from our huge trade data files. Despite there only being some arcane methods when I joined the team, and it being a fundamental part of our testing strategy, I was somewhat surprised when my efforts were questioned rather than applauded. Perhaps if I had spent weeks writing it when something more pressing was needed I could understand my poor judgment, but it was developed piecemeal a few hours at a time. Fortunately its true value was realised soon after when the BAU and Analysis teams both started using it to extract data too. However, although I felt partly vindicated, it still seemed like a shallow victory because it didn't spark the interest in building and sharing tooling that I hoped it would.

From test to support

One of the things I've discovered from writing tools is that your audience often extends far wider than perhaps you envisaged. I mostly write them for my benefit – to make my own life easier – but naturally if it solves a problem for me then it probably will for someone else too. The area I find this has happened most in the past is with test tools that grow into administration or support tools.

CHRIS OLDWOOD

Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race andcan be easily distracted via gort@cix.co.uk or @chrisoldwood

With any complex system you often need to be able to get 'inside it' to be able to diagnose a problem that is not apparent from the usual logging and monitoring tools. This might mean that you need to replay a specific request, perhaps using a custom tool that you can drive via a debugger in the comfort of your own chair. Whilst remote debugging of a live system is possible, it should obviously be reserved for extreme cases due to the disruption it causes.

This is exactly what happened to the tool I mentioned earlier that I was chastised for spending time on. It was written initially to solve my own needs to generate test data sets and to safely and quickly get hold of production data so that I could replay requests to debug the large, monolithic service. I added a few extra filters to allow me to create data sets such as one that could replay the exact sequence of requests that had caused a grid computing engine to crash. When the BAU team 'discovered' it, they used it to answer questions about specific customers and the Analysis team used it to address regressions and regulatory questions.

Sometimes it's obvious how something you write can be used in multiple contexts and there is a clear path around adding features that bias it for one use or another. For instance, in production scenarios I like to ensure that any tool behaves in a strictly read-only manner, which I naturally make the default behaviour. On occasion, when re-purposing someone else's creation or a tool that's already being automated in production, I might have to add a command line switch (e.g. `--ReadOnly`) to allow it to be run in a non-destructive 'support mode'. This is one of the reasons why I prefer for all integration test environments to be locked down by default as tight as production because it allows you to drive out the security and support requirements by dog-fooding in your non-production environments.

one only has to look at the continued release of new text editors, new programming languages and new build systems to see that we are far from done yet

Even tools that you never thought had any use outside the development team have a funny way of showing up in support roles. A mock front-end I once wrote to allow the calculation engine developers to test and debug without waiting an eternity for the real front-end to load got used by the test users to work-around problems with the actual UI. Despite its Matrix-esque visuals, one savvy user was so experienced they could see past the raw object identifiers and could have a good idea of what the results were based on the rough shape and patterns of the calculation's solution.

System testing

By now it should be pretty apparent that the lion's share of the tools I write are for automating development tasks or for some form of system testing. Whilst I rely heavily on automated unit, integration and acceptance tests for the majority of test coverage, I still prefer to do some form of system testing when I make changes that sit outside the norm. For example, changes to 3rd party dependencies such as library upgrades or major tooling which might throw up something peculiar at deploy or runtime are good candidates. Also, any change where I know the automated integration tests might be weak will cause me to give it the once over locally first before committing and publishing, to avoid unnecessarily breaking the build or deployment.

Replacing the real-front end with a lightweight alternative for testing, profiling and debugging is a common theme too. The calculation engines I've been involved in often have an extensive front-end, perhaps written Visual Basic or another GUI tool that makes getting to the library entry points I'm concerned with time consuming. In one case the front-end took 4 minutes before it even hit our initial entry point and so building alternative scaffolding is a big time-saver. Often the library will just be fronted by a command line interface with various switches for common options. The command line approach also makes for a great host to use when profiling the library with specific data sets. Naturally this again leads towards a path of automation.

Occasionally I've built a desktop GUI based front-end too if I think it will help other developers and testers. When the product has historically been manually tested through a fat client, it can provide a half-way house that still helps on the exploratory side without the rawness of a hundred command line arguments and complex configuration files. When the interop layer involves a technology like COM, then a proper UI can help unearth some quirks that only occur in these scenarios.

More recently I've been working on web-based APIs, which tend to be quite dry and boring affairs from a showcasing perspective. Like libraries, they are quite tricky things to develop without having some bigger picture to drive them. Hence I favour creating a lightweight UI that can be used to invoke the API in the same manner as the anticipated clients. Not only does this help to elevate the discussion around the design of the API to the client's perspective, it also provides a useful exploratory testing tool to complement the suite of automated tests.

Architecture benefits

Trying to isolate portions of a system, whether at the class or function level for unit testing, or subsystem level to support finer-grained servicing, generally has beneficial effects on the overall architecture. The need to be able to interact with a system through interfaces other than those which an end user or downstream system might use, for the purposes of testing or support, forces the creation of stable internal interfaces and consequently looser coupling between components.

By providing a clear separation of concerns between the layers performing marshalling and IPC and the logic it encapsulates we can provide seams that allow us to compose the same components in different ways to achieve different ends. For example one system I worked on had a number of very small focused 'services' which were distributed in the production deployment scenario, hosted in-process in a command line host for debugging and support, and scripted through PowerShell for administration of the underlying data stores. Supporting different modes of composability means that any underlying storage remains encapsulated because all access happens through the carefully designed interface rather than with ad hoc scripts and general purpose tools that end up duplicating behaviour or taking shortcuts. This makes the implementation harder to change due to the unintended tight coupling, or the tools go stale and become dangerous because they may no longer account for any quirks that have developed.

Always room for more

You might think that in this day and age most of our tooling problems have been solved. And yet one only has to look at the continued release of new text editors, new programming languages and new build systems to see that we are far from done yet. Even if your ambitions are far more modest, there are still likely to be many problems specific to your own domain and system that would benefit from a sprinkling of custom tooling to reduce the burden of analysis, development, testing, support, documentation, etc. Whilst we should be mindful of not unnecessarily reinventing the wheel or blinkering ourselves with a Not Invented Here (NIH) mentality, that does not mean we should also have to put up with a half-baked solution just to drink from the Holy Grail of Reusability. Even the venerable Swiss Army knife can't be used for every job. ■

References

[1] In The Toolbox: Wrapper Scripts, *C Vu 25-3*, http://www.chrisoldwood.com/articles/in-the-toolbox-wrapper-scripts.html

[2] Joel on Software, 'In Defense of Not-Invented-Here Syndrome', http://www.joelonsoftware.com/articles/fog0000000007.html

[3] Wikipedia, KA9Q, https://en.wikipedia.org/wiki/KA9Q

Why Floats Are Never Equal

Silas S. Brown tries his hand at optimising floating point equality comparisons.

