the magazine of the accu

www.accu.org

{cvu}

Volume 27 • Issue 6 • January 2016 • £3

Features

Regulars

accu 2016

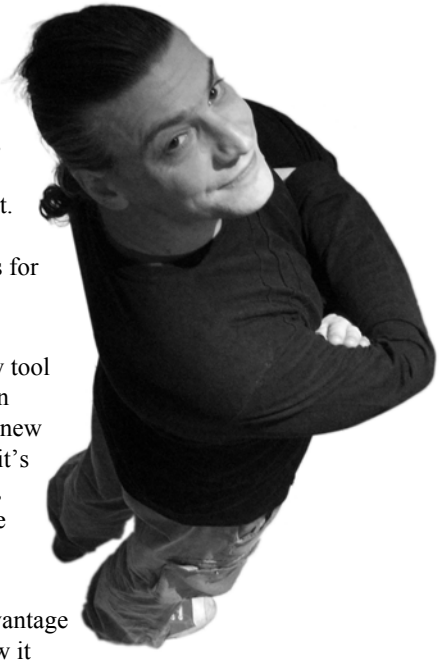
bristol
marriott
hotel
city
centre

pre-conference
tutorials
19 April

www.accu.org/
conference

{cvu}
Volume 27 Issue 6
January 2015
ISSN 1354-3164
www.accu.org

Choose your mask



How much are we defined by the tools we use? I've written about this a little before, and if you've been a *CVu* reader for a while you'll be familiar with much of Chris Oldwood's Toolbox. And it is a rich vein for examination. For those of us closely aligned with software development (and I guess that's probably most of you reading this), there is a huge variety of tools, and purposes to which they can be put. Partly that's because there are so many purposes that have tools, but also a great deal of choice among tools for one particular task.

The vastness of this choice can appear overwhelming sometimes, and for some things it seems there's a new tool every few minutes: new compiler version, new version control system, new editor, new bug tracking system, new web-server platform, new library to do /X/, new /X/...it's a full-time task just staying aware of all the new tools, never mind evaluating them and learning enough to be able to use them properly. Which is why we choose a small set of things, and concentrate our efforts there.

This runs the obvious risk of not being able to take advantage of some new tool (or /X/) because we don't even know it exists (which is why columns like Chris's are so valuable), or unaware of wider changes in technology or practice that might make our jobs or lives easier, or perhaps we might just find interesting. One of the fundamental things that I think many of us (I'm thinking of 'us' as software developers, but really it's more than that) have in common is a desire to learn new things. Whether we're pushing some existing tool or technology beyond the bounds we know, or pushing ourselves into new arenas and different technology altogether, we are a curious bunch (deliberately ambiguous phrasing alert!).

The tools we choose to use, which to learn, which to ignore, which to create, definitely define us in some way or another, but the ease with which we can re-define those tools, and thereby ourselves, means we are more than a stereotype. So, learn a new thing today. Then write an article on it, and send it to me!

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 28.1 1st February 2016
C Vu 28.2: 1st April 2016

Overload 132: 1st March 2016
Overload 133: 1st May 2016

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

Bug Hunting

Pete Goodliffe continues the hunt for software faults.

You can see a lot by just looking.
~ Yogi Berra

In the previous article we looked at the (somewhat obvious) reasons that an effective programmer has to be an effective debugger (or is that debuggist?) (or debugorisator?). And we began to look at strategies and tools that help us perform this task.

In this concluding part, we'll continue our journey though the useful strategies and tools that help us to find, and remove, those pesky varmints.

Invest in sharp tools

The are many tools that are worth getting accustomed to, including memory checkers like Electric Fence, and Swiss Army knife tools like Valgrind. These are worth learning about *now* rather than reaching for them at the last minute. If you know how to use a tool before you have a problem that demands it, you'll be far more effective.

Learning a range of tools will prevent you from cracking a nut with a pneumatic drill.

Of course, the tool of debugging champions is the *debugger*. This is the king of tools that allows you to break into the execution of a running program, step forward by a single instruction, or step in and out of functions. Other very handy facilities include the ability to watch variables for changes, set conditional breakpoints (e.g., `"break if x > y"`), and change variable values on the fly to quickly experiment with different code paths. Some advanced debuggers even allow you to step backward (now that's real voodoo).

Most IDEs come with a debugger built in, so you're never far from deploying a breakpoint. But you may find it worth investing in a higher quality alternative, don't rely on the first tool that falls to hand.

In some circles there is a real disdain for the debugger. *Real programmers don't need a debugger*. To some extent this is true; being overly reliant on the debugger is a bad thing. Single-stepping through code mindlessly can trick you into focusing on the micro level, rather than thinking about the macro, overall shape of the code.

But it's not a sign of weakness. Sometimes it's just far easier and quicker to pull out the big guns. Don't be afraid to use the right tool for the job.

> Learn how to use your debugger well. Then use it at the right times.

Remove code to exclude it from cause analysis

When you can reproduce a fault, consider removing everything that doesn't appear to contribute to the problem to help focus in on the offending lines of code. Disable other threads that *shouldn't* be involved. Remove subsections of code that do not look like they're related.

It's common to discover objects indirectly attached to the 'problem area' – for example, via a message bus or a notifier-listener mechanism. Physically disconnect this coupling (even if you're *convinced* it's benign). If you still reproduce the fault, you have proven your hunch about isolation, and have reduced the problem space.

Then consider removing, or skipping over, sections of code leading up to the error (as much as makes practical sense). Delete, or comment out blocks that don't appear to be involved.

Cleanliness prevents infection

Don't allow bugs to stay in your software for longer than necessary. Don't let them linger.

Don't dismiss niggling problems as *known issues*. This is a dangerous practice. It can lead to *broken window syndrome* [1], making it gradually feel normal and acceptable to have buggy behaviour. This lingering bad behaviour can mask the causes of other bugs you're hunting.

> Fix bugs as soon as you can. Don't let them pile up until you're stuck in a code cesspit.

One project I worked on was demoralisingly bad in this respect. When given a bug report to fix, before managing to reproduce the initial bug you'd encounter 10 different issues that all also needed to be fixed, and may (or may not) have contributed to the bug in question.

Oblique strategies

Sometimes you can bash your head against a gnarly problem for hours and get nowhere. Try an oblique strategy to avoid getting stuck in a debugging rut.

- Take a break

  It's important to learn when you should simply stop and walk away. A break can give you fresh perspective.

  This can help you to think more carefully. Rather than running headlong back into the code, take a break to consider the problem description and code structure.

  Go for a walk to force you to step away from the keyboard. (How many times have you had those 'eureka' moments in the shower? Or in the bathroom?! It happens to me all the time.)

- Explain it to someone else

  Describe the problem to someone else. Often when describing any problem (including a bug hunt) to another person, you instantly explain it to yourself and solve it.

  Failing another actual, live person, you can follow the *rubber duck strategy* described by Andrew Hunt and David Thomas [2].

  Talk to an inanimate object on your desk to explain the problem to yourself. It's only a problem if the rubber duck starts to talk back.

Don't rush away

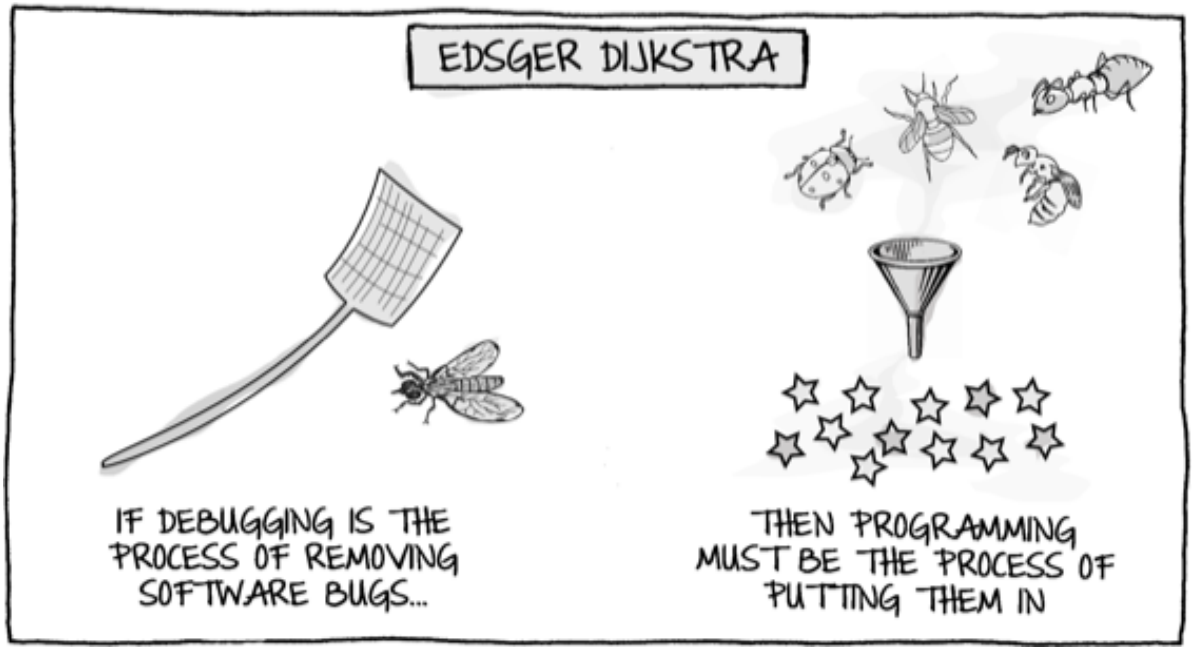
Once you find and fix a bug, don't rush mindlessly on. Stop for a moment and consider if there are other related problems lurking in that section of code. Perhaps the problem you've fixed is a pattern that repeats in other sections of the code. Is there further work that you could do to shore up the system with the knowledge you just gained?

Keep notes on which parts of the code harbour more faults. There are always hotspots. These hotspots are either the 20% of the code that 80% of users actually run, or a sign of ropey, badly written software.

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

When you have spent enough time gathering notes, it may be worth devoting time to those problem areas: perhaps a rewrite, a deep code review, or an extra unit test harness.



Non-reproducible bugs

Sometimes you discover a bug for which you can't easily form a set of reproduction steps. The bug defies logic and reason; it's not possible to determine the cause-and-effect. These nasty, intermittent bugs seem to be caused by *cosmic rays* rather than any direct user interaction. They take ages to track down, often because we never get a chance to see them on a development machine, or when running in a debugger.

How do we go about finding, and fixing, these fiends?

- Keep records of the factors that contribute to the fault. Over time you may spot a pattern that will help you identify the common causes.
- As you get more information, start to draw conclusions. Perhaps you can identify more data points to keep in the record.
- Consider adding more logging and assertions in beta or release builds to help gather information from the field.
- If it's a really pressing problem, set up a test farm to run long-running soak tests. If you can automate driving the system in a representative manner, then you can accelerate the hunting season.

There are a few things that are known to contribute to such unreliable bugs. You may find they provide hints for where to start investigating:

- Threaded code

  As threads entwine and interact in non-deterministic and hard-to-reproduce ways, they often contribute to freaky intermittent failure.

  Often this behaviour is very different when you pause the code in a debugger, so it is hard to observe forensically. Logging can also change the interaction of the threads and mask the problem. And non-optimised 'debug' builds of your software can perform rather differently from the 'release' builds.

  These are affectionately known as *Heisenbugs*, after the physicist Werner Heisenberg's 'observer effect' in quantum mechanics. The act of observing a system can alter its state.

- Network interaction

  Networks are, by definition, laggy and may drop or stall at any point in time. Most code presumes that all access to *local* storage works (because, most often, it does). This is careless, and will not scale to storage over a network, where failures and intermittent long load times are common.

- The variable speed of storage

  It's not just network latency that can cause this. Slow spinny disks, or database operations, may change the behaviour of your program, especially if you are balanced precariously on the edge of timeout thresholds.

- Memory corruption

  Oh, the humanity! When your aberrant code overwrites part of the stack or the heap, you can see a myriad of unreproducible strangenesses that are *very* hard to detect. Software archaeology is often the easiest route to diagnose these errors.

- Global variables/singletons

  Hardcoded communication points can be a clearing house for unpredictable behaviour. It can be impossible to reason about the correctness of your code, or predict what will happen, when anyone at any time can reach into a piece of global state and adjust it under your feet.

Conclusion

Debugging isn't easy. But it's our own fault. We wrote the bugs. Effective debugging is an essential skill for any programmer. ■

Questions

1. What tools or techniques do you fall back on when hunting a bug?
2. Are there other techniques you should try?
3. What was the trickiest bug you've ever had to find? What was the key thing that helped you find the cause?
4. Do you know other programmers who are better at finding and fixing bugs? What makes them more capable? How can you learn from them?
5. How can you close the gap between the introduction of a bug into a software system and the point at which it is observed, and the point at which it is fixed?