Optimiser 1, Silas Brown 0

You're probably aware that comparing two floats or doubles for equality is a bad idea due to rounding error. If two numbers which you expect to be equal were calculated in two different ways, the two different calculations will likely have different rounding errors and you don't quite end up getting an equal result, so you have to do something like:

```
fabs(b-a) < .0001
```

where .0001 is a tolerance chosen appropriately for your application (there are ways of working it out in the general case, but if you know your application then you probably have a good idea of what sort of numbers you're dealing with). Do use `fabs` rather than coding it yourself, as many compilers will convert `fabs` to a single instruction.

I was doing some voluntary programming for a cancer research team [1], and it involved a thermodynamics calculation on fragments of DNA at different alignments. Once all possible alignments had been checked and we knew which one gave the lowest result, I wanted to display information about it. I had three options:

1. Calculate display information about ALL the items, store it, and then display only the one we found to be minimum;
2. Store the loop counter of the minimum item so far so we can go back to it;
3. Just store the minimum value, then go back through the whole loop and find it again.

Option 1 would run slower and moreover would be a memory management hassle to code. Normally I would have picked option 2, but in this case each loop iteration was making incremental changes to quite a few variables along the way, and on the other hand there weren't very many loop iterations, so I decided on option 3.

Now surely, I thought, this could be the one occasion where it *is* all right to save a couple of CPU cycles by comparing two floating-point values for equality. After all, the second version of the loop is exactly retracing the steps of the first, with exactly the same formulae and only the addition of some extra display code in between. The rounding error can't possibly be any different this time, can it?

Wrong.

The extra display code was nothing to do with the floating-point calculation, but it did affect the optimiser's decisions of which values to keep in registers and which ones to write back to memory. And it turned out the x86 CPU had 80-bit floating-point registers, which were being rounded down to 64-bit (or 32-bit) when writing to memory. So if the optimiser found enough room to keep an intermediate result in a register, it would be kept at 80-bit precision, but if it decided to spill that register to memory to make room for something else, precision would be lost. And optimisers these days have ways of using floating-point registers as holding bays for non-floating point data, so just because the extra code wasn't doing any floating point itself, didn't mean it wouldn't be compiled to something that 'wanted' those registers. The presence of this display code was causing the floating-point registers to be saved out to memory, and precision to be lost, making the equality comparison still wrong even though I was looking at the same C formulae in both cases.

I didn't spot this bug at first because, when I was developing, I happened to be using a Mac, which is based on BSD. BSD defaults to putting the x86 CPU into a mode where it always rounds its intermediate results, so there is no difference between a result held in a register and one written to memory. So I was getting away with it. But then we tried to run the same code on GNU/Linux, which *doesn't* tell the CPU to round, because it prefers accuracy to predictability. And it got stuck in an infinite loop looking for a minimum that didn't exist.

Floats are never equal. Well they are sometimes, but you don't know when, even if you think you do.

Of course there are ways to dodge the issue on x86 CPUs. You could find out how to change the CPU's rounding mode yourself (it involves assembly). Or you could make sure to always use 80-bit variables (usually called 'long double' by x86 compilers), although if you don't need that much precision then the extra memory operations would probably slow you down far more than a 'nearly equal' comparison would (and the same goes for GCC's `-ffloat-store` option, which prevents any intermediate floating-point values from being held in registers at all). But what if someone wants to compile your code on some other kind of CPU? Can you be sure that there won't be a similar problem?

Floats are *never* equal. Well they are sometimes, but you don't know when, even if you think you do. Not unless you're programming in assembly and you know exactly which intermediate results are held in which registers at which times, and the chances are your application doesn't make it worth your time to go there. Floats are never equal.

Finally, if you really want to save 2 CPU instructions and are looking for a known minimum, then instead of doing `fabs(x-minimum)<.0001`, you can first add .0001 to `minimum` outside the loop, then simply test for `x < minimum`. Do check this tolerance is appropriate to your application and can be represented by your chosen floating-point precision.

GCC has a warning when you try to compare two floats for equality, but you have to switch it on with the `-Wfloat-equal` option, which is sadly not included in `-Wall` or `-Wextra`. ■

Reference

[1] http://people.ds.cam.ac.uk/ssb22/pooler

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

SILAS S. BROWN

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

Smarter, Not Harder

Pete Goodliffe tries to solve the right problems the right way.

Battles are won by slaughter and manoeuvre. The greater the general, the more he contributes in manoeuvre, the less he demands in slaughter.

~ Winston Churchill

Life in the software factory can be hectic and fast-paced, with many unreasonable demands. "Make it super-elegant." "Make it feature-rich." "Make it bug-free." "And make it *now*!" With the pressures of unrealistic deadlines and tricky coding tasks looming over your head, it can be easy to lose focus, and deliver the wrong thing, or fail to deliver at all.

There's a trick, or is it an art, or is it simply a learned *skill*, to doing the *right thing* at the *right time*; to knowing how to solve the *right* problem; and to do as much (that is, as *little*) work as required to get to the *right* solution.

The wrong thing, the wrong way

Let me tell you a story. It's true. A colleague, working on some UI code, needed to overlay pretty rounded arrows over his display. After he struggled to do it programmatically using the drawing primitives provided, I suggested he just overlay a graphic on the screen. That would be much easier to implement.

So off he went. He fired up Photoshop. And fiddled. And tweaked. And fiddled some more. In this, the Rolls-Royce of image composition applications, there is no quick-and-easy way to draw a rounded arrow that looks halfway decent. Presumably an experienced graphic artist could knock one up in two minutes. But after almost an hour of drawing, cutting, compositing, and rearranging, he still didn't have a convincing rounded arrow.

He mentioned it to me in frustration as he went to make a cup of tea.

On his return, tea in hand, he found a shiny new rounded arrow image sitting on his desktop ready for use.

"How did you do that so quickly?" he asked.

"I just used the right tool," I replied, dodging a flying mug of tea.

Photoshop *should* have been the right tool. It's what most image design work is done in. But I knew that Open Office provides a handy configurable rounded arrow tool. I had drawn one in 10 seconds and sent him a screenshot. It wasn't elegant. But it worked.

The moral?

There is a constant danger of focusing too closely on one tool, or on a singular approach to solve a problem. It's tantalisingly easy to lose hours of effort exploring its blind alleys when there's a simpler, more direct route to your goal.

So how can we do better?

Pick your battles

To be a productive programmer, you need to learn to work *smarter* rather than *harder*. One of the hallmarks of experienced programmers is not just their technical acumen, but how they solve problems and pick their battles.

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net or @petegoodliffe

Good programmers get things done quickly. Now, they *don't* bodge things like a shoot-from-the-hip cowboy coder. They just work smart. This is not necessarily because they are more clever; they just know how to solve problems *well*. They have an armoury of experience to draw from that will guide them to use the correct approach. They can see lateral solutions – the application of an unusual technique that will get the job done with less hassle. They know how to chart a route around looming obstacles. They can make informed decisions about where best to invest effort.