Notes and references

[1] Broken windows theory implies that keeping neighbourhoods in good condition prevents vandalism and crime. See http://en.wikipedia.org/wiki/Brokenwindowstheory

[2] Andrew Hunt and David Thomas, *The Pragmatic Programmer* (Boston: Addison Wesley, 1999).

Finding Text

Chris Oldwood hunts for the right tool to search text files.

It started with a fairly simple question: "Do you know of anything that can open a 400 MB text file?" Whilst being new to the team, I've been programming professionally long enough to know that this isn't the real question. I have my suspicions about what my fellow programmer really wants to do but I need to ask them what this huge text file is and, more importantly, why are they trying to open this file in the first place?

My hunch is correct – it was a log file. And the reason they are trying to open it is because they are on support and what to understand why something is failing or misbehaving. Hence the real problem is about how to efficiently view and manipulate large text-based log files. On Windows the lowest common denominator for this is **`more`**, if using the command line, and Notepad for the GUI-oriented. The latter is essentially just a wrapper around the Windows edit-box control and was never designed for handling large chunks of text.

After quickly reeling off close to a dozen tools that I could use to view and process log files it got me thinking about the wider question: what tools might I use to solve the more general problem of 'finding text', and what conditions or constraints would cause me to choose one over another? After all, this task is one that we programmers probably perform many, many times a day for different reasons.

The criteria for this list are pedagogically loose and cover the need to match prose and structured text, both programmatically and also manually. When diagnosing a problem we often don't know what the pattern is at that point and so we have to rely on our built-in pattern recognition system to seek out some semblance of order from the chaos. At which point we may switch or combine approaches to delve in further. In some cases we may not even be consciously looking for it, but the tool makes it apparent and leads us to go looking for more anomalies.

This list of tools is not, and cannot be comprehensive because the very problem itself is being solved again and again. It is also not presented it any particular order because the tool might be used in a variety of contexts depending on the conditions and constraints in play. Many of these tools do have very similar variations though and are pretty much interchangeable so are discussed together.

FIND / FINDSTR

Windows comes with not one, but two command line tools for finding text within files (or the standard input). If you want to know why there are two similar tools you can read Raymond Chen's blog post [1], but in short they come from the two different Windows lineages – 9x and NT.

I rarely use **`FIND`**, except by accident, as it has the same name as one of the classic UNIX command line tools which does something different. Luckily the more recent Gnu on Windows (GoW) distributions [2] have taken the pragmatic approach of renaming its **`FIND`** tool to **`GFIND`** to avoid surprises when running scripts where the PATH order differs.

Even without this wrinkle there is little reason to use **`FIND`** over **`FINDSTR`** as the latter has some support for regular expressions. Sadly it also has this weird behaviour of treating a string with spaces as a list of words to match instead of treating the entire string literally.

Given its general non-standard behaviour, with respect to **`GREP`** (and its cousins), it might seem somewhat useless. But it has two things going for it – it's installed by default and is fast in comparison to some other similar command line tools I've used.

There was a time when production servers where tightly locked down and so this was your only option. Operations teams seem to be a little more open these days to a wider variety of tools, no doubt in part due to the use of VMs to isolate applications, and therefore teams, from each other.

GREP / EGREP / FGREP

The natural alternative to the Windows **`FIND`**/**`FINDSTR`** combination is the UNIX equivalent **`GREP`**. It has two counterparts **`EGREP`** and **`FGREP`** which are really just short-hands for **`GREP -E`** and **`GREP -F`** respectively. In fact the **`--help`** switch for the short-hands warns you they are deprecated forms. Sadly my muscle memory keeps kicking in as I've been using ports of them since my days working on DOS.

For the record the difference is that **`EGREP`** (**`GREP -E`**) enables an extended regular expression syntax whilst **`FGREP`** (**`GREP -F`**) treats the strings literally and so (I guess) is faster. I haven't timed them recently and suspect that there is little in it these days.

For a long time I used the Win32 ports distributed as UnxUtils, but that went stale years ago and GoW (Gnu on Windows) has replaced it as my port of choice as it has no extra dependencies, such as Cygwin. As such it makes it easy to include these in a diagnostics package or just **`XCOPY`** them about. What makes Operations feel uneasy is software that needs 'installing' whereas being able to run something directly from a remote file share usually won't raise their ire.

With the introduction of the Chocolately package manager [3] on Windows, this toolset is pretty much one of the first things I install, and the renaming of **`FIND`** to **`GFIND`** now makes it safer to install on a server too without fear of silently breaking something else.

Like many others this is the tool I probably reach for first, command-line wise.

AWK & SED

If the task is to simply find some text then **`GREP`** pretty much does the trick, but often I'm looking to do a little bit more. I might need to do a little parsing, such as summing numbers contained within it or transforming it slightly to reduce the noise. In these cases I'm once again looking to the UNIX toolset classics, this time **`AWK`** and **`SED`**.

I had forgotten about them for many years whilst I was heavily into doing front-end work, but as I switched to the back-end again I found myself doing a lot more text processing. In fact I wrote about my rediscovery in these very pages a few years ago [4].

Whilst I could use **`SED`** more often as a replacement for **`GREP`** I keep forgetting the differences in the regular expression syntax (there are many things it doesn't support) and so I find myself wasting time trying to debug the regex only to discover I'm using an **`EGREP`** supported construct. Hence I usually carry around a copy of the O'Reilly pocket reference books on regular expressions and **`SED`** & **`AWK`**, mostly for the tables comparing support across **`SED`**, **`GREP`** and **`AWK`**.

I could use **`AWK`** far more than I do for basic text matching because it's pretty quick, but I forget and only remember when I suddenly realize I need the power of its formatting options, not its matching ones.

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

One particularly useful feature of **SED** and **AWK** is their support for address ranges. If you have some pretty-printed XML or JSON (i.e. one tag or key/value pair per line) then you can match parts of the document using address ranges. For example, before I discovered JQ for querying JSON [5], I generated some simple release notes by extracting the card number and title from a Trello board exported as JSON via a script I wrote that invoked **SED** and **AWK**.

MORE / HEAD

Earlier I mentioned that I often rely heavily on my own human pattern matching skills to home in something hopefully buried within the chaos. If I'm already in the process of building a pipeline to do some matching I might just want to pass my eye over the content by paging through it and seeing if any patterns emerge in the flicker. Hence **MORE** (or **HEAD** if I remember it's there) are useful ways to page text for manual scanning.

I generally shy away from using **HEAD** though as the versions in the UnxUtils and GoW distributions often go bonkers complaining about a broken pipe and it quickly floods the screen with an error. I'm sure I'm just doing something wrong but it hasn't bothered me enough to discover what it is. That's the great thing about have so many choices, you just work around the limitations in one tool by picking another similar one.

to help the mind unearth patterns in text, again especially when dealing with analysing log files, apply a dash of colour

PowerShell

In more recent years as I've started to use PowerShell more and more for scripting, I naturally find myself becoming more comfortable using the built-in features of the language to create even simple text processing pipelines as well as its more powerful object base ones.

In particular the language comes with some useful cmdlets out-of-the-box for handling XML and CSV files in a more structured way. For example you can import a CSV file and name the columns (if it's not already done via a header row) which then allows you to query the data using the more natural column names instead of, say, using CUT and having to refer to them numerically. This really aids readability in wrapper scripts too [6].

Whilst PowerShell comes with great flexibility by using .Net for its underpinnings, this also comes at a cost. The performance of parsing textual log files is considerably worse than with native code, such as **AWK**. I once needed to do some analysis that involved parsing many multi-megabyte log files on a remote share. I started out using PowerShell but eventually discovered it was taking a couple of minutes just to read a file. So I switched to **AWK** instead which managed to read and parse the same file in only 8 seconds. This was on PowerShell v2 and so more recent versions of .Net and PowerShell may well have closed the gap.

LogParser

Another very old, free tool that I've found useful for parsing log files because of its performance is Microsoft's LogParser [7]. Originally written to parse IIS log files it grew the ability to read various other text and binary format files which, like the Import-Csv cmdlet in PowerShell, can give the data columns names. These can then be used within LogParser's SQL-like language to create some pretty powerful queries.

BareTail

One of the ways I've found to help the mind unearth patterns in text, again especially when dealing with analysing log files, is to apply a dash of colour to certain lines and words. Just as I use syntax highlighting to make source code a little easier to read, I apply the same principle to other kinds of files. For example I'll highlight error messages in red and warnings in yellow. If there are regular lines that are usually of little interest, such as a server heartbeat message, I'll colour it in light grey so that it blends into the white background to make the more significant behaviour (in black) stand-out.

The first decent Windows tool I found that did this was a commercial tool (with a free cut-down version) called BareTail. Naturally others have sprung up in the meantime, like LogExpert, which are free.

What really attracted me to shell out for the full version was its ability to tail a file and at the bottom have a real-time **GREP** running over the same file to filter out interesting events. This was a feature my previous team had built into its own custom log viewer years before and so was a most welcome discovery.

BareGrep

The sister tool to BareTail is called BareGrep, which is a GUI based version of the old classic described earlier. It too has the same highlighting support and also has a **TAIL** like view at the bottom which provides additional context around the lines you match with the pattern in the main view.

The two other features that made it a worthy addition to my toolbox were its ability to use regular expressions on the filename matching (rather than the usual simpler file globbing provided by the Windows FindFirstFile API), and its support for naming and saving patterns. On support I often find myself using the same regex patterns again and again to pick out certain interesting events at the start of an investigation.

Notepad

This article began by explaining what the alternatives are on Windows to the simplistic Notepad, but that doesn't mean it's not still useful. Aside from WordPad, it's the only GUI-based viewer installed by default which might be significant in a locked down environment. Even so it can still be handy for smaller stuff, like looking at .config files.

In a way its naivety is also one of its few useful traits. By only handling Windows line endings and having an insane tab width setting of 8 means that any screwy formatting shows up pretty quickly and acts as a gentle reminder to check everyone's on the same page editor-configuration wise.

Notepad++

The original Notepad is very much a tool of last resort and so I'll try to install something a little more powerful as a replacement for day-to-day, non-IDE based text editing jobs, such as writing mark-up. It's quick to open and provides all the usual features you'd expect from a plug-in enhanced text editor. It could easily be Atom, Sublime Text or any one of a number of decent editors out there.

For me the decision to use the command line or a GUI based tool when searching for text depends on how much context I need when I find what I'm looking for and also whether I'm going to edit it afterwards. When searching log files, it's ultimately a read-only affair with perhaps some statistical output. In contrast a document probably means I'm going to select some text and paste it elsewhere or even edit the prose in-situ which demands at least a spell checker.

The other factor is often whether I've navigated to the data via a command prompt or the Windows Explorer in which case a right-click is easier than opening a prompt at the folder and typing a command. That said if the file type association is already registered it's just as easy to go the other route and open it in a GUI tool from the command line. And sometimes the choice of tool is totally arbitrary and depends on whatever I've not used in a while and feel I need to remind myself about.

Visual Studio

Anyone who has ever double-clicked the Visual Studio icon by accident or forgotten to register a more lightweight choice of editor for the file association will curse their mistake as it takes an eternity to start. But if it's already running, and being an IDE means it's quite likely for that to be the case, it's just as easy to reuse it as spawn another text editor.

In Vivo, In Vitro, In Silico

Frances Buontempo examines the idea of software vivisection.

Some people get unit testing and some people don't. The reasons vary, usually based on a mixture of previous experience, lack of experience, fear of the unknown or joy at a safer quicker way of developing. One specific doubt crops up from time to time. It comes in the form of "If I test small bits, i.e. units, whatever that means, it proves nothing. I need to test the whole thing or small parts of the whole thing live."

My PhD was in toxicity prediction, which involves testing if something will be toxic or not. You can test a chemical *in vivo* – administer it to several animals in varying doses. You sit back and wait till half of them die or show toxicity symptoms and record the doses. This gives you the Lx50 – for example the LD50 is the lethal dose that kills 50% of the animals. Notice I said you can do this. You can also test the chemical on a set of cells in a test tube or petri dish – *in vitro* (in glass). Again you can find the dose which affects 50% of the specimens. I personally find this less upsetting, but I want to focus on parallels with testing code here. Finally, given all this data the previous tests have generated, you can analyse the data, probably on a computer, perhaps finding chemical structure to activity relationships – SAR, or quantitative SARS i.e. QSARs. These are referred to as *in silico* – for obvious reasons. Some *in silico* experiments will just find clusters of similar chemicals, which can either alert you to groups that might need more detailed toxicity testing, or even guide drug discovery by steering clear of molecules, say containing benzene rings which can be carcinogenic, saving time and money if you are trying to invent a drug that cures cancer. The value of testing on a computer outside a live organism should be clear. It can save time, money and even lives.