Battle tactics

Here are some simple ideas to help you work smarter.

Reuse wisely

Don't write a lump of code yourself when you can use an existing library, or can re-purpose code from elsewhere.

Even if you have to pay for a third-party library, it is often far more cost effective to take an off-the-shelf implementation than to write your own. And test your own. And then debug your own.

> Use existing code, rather than writing your own from scratch. Employ your time on more important things.

Overcome *'not invented here'* syndrome. Many people think that they can do a much better job themselves, or fashion a more appropriate version for their specific application. Is that *really* the case? Even if the other code isn't designed exactly how you prefer, just use it. You don't necessarily need to rewrite it if it's working already. Make a facade around it if you must to integrate into your system.

Make it someone else's problem

Don't work out how to do a task yourself if someone already knows how to do it. You might like to bask in the glory of the accomplishment. You might like to learn something new. But if someone else can give you a leg up, or complete the job much faster than you, then it may be better to put the task in their work queue instead.

Only do what you have to

Consider sacrilege: Do you *need* to refactor? Do you *need* to unit test?

I'm a firm advocate of both practices, but sometimes they might not be appropriate or a worthwhile investment of your time. Yes, yes: refactoring and unit testing both bring great benefits and should never be tossed aside thoughtlessly. However, if you're working on a small prototype, or exploring a possible functional design with some throwaway code, then you might be better off saving the theologically correct practices for later.

If you do (laudably) invest time in unit tests, consider exactly *which* tests to write. A stubborn 'test every method' approach is not sensible. (Often you'll think you have better coverage than you expect). For example, you need not test every single getter and setter in your API. (It's another issue whether you *should* have getters and setters in your APIs in the first place.) Focus your testing on usage, not methods, and pay particular attention to the places you would likely expect brittleness.

Pick your testing battles.

Put it on a spike

If you're presented with multiple design options and you're not sure which solution to pick, don't waste hours cogitating about which is best. A quick *spike* solution (a throw-away prototype) might generate more useful answers in minutes.

To make this work well, set a specific Pomodoro-like time window within which you will perform the spike. Stop when the time elapses. (And in true Pomodoro style, get yourself a nice hard-to-ignore windup timer to force you to stop.)

Use tools that will help you backtrack quickly (e.g., an effective version control system).

Prioritise

Prioritise your work list. Do the most important things first.

> Concentrate effort on the most important things first. What is most urgent, or will produce the most value?

Be rigorous about this. Don't get caught up on unimportant minutiae; it's incredibly easy to do. Especially when one simple job turns out to depend on another simple job. Which depends on another simple job, which depends on.... After two hours you'll surface from a rabbit hole and wonder why on earth you're reconfiguring the mail server on your computer when what you wanted to do was modify a method on a container class. In computer folklore, this is referred to as yak shaving [1].

Beware of the many small tasks you do that aren't that important; email, paperwork, phone calls – the administrivia. Instead of doing those little things throughout the day, interrupting and distracting you from your flow on important tasks, batch them together and do them at one (or a few) set times each day.

You may find it helps to write these tasks down on a small 'to do' list, and at a set time start processing them as quickly as possible. Ticking them off your list – the sense of accomplishment can be a motivating reward.

What's really required?

When you are given a new task, check what's *really* needed now. What does the customer actually need you to deliver?

Don't implement the Rolls-Royce full bells-and-whistles version if it's not necessary. Even if the work request asks for it, push back and verify what is genuinely required. To do this, you need to know the context your code lives in.

This isn't just laziness. There is a danger in writing too much code too early. *The Pareto principle* [2] implies that 80% of required benefit could come from just 20% of the intended implementation. Do you really need to write the remainder of that code, or could your time be better employed elsewhere?

One thing at a time

Do one thing at a time. It's hard to focus on more than one job at once (especially for men with our uni-tasking brains). If you try to work concurrently, you'll do both jobs badly. Finish one job then move on to another. You'll get both jobs completed in a shorter space of time.

Keep it small (and simple)

Keep your code and design as small and as simple as possible. Otherwise, you'll just add a lot more code that will cost you time and effort to maintain in the future.

> Remember *KISS*: **K**eep **I**t **S**imple, **S**tupid.

You *will* need to change it; you can never foretell exactly what the future requirements are. Predicting the future is an incredibly inexact science. It is easier and smarter to make your code malleable to change now than it is to build in support for every possible future feature on day one.

A small, focused body of code is far easier to change than a large one.

Don't defer and store up problems

Some things that are hard (like code integration) should *not* be avoided because they are hard. Many people do so; they defer these tasks to try to minimise the pain. It sounds like picking your battles, doesn't it?

In reality, the smarter thing is to start sooner and face the pain when it is smaller. It's easier to integrate small pieces of code early on, and then to frequently integrate the subsequent changes, than it is to work on three major features for a year and then try to stitch them together at the end.

The same goes for unit testing: write tests now, alongside your code (or before). It'll be far harder, and less productive, to wait until the code is 'working' before you write the tests.

As the saying goes: *If it hurts, do it more often.*

Automate

Remember the classic advice: *if you have to do it more than once, write a script to do it for you*.

> If you do something often, make the computer do it for you. Automate it with a script.

Automating a common, tedious task could save you many hours of effort. Consider also a single task that has a high degree of repetition. It might be faster to write a tool and run that once, than to do the repetitive job by hand yourself.

This automation has an added advantage: it helps others to work smarter, too. If you can run your build with *one* command, rather than a series of 15 complex commands and button presses, then your entire team can build more easily, and newcomers can get up to speed faster.

To aid this automation, experienced programmers will naturally pick automatable tools, even if they don't intend to automate anything right now. Favour workflows that produce plain text, or simple structured (e.g., JSON or XML) intermediate files. Select tools that have a command-line interface as well as (or instead of) an inflexible GUI panel.

It can be hard to know whether it's worth writing a script for a task. Obviously, if you are likely to perform a task multiple times then it's worth considering. Unless the script is particularly hard to write, you are unlikely to waste time writing it.

Error prevention

Find errors sooner, so you don't spend too long doing the wrong thing.

To achieve this:

- Show your product to customers early and often, so you'll find out quickly if you're building them the wrong thing.
- Discuss your code design with others, so you'll find out if there's a better way to structure your solution earlier. Don't invest effort in bad code if you can avoid it.
- Code review small, understandable bits of work often, not large dense bits of code.
- Unit-test code from the outset. Ensure the unit tests are run frequently to catch errors before they bite you.

Communicate

Learn to communicate better. Learn how to ask the right questions to understand unambiguously. A misunderstanding now might mean you'll end up reworking your code later on. Or suffer delays waiting for more answers to outstanding questions.