If we keep this in mind while considering testing a software system, rather than a biological system, we should be able to see some parallels. It is possible to test a live system – maybe on beta rather than 'TIP' (Test in production). This can be a good thing. However, it might save time and money, and though maybe not lives, certainly headaches, to test parts of the live system in a sandboxed environment, analogous to *in vitro*. Running an end-to-end test against a test database instance with data in a specific state might count. Pushing the analogy further, you could even test small parts of the system, say units, whatever they are, *in silico*. Just try this small part away from the live system in a computer. This is worthwhile. It will be quicker, as toxicity-*in-silico* experiments are quicker – they tend to take hours rather than days. This is a good thing. Of course, you won't know exactly what will happen in a full live system, but you can catch problems earlier, before killing something. This is a Good Thing.

Other industries also test things in units – I could put together a car or a computer hit the on switch and see if it works. However, I am given to believe that the components are tested thoroughly ***before*** the full system is built. If I build a PC and it doesn't work I will then have to go through one part at a time and check. If someone tests the parts first, this will ensure I haven't put a dodgy power block in the whole thing. Testing small parts, preferably before testing the whole system, is a Good Thing.

I don't believe this short observation will change anyone's minds. But I hope it will give pause for thought to those who think only testing from end to end matters, and testing *in silico* is a waste of time. ■

FRANCES BUONTEMPO

Frances Buontempo 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.

Finding Text (continued)

As of Visual Studio 2013, they have also replaced the byzantine regular expression syntax with the more standard .Net one which makes finding text with regexes way more palatable. And the new, cross platform Visual Studio Code editor is looking like this mistake will be far less costly in future.

Vendor specific tools

Although most content is becoming more available in simpler text formats so that the choice of tooling is much wider and freer there is still plenty of it stored in custom binary formats like old MS Word documents. The rise of wikis and the various flavours of mark-up have certainly gained in popularity but the enterprise is often locked into vendors through these bespoke formats and so for completeness these need to be accounted for, but are on the decline.

Google

The discussion thus far has largely been about finding text in files on my machine or the intranet. But every day like so many other people I spend plenty of time looking things up on the Internet and for that, naturally, I'll use one of the major search engines.

Despite them selling appliances though for well over a decade that promises to bring the power of an Internet search to the enterprise this still does not appear to have happened and finding anything on an intranet still appears to be a fruitless exercise.

Summary

Searching text, whether it be source code, prose, log or data files is a bread-and-butter activity for programmers. What we do with it when we've found it adds another dimension to the type of tools we use because it may not just be plain text we're lifting but we might want the formatting too. Throw performance and differing query languages into the mix and it's no wonder that we have to keep our hands on such a varied array of tools. ■

References

[1] https://blogs.msdn.microsoft.com/oldnewthing/20121128-00/?p=5963
[2] https://github.com/bmatzelle/gow/wiki
[3] https://chocolatey.org/
[4] http://www.chrisoldwood.com/articles/reacquainting-myself-with-sed-and-awk.html
[5] https://stedolan.github.io/jq/
[6] http://www.chrisoldwood.com/articles/in-the-toolbox-wrapper-scripts.html
[7] https://technet.microsoft.com/en-gb/scriptcenter/dd919274.aspx

"HTTPS Everywhere" considered harmful

Silas S. Brown considers an unintended cost of security.

You're not rich enough to look at the Internet. Google has banned you. That's an exaggeration (well I had to catch your attention somehow, didn't I?) but it may reflect what some feel as they cope with the collateral damage caused by Google's 'HTTPS everywhere' drive.

With the overheads of certificate validation, loading a Web page over 'secure' HTTPS can cost the best part of a megabyte in data transfers, even if the underlying page size is only a few dozen kilobytes. Extra data transfers slow things down, and if your Internet connection is in any way 'wobbly' (such as if you're using mobile data, especially if it's 2G-only which runs at a lower priority than voice traffic, or if you're stuck with a particularly poor-quality ADSL service) then it might make the difference between being able to load the site and not being able to load the site. (With HTTPS you can't continue where you left off when your link comes back up.) Add to this the cost considerations if you have a limited data plan, not to mention the battery life and environmental considerations associated with the extra processing needed, and you can see why it makes sense not to use HTTPS except for pages that really need to be 'secure', such as a logged-in session after a user has provided a password or other credentials.

I agree it is a mistake to use HTTPS for the login itself but then issue an insecure cookie to identify the user to HTTP pages, as this can be stolen by anyone sharing the same IP address, as demonstrated by FireCrack and other 'profile-grabbing' attacks over shared WiFi. HTTPS should be used for the whole session after login. But that doesn't automatically mean HTTPS should be used for normal public web pages that you don't even log in to see.

You might have noticed an increasing number of websites such as news sites using HTTPS only. Before I realised the more likely explanation, I thought they were merely echoing WikiMedia's decision to go all-HTTPS in the light of Edward Snowden's publicity about NSA data monitoring, so that it's harder for authorities to determine which of their pages are being accessed (which in practice might mean an authority bans an entire site rather than specific pages of it, but I won't go into that). However, it did strike me as a bit odd that an average news site would go out of their way to protect their readers' choice of articles from being monitored, particularly given the added costs to them of running HTTPS servers.

And then I saw Google's 2014 announcement [1] that the use of HTTPS is now going to be used as a ranking signal. Because Google engineers like the idea of having HTTPS everywhere, they are 'encouraging' other websites to move to it by rewarding them with higher rankings in Google search results. Ah, so *that's* why everyone seems to be jumping on the bandwagon.

Some have tried to make business decisions like 'the extra cost of hosting HTTPS is worth a 1% increase in ranking'. But how can you be so sure? What exactly *is* a 1% increase anyway? Few people understand the exact relationship between a given amount of change in one component of the ranking algorithm and the actual effect on one's listing position, let alone on how much resulting traffic one gets. And let's not even try to go into considerations of how much of that traffic is actually profitable to a business, or how much of it is from people who are interested enough to stay and read what you have to say instead of deciding within seconds that they've been misdirected by the search engine and surfing off somewhere else.

As the situation is so ill-understood, it seems to me that people are jumping to the whims of Google engineers in at best a semi-rational fear due to Google's vague threats of reprisals. (It's even more vague when they say the weight of the HTTPS ranking signal might be increased by an unspecified amount at an unspecified time in the future.) That reminds me of the way things are supposed to work in certain countries of this world whom I daren't name here or they might ban ACCU in retaliation. (Perhaps we'd better not make the full text of this article visible to Google: they might find a way to down-rank us for saying something negative about them. See the similarity?)

HTTPS when you have to log in might be good, and even for public pages it might be arguably a good thing to offer the user a chance of accessing the site over either HTTPS or HTTP according to their wishes, but forcing users to use HTTPS everywhere for public read-only pages (without even giving them an HTTP fall-back option) just slows things down for everybody, consumes more resources, and might lose visitors (like me when I'm on my mobile in a patchy-signal area).

If Google rewards HTTPS too much then they will essentially be giving a boost to corporate sites that are well-heeled enough to afford it, at the expense of academic and non-profit organisations that often aren't, not to mention hobbyists and staff who do not have their own domains but are at the mercy of whatever server their user pages are stored on. While a reward of HTTPS might help to down-rank some (but not all) 'spammy' search results, it's almost certainly a case of 'throwing the baby out with the bath-water', since a lot of good-quality online information is not provided by well-heeled corporations.

If Google really does end up boosting HTTPS so much that sites which don't have it end up being seriously down-ranked for no good reason, even though they have better information, then I hope that causes the population at large to realise that Google search is not as good as it used to be and to try some other search company instead. That's what Google would arguably deserve if they tweak this knob too much. So I wouldn't be too worried about this in the long term: if Google goes crazy then it will no longer be cause for concern.

Meanwhile, perhaps the best way to make Google engineers realise the error of their ways is for as many people as possible who have good worthwhile information online to stand firm with the HTTP protocol and do not kowtow to Google's whim of having it on HTTPS. We might be able to make it obvious to them that too much of a boost to HTTPS would down-rank good stuff. Since Google's engineers are particularly interested in technical material, I would highly recommend anyone who has good technical material online to stand firm for HTTP so as to show Google they can't get away with writing that off too much. Each of our sites may be small, but if enough of us do it then they might see us. That is, as long as Google engineers don't end up ignoring all information that's not on the likes of Stack Overflow.

As for me, I have tweaked my Web Adjuster [2] so it can proxy HTTPS onto plain HTTP. With suitably frightening 'you do realise this trashes your security' warnings, this could be a 'lifesaver' in some mobile situations when a site is using HTTPS unnecessarily. And if anybody ends up not finding my code just because I don't have an HTTPS site, I refuse to make that my problem: they should be better at finding things. I'm not giving in to vague threats about HTTPS just yet! ■

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

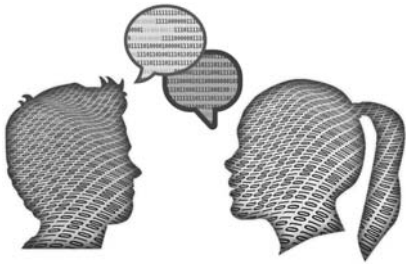
References

[1] https://googleonlinesecurity.blogspot.co.uk/2014/08/https-as-ranking-signal_6.html

[2] http://people.ds.cam.ac.uk/ssb22/adjuster

Code Critique Competition 97

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



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. (We will remove email addresses!)

Last issue's code

I have written some code to read in a CSV file and handle quoted strings but I seem to get an extra row read at the end, not sure why.

If I make a file consisting of one line:

--- Book1.csv ---

word,a simple phrase,"we can, if we want, embed commas"

--- ends ---

I get this output from processing the file:

```
Rows: 2
Cells: 3
  word
  a simple phrase
  "we can, if we want, embed commas"
Cells: 1
```

What have I done wrong?

The code is in Listing 1.

Listing 1

```
// Reading CSV with quoted strings.
#include <iostream>
#include <string>
#include <vector>

typedef std::string cell;
typedef std::vector<cell> row;
typedef std::vector<row> table;

table readTable()
{
  char ch;
  table table; // the table
  row *row = 0; // the row
  cell *cell = 0; // the cell
  char quoting = '\0';
  while (!std::cin.eof())
  {
    char ch = std::cin.get();
    switch (ch)
    {
    case '\n':
    case ',':
      if (!quoting) {
        cell = 0;
        if (ch == '\n') {
          row = 0;
        }
        break;
    case '\'':
    case '"':
      if (quoting == ch) {
        quoting = '\0';
      }
      else if (!cell) {
        quoting = ch;
      }
    }
    default:
      if (!row) {
        table.push_back({});
        row = &table.back();
      }
      if (!cell) {
        row->push_back({});
        cell = &row->back();
      }
      cell->push_back(ch);
      break;
    }
  }
  return table;
}

int main()
{
  table t = readTable();

  std::cout << "Rows: " << t.size() << "\n";
  for (int r = 0; r != t.size(); ++r) {
    std::cout << "Cells: "
      << t[r].size() << "\n";
    for (int c = 0; c != t[r].size(); ++c)
    {
      std::cout << "  " << t[r][c] << "\n";
    }
  }
}
```

Listing 1 (cont'd)

Critiques

James Holland <James.Holland@babcockinternational.com>

When I compiled the student's code I received three warnings and no errors. It is always a good idea to produce code that compiles with no warnings and so I set about seeing what the compiler was complaining about. A brief investigation showed the cause of the problems. There is a redundant declaration of **ch** at the start of **readTable()**. The declaration should be deleted. (The **ch** that is used by the program is the one declared at the start of the **while** loop within **readTable()**.)

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 other warnings concern the nested for loops within `main()`. The problem is that `size()` returns an unsigned type that is being compared with a signed type. Mixing types in this way can produce unexpected results and so is best avoided. Changing the type of variables `r` and `c` to `size_t` will keep the compiler happy.

I see the student talks about a file named 'Book1.csv' and yet the supplied program receives input from the console. I assume the student was experimenting in an attempt to debug the code. [**Ed:** Apologies that I failed to make that clear – the student was using command-line redirection from the file.] I found it more convenient to input data from the file and so I modified the code accordingly but keeping the student's use of the `eof()` function. Running the amended code did not result in an extra blank line reported by the student. I suspect the student's Book1.csv file consisted of a carriage return on the end of the first line and it is this that caused of the extra blank row in `table`. In any case, the student's use of `eof()` is incorrect.

The student used a `while` loop to read characters from the file. The body of the loop will repeatedly execute until the end of file marker is set. The marker is set only when an attempt has been made to read a character from beyond the end of the file. The student's code reads characters at the start of the loop body and this is where the problem lies. Within the loop, characters are read up to and including the last character of the file. At this point the end of file marker has not been set and so the loop is executed one more time. An attempt is made to read another character and the end of file marker is set. Unfortunately it is set too late, the loop body has already started to execute. Instead of returning a character from the file, `get()` returns `EOF`. The program then processes this value (`-1` on my machine) in the same way it would a character from the file. This is clearly not what the student intended. To cure this problem, the body of the loop must not be allowed to execute after an attempt has been made to read beyond the last character of the file. Fortunately this is easy to arrange.