Learn how to run productive meetings so your life is not sucked out by the demons who sit in the corners of meeting rooms.

An Introduction to OpenMP

Silas S. Brown dabbles in multiprocessing to speed up his calculations.

If you use a CPU that was manufactured during the last few years, then the chances are it has more than one core, most likely two or four. Multi-core programming can be difficult (I would certainly recommend putting in a little effort to make sure you're using a fast-enough algorithm on one core first), but it was made easier by GCC's adoption of the OpenMP (Open Multi-Processing) standard since version 4.2 (2007). If you use a recent version of GCC, you might have OpenMP without knowing it. Try:

```
gcc my-program.c -fopenmp
```

and see whether or not it calls it an unknown option. (I do this in a script to decide what compilation options to use on a deployment machine.)

Adding OpenMP directives to a program can be surprisingly simple. Consider a `for` loop:

```
for (int i=0; i < nItems; i++)
  process_item(i);
```

If `process_item` looks only at item `i` and nothing else (no memory conflicts) then all you need to add before the `for` is:

```
#pragma omp parallel for
```

and, by default, the OpenMP library will find out at runtime how many cores are available on the CPU, split into that number of threads, divide `nItems` by the number of threads, let each thread process its 'chunk' of the items, and wait for them to finish. It will also add code to let the user override the number of threads at runtime by setting an environment variable (`OMP_NUM_THREADS`). This is all rather powerful just for one `#pragma`. Of course, if that code is compiled without OpenMP support, the pragma will be ignored and the code will run sequentially. But some compilers warn about unknown pragmas, so to suppress this warning you could wrap the pragma in an `ifdef`:

```
#ifdef _OPENMP
#pragma omp parallel for
#endif
```

which you can even extend to let you use macros to control exactly which parts of your program are parallelised:

```
#define Parallelise_The_XYZ_Loop 1
...
#if defined(_OPENMP) && Parallelise_The_XYZ_Loop
#pragma omp parallel for
#endif
```

Since the extra work of creating and managing the threads has an overhead, you should only use `parallel for` if you're sure the benefits will be worth the overhead. For very short loops, you might actually slow the program down. Always measure to check you are actually getting a speed increase.

By default, the loop counter and any variable you declare inside the loop will be private to that thread, but other variables will be shared, so if you want to change them you had better write a `critical` section to ensure only one thread at a time can get in:

```
#pragma omp critical
update_a_shared_variable();
```

`critical` is not needed if all you're doing is writing to an array when the element number you write to is the item number you're processing, as

SILAS S. BROWN

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

Smarter, Not Harder (continued)

Avoid burnout

Don't burn yourself out working silly hours, leading people to expect unrealistic levels of work from you all the time. Make it clear if you are moving beyond the call of duty, so people learn not to expect it too often.

Healthy projects do not require reams of overtime.

Power tools

Always look out for new tools that will boost your workflow.

But don't become a slave to finding new software. Often new software has sharp edges that could cut you. Favour tried-and-tested tools that many people have used. You can't put a price on the collected knowledge of these tools available via Google.

Conclusion

Pick your battles. (Yeah, yeah.) *Work smarter, not harder.* (We've heard it all before.)

They are trite maxims. But they are true.

Of course, this doesn't mean *don't work hard*. Unless you want to get fired. But that's not smart. ■

Questions

1. How do you determine the right amount of testing to apply to your work? Do you rely on experience or guidelines? Look back over your last month's work; was it really tested adequately?
2. How good are you at prioritising your workload? How can you improve?
3. How do you ensure you find issues as soon as possible? How many errors or re-workings have you had to perform that could have been avoided?
4. Do you suffer from *not invented here* syndrome? Is everyone else's code rubbish? Could you do better? Can you stomach incorporating other's work in your own?
5. If you work in a culture that values the number of hours worked over the quality of that work, how can work reconcile 'working smart' with not looking lazy?

References

[1] http://bit.ly/Y1J0f

[2] For many events, roughly 80% of the effects come from 20% of the causes. For more on this, see http://en.wikipedia.org/wiki/Pareto_principle

the other threads won't be writing to the same element. But it is often needed in other shared-variable circumstances; you are going to have to think.

One pattern that is often seen in OpenMP programming is to check if a shared variable needs updating, then enter a **critical** section and repeat the check:

```
if (shared_variable_needs_updating())
  #pragma omp critical
  if (shared_variable_needs_updating())
    update_a_shared_variable();
```

The second check is there in case another thread beats us to it with updating the shared variable. For example, this might be used if the shared variable is 'best solution found so far': just because we found a better solution outside the **critical** section doesn't mean nobody else posted an even better one just before we entered it. We could save the extra comparison by entering the **critical** section unconditionally and THEN making the comparison, but that would be inefficient because it would hold up other threads unnecessarily.

One trick that might be useful during debugging is to add **default(none)** to the end of the **parallel for** pragma. That tells OpenMP to refrain from its default behaviour of making variables within the loop private to each thread and other variables shared, and forces you to declare the shared/private status of each variable explicitly. If you haven't done so, you get some handy error messages pointing out each variable referred to from the parallel section. This can be very useful indeed when retro-fitting OpenMP to existing code and the loop is too large for you to be sure you've noticed everything.

parallel for can take only **normal** for loops that count items as they go; trying to be more 'clever' with the **for** statement will not work with OpenMP. You may use the **continue** statement in a **parallel for**, but not **break** (unless it's inside another loop etc that's nested inside the parallel one), and not **return**. This is for obvious reasons: there would be no way for the OpenMP libraries to make sure that **break** or **return** stops other iterations of the loop if some other thread is already running away with them.

By default, **parallel for** assumes that each loop iteration will be roughly equal, and so it splits the number of required iterations evenly among the threads. You could instead add **schedule(dynamic)** to the pragma to take the alternative approach of sending just one iteration at a time to each thread (so for example if there are four cores, the first four iterations will be started on immediately, and as soon as one of the cores finishes its iteration it will be given the fifth iteration to do), but that tends to work well only if each iteration is quite long; if iterations are short then the overheads of managing the **dynamic schedule** slow things down. You can however do your own scheduling: instead of using **parallel for**, just say:

```
#pragma omp parallel
some_function_or_block()
```

which will run *N* identical copies of **some_function_or_block()**; these copies will then need to work out amongst themselves which one does which section of work. To help with this, `omp.h` defines the functions **omp_get_thread_num()** and **omp_get_num_threads()**: the thread number will be between 0 and threads-1 inclusive. Since I like to make sure my programs can still compile even if OpenMP is not present, I do this:

```
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_num_threads() 1
#define omp_get_thread_num() 0
#endif
```

You have to be careful, when dividing your work units by the number of threads, to make sure no work is left out due to the division result being rounded down. If your units are fairly even then it's probably best just to use OMP's own **parallel for** which does all the work for you.

Signals are usually sent to an arbitrary thread, so the best thing to do in a signal handler is probably just to set a flag which all threads regularly check.

OpenMP works in C++ as well, but if you are using a lot of objects then you might need to be even more careful of where you put your **critical** sections.

Besides GCC, other compilers that support OpenMP include Visual C++ (from its 2005 version onward) and the Intel compiler, but I haven't tried these. Clang 3.7 supports it, but some older Macs (e.g. OS X 10.7) have both Clang and GCC installed where the GCC supports OpenMP but the Clang does not. OpenMP implementations are generally limited to multicore CPUs with shared memory, as in a modern multicore desktop; more advanced approaches are needed if you're running on a supercomputer or cluster that does not share its memory between all the cores, or if you want to run your processing on graphics cards (GPUs).

On slightly older Apple computers, there's some strange bug that means you can't call **memcpy()** from inside a function that uses OpenMP: you have to wrap that **memcpy()** into another function of your own and call that. But the function you wrap it in can be 'inline' so you don't actually lose anything. If you get other problems on Apple, try:

```
#define _FORTIFY_SOURCE 0
```

as a workaround.

Finally, if you are cross-compiling for Windows using MingW, you might want to use the **-static** flag to make sure the `.exe` file doesn't depend on OpenMP and threading DLLs. Windows `.exe` files are easier to distribute if they don't need DLLs. ■

Random Confusion

Silas S. Brown tries to clear up a muddle about Standard C's rand().

The Standard C way of obtaining random numbers is to call **srand()** and **rand()**, defined in stdlib.h. Unfortunately, the manual pages of BSD state categorically that **rand()** is not very good, and a function called **random()** should be used in its place. This includes Mac OS X (which was derived from BSD), and it would appear that some poorly-written books have copied this advice without realising that **random()** is not actually Standard C. Searching the Web will show quite a few discussion pages where beginners have read books that told them to use **random()** instead of **rand()**, and others have told them this is not Standard C and the book is not very good, but they rarely note where the confusion came from. (I would chime in myself on those threads if it weren't so much hassle to sign up for an account on each one.)

So perhaps we should make a little public service announcement: **rand()** is Standard C; **random()** is Standard POSIX.

True, **random()**'s being in POSIX means it's available on all modern Unix systems including BSD, Mac OS X and GNU/Linux. But it's not there on Windows and the confusion is (for once) not Microsoft's fault. It's also unlikely to be there on other platforms (PDAs and so on).

When the BSD manual says **rand()** is not very good, what it really means is, BSD's implementation of it is not very good. The C standard gives a sample implementation of **rand** but does not mandate that particular implementation. BSD (at least the version of it in Apple's source) essentially just multiplies the previous number by 16807 and leaves it at that. In the early 1980s, 4.2BSD (1983) and 4.3BSD (1986) introduced **random()** to be a 'better **rand**', but they didn't upgrade the behaviour of their **rand()**, presumably because they didn't want to break any programs which relied on their old implementation-specific behaviour of this function.

The GNU/Linux C library (at least nowadays) does the sensible thing and makes **rand()** equal to **random()**, so you get the better behaviour no matter which one you call. In that case I suggest you call **rand()** because then you're writing Standard C. But their manual page suggests you prefer **random()** because then your code will work better on BSD.

The Windows implementation of **rand()** is better than BSD's: it multiplies by 214013, adds 2531011 and returns bits 30 to 16 of the result. Of course, that's not good enough for cryptography (and you certainly shouldn't rely on the standard **rand()** being good enough for cryptography on any platform; if you want cryptographically-strong random numbers, make sure you can find out how to get them properly!), but it should be fine for most scientific or simulation purposes and you don't have to worry about disregarding the low-order bits (which is just as well, as there are only 15 bits to start with).

(Versions of WINE prior to 2006 just did **rand()** & 0x7fff, but this was then replaced with an implementation of what MSVC actually does. So if you're on Unix and cross-compiling for Windows, you can run your program in WINE and get the same random sequence that it would get on Windows if seeded to the same value.)

Listing 1

```
#include <stdlib.h>
#include <sys/param.h>
#ifdef BSD
/* avoid BSD's bad old rand() implementation */
#define rand random
#define srand srandom
#endif /* BSD */
```

So what to do about **random()**? A first thought might be 'use **rand()** on Windows, **random()** everywhere else' but that won't help with ports to non-Windows non-Unix systems (yes these do exist). Since GNU/Linux nowadays gives you good behaviour no matter which one you call, it would be better to say 'use **random()** on BSD, **rand()** everywhere else'. You can do that by checking for the **BSD** macro, which is defined in sys/param.h (a header file which is also available when cross-compiling for Windows) – see Listing 1.

If you want to make your program's behaviour 100% reproducible no matter which system it is compiled on, you had better provide your own random number generator

If you want to make your program's behaviour 100% reproducible no matter which system it is compiled on, you had better provide your own random number generator, because you never know which one you're going to get from the standard library. Seeding it to the same value makes the sequence reproducible only on that particular implementation of the C library (and that particular version of it at that: most platforms are not as obsessive as BSD is about keeping their old behaviour from version to version). The code in Listing 2 should reproduce Microsoft's behaviour; I'm setting the functions to static so they won't be visible outside the current C module, just in case some library function depends on the library's implementation of **rand()**. Feel free to use this in your code as it seems to be a public-domain method (at least I hope so; if Microsoft sues you for using two of their numbers, change them! but do check some good texts for alternatives, as you're less likely to end up with a good generator if you just pick them out of thin air yourself.)

Listing 2

```
#ifdef RAND_MAX
#undef RAND_MAX
#endif
#define RAND_MAX 0x7FFF
static unsigned long seed = 1;
static void srand(unsigned int newSeed) {
   seed = newSeed;
}
static int rand() {
   seed = (seed*214013L) + 2531011L;
   return (seed >> 16) & 0x7FFF;
}
```

SILAS S. BROWN

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

High Rollers

Baron M proposes a new wager over a glass of wine.

Sir R-----, my fine fellow, it does my heart good to see you upon this summer's eve! Will you take a glass of muscatel and, perchance, a wager?

As I should have expected, you have not disappointed me sir!

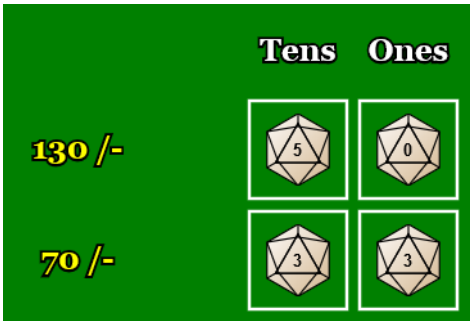
Might I propose a game native to the Isle of Cockaigne, that land of plenty where the fountains run with this elixir, where the vintners string up their vines with sausages and where, whensoever it rains, it rains gravy?