What is required is for the `while` loop to attempt to read a character from the file and then check for the end of file marker being set all within the predicate of the loop. The code shown below achieves the required result.

```
char ch;
while (file.get(ch))
{
  …
}
```

In fact, the `file.get()` returns the file stream and so the predicate tests, with an implicit conversion to `bool`, whether the stream is in a good state. Attempting to read one character beyond the end of the file renders the stream not in a good state and so the body of the loop will not be executed, as required.

The program will now work as expected. However, one cannot help but notice that the function `readTable()` is quite involved. It consists of `switch` statement where one case falls through to the next, there are `case` labels within `if` statements. In fact, there are a total of six `if` statements within the `switch` statement. All of this wrapped up in a `while` loop. As far as I can determine, the code works correctly. It is, however, very difficult to reason about. It is just far too complicated. There has to be another way. Fortunately there is.

The function `readTable()` is mostly concerned with searching for patterns within text strings. Whenever searching for patterns is mentioned, regular expressions should come to mind. C++11 has a regular expression library that is based on the Boost library. Using regular expressions greatly simplifies `readTable()` as is shown below.

```
table readTable()
{
  boost::regex regular_expression(
    "('.*')|(\".*\")|[^, ]+( *[^, ]+)*");
  boost::sregex_iterator end;

  table table;
  std::ifstream file("Book1.csv");
  for (std::string record;
       getline(file, record);)
  {
    boost::sregex_iterator position(
      record.cbegin(), record.cend(),
      regular_expression);
    row row;
    for (; position != end; ++position)
    {
      row.push_back(position->str());
    }
    table.push_back(row);
  }
  return table;
}
```

The first statements of the replacement `readTable()` contains the regular expression that defines what is being searched for and consists of three parts that are separated by the 'or' symbol `|`. The first part, `('.*')`, defines a search for any number of characters starting with a single quote mark and ending with a single quote mark. The next part of the expression defines a search of any number of characters starting with a double quote mark and ending with a double quote mark. The third part of the expression is a little more complicated. It defines a search for a string starting with any character that is not a comma and is not a space. The string may then contain any number of spaces and any number of characters that are not a comma and not a space. This, in effect, defines a word of a phrase. The expression then goes on to define that a phrase can contain any number of words.

It has to be said that using the regular expression grammar effectively requires a little practice. Once mastered, however, regular expressions are a concise way of defining search patterns that do not require the programmer to construct complicated and hard to fathom state machines.

The rest of `readTable()` is fairly conventional. It reads the file a line at a time and then, using the regular expression, searches for the words and phrases within the line. `readTable()` then assembles the words and phrases ready to be pushed onto `table`.

Commentary

There were a number of problems with the supplied code including the 'presenting problem' of the extra empty row. As James correctly points out, this one is caused by the erroneous use of the `eof()` method in the program. In my experience, code that uses an explicit call to `eof()` is often incorrect. It is generally easier to use the implicit conversion to `bool` as this seems to naturally help to create better handling of end of input.

I do not recommend the naming convention used for `table`, `row` and `cell` in this program where the variables have the same name as their type as this can lead to hard-to-understand code. Additionally, since the meaning of the symbol depends on the scope, moving code around during maintenance or refactoring can lead to obscure breakages.

People are often surprised to come across the interactions between `switch` and `if` demonstrated in this program. Again, I would not recommend this style unless there are some overwhelming reasons why it needs to be used (such as a hard performance requirement if this technique can be shown to be measurably faster than a more structured alternative). The reason is the additional complexity that intermingling `case` statements and `if` statements leads to. It becomes hard to reason correctly about the code as most programmers in C or C++ have a natural tendency to unconsciously treat each `case` statement separately and so fail to fully take into consideration the 'dangling' portion of the earlier `if` statements.

While I was expecting to receive critiques of the code that attempted to make the structure of the state machine clearer, I think that James' approach of using a pre-written standard solution has a lot to recommend it. There may be an initial conceptual barrier of understanding the syntax of the regular expression being used – but since this is using a standard regular expression grammar this will be a transferable skill! Once the regular expression is checked, the rest of the code is simple enough to be clearly correct.

The main downside of using regular expression matching is likely to be a performance one since it is likely to be slower than a tailored state machine solution. As always though, even when performance is relevant, it is important to measure rather than rely on a guess. I did a naive measurement comparing the original code and James' solution; the regular expression code only took 4.4 seconds to read a quarter of a million rows which, while slower than the original code at 1.2 seconds, is likely to be fast enough in many cases.

The winner of CC 96

There was only one critique this time – perhaps the unstructured use of control flow put people off!

As I mentioned in my commentary, I liked James' approach of raising the abstraction level of the problem and using the power of the regular expressions handling in the standard C++ library to solve the problem.

His solution is written in terms of boost, but you can simply replace **boost** with **std** for compilers supporting C++11 or above.

So, despite being the only entrant, I consider that James deserves the award of the prize for this issue's critique.

Listing 2

```
#include <utility>
#include <vector>

#pragma once

// An unordered collection of values
template <typename T>
class Values
{
public:
  void add(T t);
  bool remove(T t);
  std::vector<T> const & values() const
  { return v; }
private:
  std::vector<T> v;
};

// Add a new item
template <typename T>
void Values<T>::add(T t)
{
   v.push_back(t);
}

// Remove an item
// Returns true if removed, false if not present
template <typename T>
bool Values<T>::remove(T t)
{
  bool found(false);
  for (auto i = 0; i != v.size(); ++i)
  {
    if (v[i] == t)
    {
      v[i] = std::move(v.back());
      v.pop_back();
      found = true;
    }
  }
  return found;
}
```

Code critique 97

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

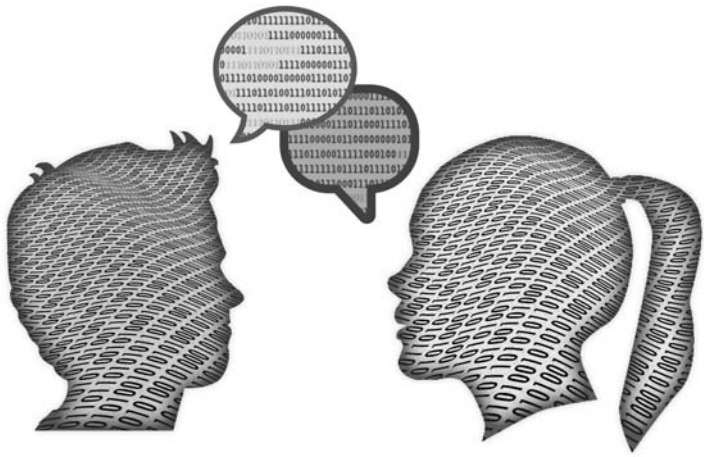
> I have a simple template class that holds a collection of values but sometimes code using the class crashes. I've written a simple test for the class, which works, but I do get a warning about a signed/unsigned mismatch on the **for** loop. I thought using **auto** would stop that. Is that anything to do with the crash? I've commented out all the other methods apart from **add** and **remove**.

The code is in Listing 2 (`values.h`) and Listing 3 (`test.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://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 3

```
#include <iostream>
#include "values.h"

void test()
{
  Values<int> vi;
  for (int i = 0; i != 10; ++i)
  {
    vi.add(i);
  }

  if (!vi.remove(1))
  {
     std::cout << "Can't remove 1\n";
  }

  if (vi.remove(1))
  {
     std::cout << "Can remove 1 twice\n";
  }

  if (!vi.remove(9))
  {
    std::cout << "Can't remove 9\n";
  }

  std::cout << "size: " << vi.values().size()
    << std::endl;
}

int main()
{
  test();
}
```



View from the Chair

Alan Lenton
chair@accu.org

If you haven't got the dates, Wednesday 20th April to Saturday 23 April in your diary, then put them in now. They're the dates of the 2016 ACCU Conference, a must for all serious programmers. This year, the conference is again in Bristol, so that's also the venue of the AGM – on the Saturday lunchtime of the Conference. In a way it's an interesting reversal of what originally happened. When we first started, we used to have a couple of short talks before the AGM, Now the talks have blossomed out into a full blown conference and we wedge the AGM into the Saturday lunchtime!

Our existing local groups are, as usual, organising regular meetings, but we are still finding it difficult to get new groups off the ground and boost the smaller groups. It's perhaps significant that our most successful local groups – London, Oxford, and Bristol (travelling east to west – no other significance!) are where our biggest concentrations of members are. The question is, of course, do they organise more meetings because they are larger and have a larger pool of people to draw from, or are they larger because they hold more meeting and attract more people?

Houston – we have a chicken and egg problem!

If anyone has any bright ideas about how to solve this problem, preferably without having to cough up enormous sums of money, then please let me (or any other member of the committee) know – alan@ibgames.com, or at conference, in the bar in the evenings, or the ACCU session on Saturday morning.

Because of the publication lead times, I'm writing this just before Christmas, but by the time you read this, the full details of the conference talks and keynotes should be available on the ACCU web site. In the next issue of *CVu* I should have some details of the issues for discussion at the AGM, so until then I'll wish you a rather belated prosperous 2016, and I look forward to seeing you all at the ACCU Conference.

Bookcase

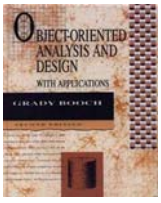
The latest book review.

If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.
Astrid Byro (astrid.byro@gmail.com)

Object-Oriented Analysis and Design with Applications (2nd Edition)



By Grady Booch. 589 Pages. Published by Addison-Wesley. ISBN 0-8053-5340-2

Reviewed by Ian Bruntlett

I decided to study this book to improve my OO skills. It is reasonably well-written and, as a hardback, is sturdy enough to see a lot of use. It is sufficiently in-depth enough to require deep study and multiple readings.

The inside front and back covers act as a convenient quick reference regarding the process of Object-Oriented Development process and Booch's notation (probably superseded by the UML notation).

The preface discusses the evolution of this book, its goals, audience and structure. I am reviewing this book from the perspective of a Software Developer wanting to refurbish my OOA, OOD and OOP skills.

This book is divided into three main sections – Concepts, the Method, Applications/Case studies. Also, there is an appendix covering different OOP languages as well as Notes, Glossary and a classified Bibliography and Index.

Section One – Concepts – has 4 chapters: Complexity, The Object Model, Classes and Objects and Classification. Indeed, I was so determined to master this section that I spent a lot of time reading it, re-reading it, making notes by summarising the text and re-reading the notes. I made these notes in a Journal and there were 161 pages of notes. In addition, I used marker pens in the book itself.

Chapter 1: Complexity (covers Inherent complexity of software, Structure of Complex Systems, Bringing Order to Chaos and On Designing Complex Systems). This discusses the inevitability of complexity in I.T. systems and the role of OO in dealing with that complexity.

Chapter 2: The Object model discusses the evolution of the object model in software systems. From early programmes that dealt with mathematical things through to current day OO languages. It discusses OOA (Object Oriented Analysis), OOD (Object Oriented Design) and OOP (Object Oriented Programming). It discusses four major elements of the Object Model (Abstraction, Encapsulation, Modularity and Hierarchy) and three minor elements (Type system, Concurrency, Persistence).

Chapter 3: Classes and Objects (The Nature of an Object – an object has State, Behaviour and Identity, Relationships among Objects, The Nature of a Class, Relationships Among Classes, The Interplay of Classes and Objects and On Building Quality Classes and Objects). By the time you finish this chapter you should be confident when dealing with Data Abstraction.

Chapter 4: Classification (The Importance of Proper Classification, Identifying Classes and Objects, Key Abstractions and Mechanisms) builds on the previous 3 chapters to produce confidence with inheritance (generalisation-specialisation mechanisms).

Section Two – The Method – has 3 chapters. The first two (The Notation, The Process) are largely superseded by UML).

The third chapter, Pragmatics covers considerations beyond programming – in particular Management and Planning, Staffing, Release Management, Reuse, Quality Assurance and Metrics, Documentation, Tools, Benefits and Risks of OOD. This chapter is easier to read than most and is likely to be relevant for a very long time.

Section Three – Applications – has five chapters, each of which is a case study of an application of the OO process. Most of its pseudo-code is in C++ and some of the examples are (currently) difficult to understand. However, I'll be improving my C++ and revisiting this section sometime in the future. The case studies involved are Weather Monitoring Station, Frameworks (Foundation Class Library), Inventory Tracking, A.I. (Cryptanalysis) and Traffic Management.

Conclusion. I really liked this book and find it difficult to review because there are so many things I find interesting in this book that I'd like to share in this review.