I first visited Cockaigne upon the direction of the Pope, who had instructed me to do penitence by travelling there. What cause there should have been I cannot fathom, but I shall never allow it to be said that I have been derelict in my duties, neither earthly nor heavenly. The journey was an arduous affair, some twenty-eight months' sailing, during which we exhausted our provisions and were forced to make stew of the fo'c'sle. Needless to say, we were most relieved to find that at our destination no fellow wanted for aught and we had ourselves a feast that would have put the Pope's court to shame!

Having no need to strive for their daily bread, the denizens of that fair island took to chance to decide who should be the master and who the servant. Each day they diced for their roles and it is by the rules that they did so that I suggest we gamble.

Here, I have chalked out upon the table two spaces for each of us, one marked for tens and the other for ones. I shall cast a twenty-sided die upon which are struck the digits zero to nine twice over and, once it has settled upon a number, I shall choose which of my spaces to place it upon. You shall then do the same with one of these two dice. We shall then each cast another and place it upon the space that we have yet not chosen and if the number that you have built exceeds mine then you shall have a prize of twenty nine coins from me, otherwise I shall have one of thirty coins from you.

That godforsaken student, whose unrelenting nonsense it seems that I must unceasingly bear witness to, upon having the rules of this game



painstakingly explained to him, commenced to blathering on and on about the tragedies that he had suffered at the hands of his opticians. As a marksman of the first water, I must confess that I have little sympathy for the trouble that those who bury their noses in books bring upon their eyes, damn them!

Now that's more than enough of that wretch! Come, take another glass and think upon your chances! ■

Baron M

Courtesy of www.thusspakeak.com

BARON M

In the service of the Russian military the Baron has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.

Random Confusion (continued)

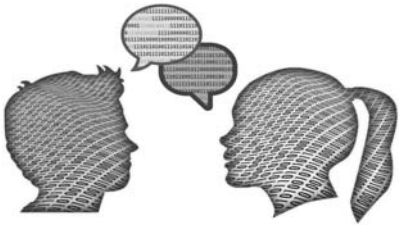
The XorShift generator discovered by George Marsaglia in 2003 gives generally better random numbers and is usually faster. It can be implemented thus:

```
#include <stdint.h>
int XS_rand() {
  static uint64_t s=1;
  s^=s>>12; s^=s<<25; s^=s>>27;
  return s & 0x7FFFFFFF;
}
```

On x86 processors, bit shifts and XORs can be done in one cycle each, whereas multiplications can take 3 or 4 cycles depending on the processor (although pipelining and other types of instruction-level parallelism can hide that latency in some circumstances). Even additions can take multiple cycles on some CPUs. Thus Xorshift (which takes three XORs and three shifts) is almost certainly going to be at least as fast, and likely faster, than a common linear congruential generator which uses multiplication and addition. ■

Code Critique Competition 100

Set and collated by Roger Orr. A book prize is awarded for the best entry.



Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Note: If you would rather not have your critique visible online, please inform me. (Email addresses are not publicly visible.)

Last issue's code

I wanted to do something slightly different for the 100th code critique column, so I based this critique on the 'left-pad' function that was part of npm but was withdrawn by the author, breaking a large number of components on the Internet.

See http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm for some more official information about the issue and this blog:http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/ for some discussion about some of the issues it raises.

Please feel free to comment on the Javascript code itself, or on the wider issues raised by the story.

Listing 1 contains `leftpad.js`, and a trivial test page is provided in Listing 2 if you want to play with the function in a browser.

Listing 1

```
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

Listing 2

```
<!DOCTYPE html>
<html>
<head><title>CC100</title></head>
<body>
<h1>Code Critique 100</h1>
<script src="leftpad.js"></script>
<script>
function testLeftpad() {
  result.innerHTML = '"' +
  leftpad(text.value, len.value, pad.value) +
  '"';
}
</script>
<table>
<tr><td>Text to pad</td><td>
  <input type="text" id="text" value="1234">
</td></tr>
<tr><td>Length</td><td>
  <input type="text" id="len" value="10">
</td></tr>
<tr><td>Pad char</td><td>
  <input type="text" id="pad" value="0">
</td></tr>
</table>
<br/><button onclick="testLeftpad()">
Try out leftpad
</button>
<pre id="result"></pre>
</body>
</html>
```

Critique

Juan Zaratiegui <yozara@outlook.com>

I am not a Javascript fan or expert, but languages have similar foundations, so let's give it a try.

For a start, let's look at the function as a whole: we are trying to pad the left of a string with some **`char`** until we fill a specified length.

This length comes from the parameter **`len`**, and it is never checked against the natural limits: it should be a positive integer, and greater than the original string length to be meaningful.

If the length desired is less or equal to the original string length, no operation should be performed and the original string would be returned.

Now that we have corrected length treatment, let's look at the fill **`char`**. We want 1 fill **`char`**, but there is no precaution taken about it. Instead, we have a strange condition that replaces a string that is simultaneously empty and '0' with a space. That will never happen.

Instead we will make three tests: empty string, string of length not 1 and **`string = Char(NULL)`**. If any of these tests are true, we will use the default string `' '` to fill.

And finally, as a question of cleanness, I will use local variables with names different from the parameters, thus giving the following listing:

```
function leftpad (str, len, ch) {
  result = String(str);
  var i = -1;
  if (!ch || ch.length != 1 ||
    ch.charCodeAt(0) == 0) ch = ' ';
  if (len > str.length ) {
    missing = len - str.length;
      while (++i < missing) {
        result = ch + result;
  }
 }
 return result;
}
```

Commentary

There are a number of issues that could be covered with this story. The original one was that the code as written appears to have a number of problems (or potential problems) but there are some rather wider questions that I might touch on too.

ROGER ORR

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

The requirements on the input types and values for **leftpad** are unclear. Javascript is a dynamically typed language so the three parameters can be filled by a variety of different types at runtime; which ones are expected?

The first argument is converted to a String so can be (almost) any type – although passing it an arbitrary Object, for instance, is unlikely to produce useful output. The likely data types are String and Number and either of these will 'do the right thing' but this has to be inferred.

The second argument is the desired output string length. As Juan noted, it's not clear what the effect of providing an out of range value ought to be. However, the code as written will make no change to **str** if **len** is less than the string length or negative. It's not clear whether the behaviour if the value is non-integral is expected (eg. len = 10.01 pads to 11).

The third argument, **ch**, is the fill character. Except when it isn't. The complication is the line that 'sanitises' the fill character:

```
if (!ch && ch !== 0) ch = ' ';
```

The first expression, **!ch**, will be true if **ch** is undefined, **null**, **false**, +0, -0 or NaN, or an empty string. The second expression, **ch !== 0**, will be true if **ch** is either not a number or is a non-zero number. So the overall effect of the line is to set the fill character to a space if the argument was undefined, **null**, **false**, NaN or an empty string. Note that **ch** will be undefined when the argument is omitted, hence 'by default' the function pads with spaces.

However, as Juan also noted, there is no check that the string representation of **ch** (used later when actually padding the string) is of length 1. If the length is greater than 1 the code will prepend *n* copies of the string to **result**, leading to a wider string than expected being returned. We could use **ch.charAt(0)** to create a single-character fill. However, if **ch** is supplied as a number, not as a string, the call to **charAt(0)** will fail. It might be better to *explicitly* convert **ch** into a string first.

This is obviously not the only design option; another option is to prepend enough copies of the string so the new length is a least that desired: **leftpad("+-", "test", 7)** would then produce as output **"+-+-test"**.

Finally, the performance of the padding itself is non-optimal: the loop creates a new string each time which is one character longer with the new character at the front. For modern (ECMAScript 6) browsers the **repeat** function can be used to create a string of the right length and remove the loop completely:

```
if (len > 0)
  str = ch.repeat(len) + str;
```

If the code has to run on slightly older browsers which do not implement **repeat()** then it might be worth supplying a function with similar semantics.

The current version of npm's **leftPad** function builds up the padding using a double-and-add algorithm based on the binary decomposition of the length required.

From https://github.com/stevemao/left-pad:

```
var cache = [
  '',
  ' ',
  '  ',
  '    ',
  '      ',
  '        ',
  '          ',
  '            ',
  '              ',
  '                '
];

function leftPad (str, len, ch) {
  // convert `str` to `string`
  str = str + '';
  // `len` is the `pad`'s length now
  len = len - str.length;
  // doesn't need to pad
  if (len <= 0) return str;
  // `ch` defaults to `' '`
  if (!ch && ch !== 0) ch = ' ';
  // convert `ch` to `string`
  ch = ch + '';
  // cache common use cases
  if (ch === ' ' && len < 10)
    return cache[len] + str;
  // `pad` starts with an empty string
  var pad = '';
  // loop
  while (true) {
    // add `ch` to `pad` if `len` is odd
    if (len & 1) pad += ch;
    // devide `len` by 2, ditch the fraction
    len >>= 1;
    // "double" the `ch` so this operation
    // count grows logarithmically on `len`
    // each time `ch` is "doubled", the `len`
    // would need to be "doubled" too
    // similar to finding a value in binary
    // search tree, hence O(log(n))
    if (len) ch += ch;
    // `len` is 0, exit the loop
    else break;
  }
  // pad `str`!
  return pad + str;
}
```

This is a degree of cleverness that may only be required for some uses of the left pad function; but given how widespread use of the original function was I suspect it is worth doing. I think though that I'd have possibly made the creation of the fill string into a separate function as this can be useful on its own.

The replacement code comes with a simple test program, which is good, but it doesn't cover some of the troublesome cases for input arguments of unexpected types or ranges. mentioned earlier.

The bigger questions that this raises are at least partly covered in the two links I gave in the critique preamble.

I think for me a couple of the major concerns are:

- the number of hits the website took in a a very small time implies that an *outage* of the website (which could be caused by a number of things) would have a wide ripple effect.
- relying on code that relies on other code. A lot of people were hit by the removal of left-pad although they didn't use it directly but used something that did.

Taken together this means a number of people had a hard-to-diagnose failure because of a tangle of dependencies outside their direct control.

This seems to me quite a large 'business' risk and I'm not sure whether those using this sort of 'live' dependency on the Internet are aware of the potential problems.

There is then a security risk of relying on unexamined code; in this day and age I think you have to be careful about the possibility of both code with accidental security holes and of malicious code being injected into a website.

The winner of CC 100

I was a little disappointed that the discussions on accu-general didn't result in more entries, but thank you to Juan for providing his critique.

He appears to have been led a little astray by the slightly unusual comparison modes supported by Javascript (so the proposed solution no longer handles using a literal 0 as a fill value) but I think he still deserves the prize for this critique.

Code critique 101

(Submissions to scc@accu.org by Oct 1st)

I'm trying to read a list of test scores and names and print them in order.

I wanted to use exceptions to handle bad input as I don't want to have to check after every use of the **>>** operator.

However, it's not doing what I expect.

I seem to need to put a trailing /something/ on each line, or I get a spurious failure logged, and it's not detecting invalid (non-numeric) scores properly: I get a random line when I'd expect to see the line ignored.

The scores I was sorting:

```
-- sort scores.txt --
34 Alison Day
45 John Smith
32 Roger Orr
XX Alex Brown
```

What I expect:

```
$ sort_scores < sort_scores.txt
Line 4 ignored
32: Roger Orr
34: Alison Day
45: John Smith
```

What I got:

```
$ sort_scores < sort_scores.txt
Line 2 ignored
Line 3 ignored
0:
32: Roger Orr
34: Alison Day
45: John Smith
```

I tried to test it on another compiler but gcc didn't like

```
iss.exceptions(true)
```

I tried

```
iss.exceptions(~iss.exceptions())
```

to fix the problem.

Can you help me understand what I'm doing wrong?

Listing 3 contains `sort_scores.cpp`.

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://accu.org/index.php/journal). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 3

```
#include <iostream>
#include <map>
#include <sstream>
#include <string>
using pair = std::pair<std::string,
  std::string>;
void read_line(std::string lbufr,
  std::map<int, pair> & mmap)
{
  std::istringstream iss(lbufr);
  iss.exceptions
#ifdef __GNUC__
    (~iss.exceptions());
#else
    (true); // yes, we want exceptions
#endif
  int score;
  iss >> score;
  auto & name = mmap[score];
  iss >> name.first;
  iss >> name.second;
}
int main()
{
  std::map<int, pair> mmap;
  std::string lbufr;
  int line = 0;
  while (std::getline(std::cin, lbufr))
  try
  {
    ++line;
    read_line(lbufr, mmap);
  }
  catch (...)
  {
    std::cout << "Line " << line
      << " ignored\n";
  }
  for (auto && entry : mmap)
  {
    std::cout << entry.first << ": "
      << entry.second.first << ' '
      << entry.second.second
      << '\n';
  }
}
```

Standards Report

Jonathan Wakely brings the latest news.

In June the C++ committee met in Oulu, Finland, and voted out the C++17 Committee Draft (CD). The CD represents what the committee think C++17 should be and is sent out to the ISO National Bodies for review and balloting. The National Bodies will respond with a Yes or No vote (with comments saying what they think should be changed) and the committee will spend the next couple of meetings addressing those comments, before sending the final document to Geneva for standardisation.

At the Oulu meeting a number of new features were approved, as well as a huge number of defect reports resolved. Some of the highlights include **`constexpr if`**; inline variables; allocation of over-aligned data; guaranteeing expression evaluation order (but not function argument evaluation order); template argument deduction for class templates; guaranteed copy elision; and **`std::variant`**. A last minute surprise that wasn't expected to make it into C++17 was the 'structured bindings' proposal, which allows variables to be declared and initialized from tuple-like types, for example:

```
auto [position, inserted] = set.insert(val);
```

will declare the variables **`position`** and **`inserted`** and initialize them to the 'first' and 'second' members of the pair returned by the **`insert`** function.

If you would like to help review the CD you can download N4604 [1] and send comments via your National Body (or contact someone on a National Body and persuade them to do so for you). The ballot for the CD ends in October, and the next C++ committee meeting will be in Issaquah, WA, in November. At that meeting we'll be starting to respond to NB ballot comments, and (I hope) voting out a Draft Technical Specification (DTS) of the Networking library extensions.

Reference

[1] N4606 is available from http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4604.pdf

JONATHAN WAKELY

Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee. He can be reached at accu@kayari.org

View from the (Acting) Chair

Bob Schmidt
chair@accu.org

I'm starting this month's 'View' in mid-July. By the time it is published, September will have arrived and summer will be almost over. The time lag between writing and publishing is something I'm going to have to get used to. Let's tuck in to this month's subjects.

Special General Meeting As I mentioned in the last issue, the positions of Chair and Secretary are being held by caretakers; a Special General Meeting needs to occur to vote on the positions. Malcolm Noyes has been performing the Secretary's duties, and I have been acting as Chair; we both are standing for election. As announced on accu-members at the end of June, the SGM will be held on Wednesday 28th September in Oxford in combination with the ACCU Oxford local group. The SGM is scheduled for 7:00 PM at St. Aldates Tavern, 108 St. Aldate's, Oxford, Oxfordshire. If you plan to attend in person, you will need to sign up on ACCU Oxford's Meetup site – watch for the announcements.

The last date for nominations for a position was July 30th, 2016 (the 'proposal deadline', section 5.3.3 of the constitution). As of the writing of this View, no other persons have volunteered or been nominated to stand for election, so, according to section 5.3.4 the nominations stand at Malcolm and me.

Diversity statement Last issue I put out a call to anyone interested in participating in the drafting of a diversity statement for ACCU. I received feedback from both persons to whom I sent targeted requests; unfortunately, I haven't heard from anyone else. With the committee's agreement, I'm going to leave the process open to comments for another two-month magazine cycle, with the goal of finalizing the statement before the end of the calendar year.

Here is the draft Diversity Statement I shared with the committee at the May committee meeting:

> ACCU is committed to a culture of diversity and inclusion. We embrace and encourage our members' differences – including, but not limited to age, colour, ethnicity, ability or disability, gender identity or expression, sex or sexual orientation, language, national origin, religion or lack thereof, race, political persuasion, socio-economic status, and veteran status. ACCU will not tolerate discrimination, harassment, or bullying; in any form, for any reason. Our members deserve to be treated fairly, equally and with respect, and are expected to treat others the same way.

ACCU already has a Code of Conduct (CoC) for our conference. The Diversity Statement (DS) complements the CoC. Together they represent the values of our organization. Our next step will be to develop and publish a procedure to be followed if and when someone violates either the CoC or the DS, so that our response as an organization is appropriate and consistent.

Website migration Jim Hague volunteered to migrate the ACCU website and associated software to a new platform. The new site went live in mid-July. I'll have more to report in the next issue of *C Vu*, but this month I want to convey to Jim the thanks of the committee for all his hard work, and hope that you will join us in doing so. I expect that it was a bigger effort than Jim thought when he volunteered.

Also, please join the committee in thanking Tim Pushman for hosting the web site for more than 10 years.

Call for volunteers We still have several positions in search of volunteers:

- We need an additional auditor to fill out my term. Please send me an email if you are willing to join Guy Davidson on the auditing team.
- We are still looking for volunteers to make improvements to the web site. In particular, Russel Winder, our Conference Chair, has some ideas for improving the conference section of the site, and has asked for volunteers. I'll let him describe his ideas for the upgrade:

  Say ACCU was rearranging its Web resources. Say the ACCU Conference website was going to be separated from the main ACCU site in terms of the infrastructure used, but not the harmony and integration, clearly the ACCU Conference pages are an integral part of the ACCU website. Say we wanted to do the ACCU Conference bit sooner rather than later, not least because we want to set up the 2017 conference session submission system, and possibly the session reviewing system.
  I am thinking of using a Nikola/Python/Flask/MongoDB/Django/SQLite-type tool chain to achieve the goal, though I can easily entertain alternatives if there are any that are likely to be good and work solely with Debian packaged software.
  Given all this, is there anyone out there who might be interested in volunteering to assist in getting something together over the summer?

  Please contact Russel directly at conference@accu.org if you are interested in helping.
- We also have a more general request for anyone interested in serving on the conference committee. You would be working with Russel and Roger Orr on putting together the program for the April 2017 conference in Bristol.
- In addition to help with web site improvements, we are hoping to recruit someone to assist with the web site sys admin duties. Please contact me if you are interested.

In keeping with our commitment to value diversity, we encourage all members, including but not limited to members of under-represented groups, to volunteer for all positions.

Committee spotlight Our Treasurer is Rob Pauer, back for another year of keeping our books. Rob has been treasurer since 2011, when he took over from Stewart Brodie. He has been a member of ACCU since 1987, shortly after its foundation. He joined so that he could learn about C programming (it seemed more useful than Fortran) but never took it up as a profession so he remains an interested amateur and if he can understand a solution to 'Code Critique' he's quite happy.

Retirement from a career in insurance and pensions gave him an opportunity to help the Association in financial matters and he intends to continue in that role until someone else volunteers!

Rob put together the 2015 financials that were presented to the membership in the AGM packet. For those of you who didn't read through the details, here's an overview of our finances for the year ending 31 December 2015 (all numbers in British Pounds, rounded):

Income

Membership	28,160.00
Advertising	3,625.00
Local Group Sponsorship	270.00
Interest	6.00
Conference	5,952.00
Total Income	**38,013.00**

Expenses

Journal Design	8,400.00
Printing & Postage	21,744.00
Accountancy, Legal, Professional	743.00
Direct Costs, Office Expenses	774.00
Depreciation	175.00
Total Expenses	**31,836.00**
Net Income	**6,177.00**

As you can see, the association once again has run a surplus for fiscal year 2015. (The surplus is approximately the same as for 2014, which was £6,239.00). In general, membership and advertising income cover our expenses, and the conference supplies our surplus. The committee does not foresee a significant change to this situation for 2016.

Journals Finally, please consider writing something for *C Vu* or *Overload*. Steve and Fran would love to hear from you. The content of our magazines is mostly member-generated – without you, there wouldn't be an *Overload* or *C Vu*.

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe