the magazine of the accu

www.accu.org

{cvu}

Volume 28 • Issue 5 • November 2016 • £3

Features

JET
BRAINS

A Power Language Needs Power Tools

Smart editor with full language support
Support for C++03/C++11, Boost and libc++, C++ templates and macros.

Reliable refactorings
Rename, Extract Function / Constant / Variable, Change Signature, & more

Code generation and navigation
Generate menu, Find context usages, Go to Symbol, and more

Profound code analysis
On-the-fly analysis with Quick-fixes & dozens of smart checks

GET A C++ DEVELOPMENT TOOL THAT YOU DESERVE

R++

ReSharper C++
Visual Studio Extension for C++ developers

AC

AppCode
IDE for iOS and OS X development

CL

CLion
Cross-platform IDE for C and C++ developers

Start a free 30-day trial
jb.gg/cpp-accu

{cvu}
Volume 28 Issue 5
November 2016
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Silas S. Brown, Pete Goodliffe, Chris Oldwood, Roger Orr, Clint Swigart

ACCU Chair
chair@accu.org

ACCU Secretary
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

Necessary technology

There is much ado in the media at the moment about the Internet of Things. Specifically it's been precipitated by the recent occurrence of 'the Internet is broken', wherein several high-traffic and high-profile Internet presences were unavailable – to pretty much *everyone* in Europe and North America. The reason for the outage was that key DNS servers run by Dyn were suffering a Denial of Service attack – since reported to have been the largest on record. My understanding from several sources is that Dyn in particular coped with the attacks, and the outages, very well under the circumstances, but that's not really what I'm writing about. It seems that much of the traffic that caused the problems was sourced from, or routed through, the so-called IoT – the Internet of Things. Web-cams, baby monitors, tooth-brushes, printers, refrigerators, who knows what else.

Some devices require access to the Internet in order to be useful – for example, if you have a broadband router, it would not be very useful without access to the Internet. The problem, of course, is that without sufficient security, access *to* the Internet means access *from* the Internet, too. Some devices, on the other hand, do not require access to the Internet but have it because it's become fashionable. I've already mentioned some such devices. It's a problem because these devices are extremely resource-restricted (you can't get much processing power onto a toothbrush), and there simply isn't space for the extra security needed to make them resilient. I do, however, question the necessity of their connectedness. I've alluded (cynically, yes) to the marketability of being an on-line device, and it's a bandwagon that's being overwhelmed by a huge variety of seemingly innocuous devices which, frankly, aren't up to the job of doing it securely. A lack of security in one device has the potential to put entire residential networks at risk, with the subsequent fall-out in identity theft, online fraud and so on.

Is there something that the software community at large can do to help? And how does one change the default password on a toothbrush, anyway?

It will come as no surprise to anyone that this is a topic to which I will return...

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

DIALOGUE

SUBMISSION DATES

C Vu 28.6	1st December 2016
C Vu 29.1:	1st February 2017
Overload 137:	1st January 2017
Overload 138:	1st March 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

Speak Up!

Pete Goodliffe urges us to speak to the animals (that is, to other developers).

The single biggest problem in communication is the illusion that it has taken place.
~ George Bernard Shaw

It's the classic stereotype of a programmer: an antisocial geek who slaves alone, in a stuffy room with dimmed lights, hunched over a console tapping keys furiously. Never seeing the light of day. Never speaking to another person 'in real life'.

But nothing could be further from the truth.

This job is *all* about communication. It's no exaggeration to say we succeed or fail based on the quality of our communication.

This communication is more than the conversations that kick off at the water cooler. Although those are essential. It's more than conversations in a coffee shop, over lunch, or in the pub. Although those are all also essential.

Our communication runs far deeper; it is multifaceted.

Code is communication

Software itself, the very act of writing code, is a form of communication.

This works several ways...

Talking to the machines

When we write code we are *talking to* the computer, via an interpreter. This may literally be an 'interpreter' for scripting languages that are interpreted at runtime. Or we communicate via a translator: a compiler or JIT. Few programmers these days converse in the CPU's natural language: machine code.

Our code exists to give a literal list of instructions to the CPU.

Every so often, my wife leaves me a list of jobs to do. *Make dinner, clean the living room, wash the car.* If her instructions are illegible, or unclear, I won't do what she actually wants me to. I'll iron the cutlery and hoover the bathtub. (I've learnt to not argue, and do what I'm told, even if it makes no sense to me.) If she wants the right results, she has to leave me the right kind of instructions.

It is the same with our code.

Sloppy programmers are not explicit. The results of their code can be the equivalent of ironed cutlery.

Code is communication with the computer. It must be clear and unambiguous if your instructions are to be carried out as you intend.

We are not talking in the CPU's mother tongue, so it's always important to know what nuances of its language get lost in translation to our programming language. The convenience of using our preferred language comes at a cost.

Talking to the animals

Although your code forms an ongoing conversation with your mechanical friend, the computer, it does not *just* speak to a CPU.

It speaks to other humans, too – to the other people who share the code with you, and who have to *read* what you have written. It is read by the people you are collaborating with. It is read by the people who review your work. It is read by the maintenance programmer who picks up your code later on. It will be read by you when you come back in a few months to fix nasty bugs in your old handiwork.

Your code is communication to other humans. Including you. It must be clear and unambiguous if others are to maintain it.

This is important.

A high-calibre programmer strives to write code that clearly communicates its intent. The code should be transparent: exposing the algorithms, not obscuring the logic. It should enable others to modify it easily.

If code does not reveal itself, showing what it does, then it will be difficult to change. And the one thing we know about coding in the real world is *the only constant is change*. Uncommunicative code is a bottleneck and will impede your later development.

Good code is not terse to the point of unreadability. But neither is it lengthy and laboured. And it is most definitely not filled with comments. More comments *do not* make code better, they just make it longer – and probably worse as the comments can easily get out of sync with the code.

More comments do *not* necessarily make your code better. Communicative code does not need extra commentary to prop it up.

Good code is not trickily clever, deftly using 'advanced' language features to such aplomb that it will leave maintenance programmers scratching their heads. (Of course, the amount of head scratching does depend on the quality of the maintenance programmers; this kind of thing always depends on context.)

The quality of our expression in code is determined by the programming languages we choose to use, and in how we use them. Are you using a language that allows you to naturally express the concepts you are modelling?

We must talk the same language at the same time, or we'll suffer a biblical Tower of Babel cacophony. The team working on a section of code must write in the same language; it's not a winning formula to add lines of Basic to a Python script. If your entire application is written in C++, then the first person to add code in another language had better have a compelling reason.

However, even in an environment using the same programming language, it is possible to use different dialects and end up introducing communication barriers. You may adopt different formatting conventions, or employ different coding idioms (e.g., using 'modern' C++ *versus* 'C++ as a better C').

Of course, using multiple programming languages is not evil. Larger projects may legitimately be composed of code in more than one language. This is a standard for big distributed systems

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

Delivering Bad News from QA

Silas S. Brown describes how not to report your senior colleague's bug.

As I type this on a vintage 1999 Psion Revo (remember them?), my Cantonese wife of 18 months is tapping away into LibreOffice on her old laptop (whose malware-infested Windows I upgraded to Linux shortly into our marriage), on the kitchen table of my parents' house in a West Dorset village (we're visiting them this week), occasionally calling me over to check some aspect of English grammar. She is writing to a *Nature* journal about a C program I wrote which we hope will speed up cancer research teams across the world. But just last week, that same program caused her to feel so bullied by a senior colleague that she resigned, took her notice period off sick and doesn't want to go anywhere near a research lab again.

About a month previously, my wife had been asked to check batches of molecular primers for a battery of DNA tests and ended up waiting on into the night for the lab's computer to finish simulating their mutual interactions, then bringing home the Visual Basic 6 program which we tried to run on WINE but it got nowhere all weekend (I later found her input was one or two orders of magnitude larger than what that program was designed to take). So I sat her down and fired off a string of stupid non-biologist questions until I'd coaxed out a specification that I could use as a starting point for re-implementing the thing, except my version was in C with bit-pattern techniques (making full use of 64-bit registers to test many DNA bases at a time) and OpenMP parallelisation. Because this ran literally thousands of times faster than their Visual Basic affair, it opened up the possibility of simulating on larger scales and automating more aspects of the pooling process; I added the extra features they needed, and it seemed none of the other pieces of software out there had features in just the right combination for that kind of cash-strapped cancer research lab (I heard the principal investigator was trying to employ three assistants on a grant that was meant to be for one, so they obviously

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

Speak Up! (continued)

where the backend runs on a server in one language, with remote clients implemented in other, often more dynamic, browser-hosted languages. This kind of architecture allows you to employ the right kind of language for each task. We see here yet another language in play: the language that those parts communicate through (perhaps a REST API with JSON data formatting).

Consider also the *natural language* you program in. Most teams are based in the same country, so this is not a concern. However, I often work on multi-country projects with many non-native English speakers. We made a conscious choice to write all code in English: all variable names, comments, class or function names, everything. This affords us a degree of sanity.

I've worked on multi-site projects that didn't do this, and it's a real problem having to run code comments through Google Translate to work out if they're important or not. I've been left wondering whether a variable name has a Hungarian wart at the start, is misspelled, abbreviated, or if I just have a very bad grasp of the natural language used.

it's a real problem having to run code comments through Google Translate to work out if they're important or not

How well code communicates depends on the programming language, idioms employed, and the underlying natural language. All these have to be understood by the readership.

Remember that code is read by humans far more often than it is written. Therefore, it should be optimised for reading, not for writing. Use a concise construct only if it's easier for someone else to understand, rather than easier for you to type. Follow a layout convention that reveals intent clearly, not one that requires fewer keystrokes.

Talking to tools

Our code communicates even further – to other tools that work with it. Here 'tools' is not a euphemism for your colleagues.

Your code may be fed into documentation generators, source control systems, bug tracking software, and code analysers. Even the editors we use can have a bearing (what character set encoding is your editor using?).

It isn't unusual to add extra directives to our code to sate these processors' whinging, or to adapt our code to suit those tools (adjusting formatting, comment style, or coding idioms).

How does this affect the readability of the code?

Next time

So: code *is* communication. It's the kind of communication we continually strive to improve on.

In the next instalment we'll step beyond this, and look at how to improve our interpersonal communication. We'll consider how we communicate with other people, with the wider development team, and even – shock horror – with the customer. ■

Questions

- How clear is the code you write?
- Does it always communicate your intent?
- What can a programmer do to improve communicate when using code as a medium?
- Should the code you write be adjusted based on the audience you expect for it (e.g. the make up of the team that are working on it)?

needed to conserve resources at every turn and that's why optimisation was important). Everyone seemed excited that I'd been able to write what they needed.

But then it turned sour. We tested the software by asking it to check the mutual interactions in a set of pools that had been meticulously derived by hand by one of that lab's senior academics a couple of years earlier, and it pointed out 11 amplicon overlaps [1]. That academic was supposed to have eliminated all amplicon overlaps, and had done a pretty good job of getting rid of hundreds of them, but the fact that there'd been 11 left called into question their work and potentially the lab's results for the last 2 years. We carefully checked if my software report was incorrect. It wasn't. My wife went to the lab and basically said 'look how good our software is: it found these 11 mistakes in Dr X's work'. And said senior academic took things very badly and, I'm told, began the workplace bullying that led to my wife's resignation as previously mentioned.

to err is human (no matter how clever you are) and it's good for all of us to run our work through some form of automated testing

I should have insisted on going to her lab myself to present the software and take Q&A. Not because I'm macho (which I'm not over much), nor because my visual impairment deprives the bullies of one of their dimensions of expression, nor because I have the experience of being physically bullied by dozens of boys in primary school and therefore ended up with a wider sense of perspective, but simply because, as a programmer, I know that bugs can happen to anybody. I could have included in my presentation tales of Cambridge professor Sir Maurice Wilkes finding a bug in his second EDSAC program in the late 1940s, and no doubt other examples to show that to err is human (no matter how clever you are) and it's good for all of us to run our work through some form of automated testing, and that sanity-checking tools that work well are good no matter who we are (note the 'we'; it sounds better than 'you' when the subject is human error). One does not join an organisation and within months walk up to seniority (especially non-programmers) and say 'I found a bug in your work' (so there). One says 'I was trying to put your work through this compiler and it said this; do you think that's a problem?' or some such. (Make sure to say it was the compiler that found the problem, not you. You might have thought finding problems gets you credit, but it might instead get you some wrath.) Non-programmers are not used to bug reports, so we have to be tactful especially if we've just developed a new QA tool that's throwing up newly-discovered problems (after all, it's also possible the tool is simply wrong to flag up all that). Unfortunately I didn't think to warn my wife about all this in advance; she didn't have previous experience delivering such reports to senior non-programmers and she naturally put the proverbial foot in it. I honestly don't know how far their project is going to get now she's left, and let's not even think about how many more patients die while things are delayed. At least we're still trying to get the software past peer review so other labs can start using it (I asked an old classmate who now works in a commercial bio lab and she basically said 'sounds good but tell me when it's gone through peer review'; many labs have a policy of not being the first to try new tools, so having it published in a medical journal is important).

Once our *Nature* paper is ready to submit, we take the daily village bus (which my late grandmother campaigned for and subsequently couldn't use) to visit the public library of the small nearby town (around which my parents lived all their lives), where we have to battle with Windows against a time limit – shorter than expected because somebody else had booked the computer, and they've locked down the configuration so I can't turn up the print size or type in the Dvorak keyboard layout, and accidentally muttering 'for goodness' sake' gets me told off for swearing in the library – but we somehow get it done. Two days later, I am able to briefly get a signal in the village by holding my mobile phone against a tree on the hillside (the capacitative coupling seems to help) and we see *Nature* thought our paper was too specialist for them. Their response is a model of how to say this the right way: they specifically said they didn't doubt the technical quality of our work or its interest to others working with primer pools; they just didn't think the advances presented would have sufficiently significant immediate impact on the broader readership. They wanted to return it to us as soon as possible so it can be sent elsewhere without delay. My wife is very happy about this and says we will look at it once we're back in Cambridge (perhaps we've done enough of rushing it from the rurals). We'll probably try the *Oxford Bioinformatics* journal (I wouldn't want to task our very own Frances Buontempo with assembling a medically-qualified peer review team for an *Overload* paper).

If we could all train ourselves to be as nice as those *Nature* editors when reporting problems within an organisation, perhaps our professional relationships would be a tiny bit better. Perhaps.

Postscript: Shortly after we returned, my father succumbed to his illness and passed away. We immediately made a second visit to Dorset and wrote the second paper while there, which my wife decided we should send to *Nucleic Acids Research*. That editor felt it's still off-topic but was kind enough to transfer it to another journal which might be appropriate; their decision is still pending. ■

Note

[1] An amplicon overlap is the error of placing primers for two overlapping sections of the DNA into the same pool.

Commit Checklist

Chris Oldwood goes through the motions of version control.

Once I've finished the inner development cycle for a new feature or bug fix – write tests, code, build and run – it's time to think about committing those changes and therefore publishing them to a wider audience, i.e. my team. With the fun part out of the way, it's quite easy to rush this last stage and therefore publish a change that falls short in some way, either from a short or longer term perspective. At its most disruptive, it could be the code which is broken through lack of proper integration or a partial (or overzealous) change-set that excludes (or unexpectedly includes) files or code that was never intended to be published. On the flip-side, the actual code might be good but the cohesiveness or surrounding documentation may be lacking, therefore making future software archaeology [1] harder than it need be.

VCS tool differences

Although the mental checklist I follow when committing any change is pretty much the same, there are some significant differences that have emerged due to the version control product I might be using. For the purposes of this article, I've tried to talk about 'publishing' a change when I mean the act of making it visible to the outside world. In a classic centralised VCS like Subversion the act of committing and publishing are one and the same (private branches notwithstanding), whereas for a distributed VCS like Git they are distinct steps. The latter (DVCS) brings a number of affordances over the former (CVCS) around committing that are particularly useful at commit time.

Hence the general order that these steps are done are somewhat dependent on the VCS tool in use. For example with a CVCS like Subversion where you have (had) no ability to commit to an integration branch locally first, you need to consider how you'll address the implicit merge that will occur as part of the integration step. In contrast with Git, you can focus on the steps one at a time – get your commit sorted, then integrate other's work, and finally tidy up before publishing.

In this article I've ordered the steps more like the CVCS world (review, integrate, then commit & publish) rather than the DVCS style (review, commit, integrate, then publish). Either way, the checklist I'll describe is essentially portable, it's just a little easier when you have more time between getting your house in order and showing it to the world at large.

Preparing the change-set

The first thing to do after hacking around the codebase trying to get the bug fixed or the feature added is to ensure the set of changes that will comprise the eventual commit is 'clean'. This means that what we eventually commit is what we intended to commit – no unexpected side-effects caused by transient workarounds, accidents or interference from the IDE and tools.

As a starter for ten, the code needs to compile and the tests need to pass. It's all too easy to make localised changes and run just the handful of tests that you believe is required to verify the change is working. Before committing, we need to take a step back as we switch mind-set from creator to publisher so, after a deep breath, I need to run the build script which should clean the workspace, compile the code and run all the necessary tests to give me the confidence I need to promote the change. It's important that the build script I use is the same one that my team-mates and the build server will use so that we all agree on what process is used to create the final deliverable.

I make an extra special effort to deep cleanse my workspace before doing the final build because too often in the past I've been caught out by stray artefacts leftover from development that have somehow managed to taint the change and make it appear to work. The CI server should be working from a fresh workspace and therefore I try and do the same, albeit by cleaning up rather than creating a fresh one. With Git, it's as easy as **`git clean -fdx`**, but with other VCS tools I manually write a script [2] to do it.

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 and can be easily distracted via gort@cix.co.uk or @chrisoldwood

Reviewing the change-set

With a (hopefully) minimal set of edited files and folders in my working copy, the next step is to verify that only the code I intended to change exists there. During development or manual testing I may have commented out some code, tweaked a default value or adjusted the configuration to get things into a known state for such testing. Or perhaps a tool did something I didn't realise and I need to back that out because it happened by accident.

Alternatively I may have added a new setting that I defaulted for the tests but which needs applying to other environments too. With the rise of automated refactoring tools and smarter IDEs many of the changes we make might now be done for us. However, a tool like this will only go so far and try to keep the code intact after each small edit, but it's common for a single small refactoring, such as the deletion of code, to unearth further similar refactorings. Static code analysis is usually limited to advising on the current snapshot instead of chasing the turtles all the way down, so we still have to do that bit manually.

In essence the review step may need to be done a number of times as we identify a problem with the change, make a correction, potentially re-build and re-run the tests, and then review the change once more in the context of the revised change-set.

Reviewing the code structure

My reviewing process probably follows that of most other people – initially I look at the set of files that have been added, modified or deleted. Depending on the type of change it might be mostly source code or project files / package configuration. Whilst scanning the list of files, I'm looking for anything that's obviously out of place, such as a project file change when I haven't added a new class, or a source file in a namespace that I wouldn't expect to have changed that might indicate some unintended coupling. Design smells that come out of this structural review usually get logged in my notebook [3] for later consideration (the discipline of staying out of the bigger rabbit holes is a tricky one to master).

The converse is also true – there may be files I'd expect to change if I changed at least one of them. For example adding a new setting to the DEV configuration probably means I need to do the same for other environments. Another is the updating of project dependencies as the version number usually causes a ripple across a couple of files such as the package configuration and project file. The same occurs at a higher level too – if a production code project file changes the associated test ones should too.

One mistake that's less of an issue with modern VCS tools is accidentally checking in temporary files because now they tend to support an exclusion list (e.g. `.gitignore`). Before this feature, you needed to be careful you didn't accidentally check-in some binary artefacts or per-user tool configuration files as it could cause very odd behaviour for your team mates. My clean script would ensure they were deleted before the review thereby side-stepping the issue, but tool upgrades and refactoring meant that the script still needed revisiting now-and-then and so I'd still be on the lookout at this stage for files which have escaped the ignore list.

Reviewing the code itself

Naturally the part we are probably most interested in is reviewing the actual code. First we need to ensure that what we think we changed was what we actually changed and that are no hangovers from spikes, testing, etc. I may have also commented out some code with the intention of deleting it before publication and so I need to make sure I actually did this as checking in commented out code is almost always a mistake.

In the pre-commit review, I'm not looking at the code changes from the perspective of 'does it work, does the design make sense and does it meet the guidelines', I'm looking at it to see if the diff makes sense to the future reader. When the software archaeologist is looking back over the revision history they are often looking at the evolution of change rather than each snapshot in isolation. This is why you often hear the advice about not checking in functional and formatting style changes in the same commit – it makes it hard to work out what the former is if it's buried in the latter. That doesn't mean I don't care about the more traditional style of code review, it's just that I would have done all that before I even get to the commit stage.

Integration is one area where the centralised and distributed VCS products differ greatly

I tend to look at diffs in two different ways depending on the change and how much I want to see of the surrounding context. The classic patch format where you only get to see the actual change and the odd line of context either side is great for simple changes where I just expect some in-place editing to have occurred. In large files this avoids all the scrolling around or navigating you might need to do as it usually fits on a single screen. Sadly many diff tools show the patch format highlighted on a per-line basis and so simple character changes can be harder to spot, but they are great for quickly eyeballing change sets you do frequently, such as updating 3rd party package versions.

The other diff view I use is the more fully featured 2-way (and 3-way for a merge) diff tool that shows the entire file with highlighting around the changes, e.g. KDiff3. I find these tend to make small intra-line changes and block movements a little more visible. When the change is a refactoring, I'm often more interested in seeing all the surrounding code because if a tool is involved it will limit its scope and seeing the change in situ often points out new inconsistencies that have resulted from the surgical precision of the tool.

Whilst I'm happy with the command line for many VCS tasks, diff'ing is one place where the GUI still does it for me. This includes both the structural and code reviews. I've found tools like TortoiseSvn (Git) or Git Extensions allow me to quickly execute different actions to clean up a change set. For example I might need to revert a few mistakes, scan the commit log of others, the odd blame here-and-there, and finally diff the lot.

Yes, you read that right, I always diff every change I commit. If that commit contains a lot of edits then it's a lot of files to diff, but I decided I'm not going to compromise on this principle because it takes time. In fact the more files in the change-set the more inclined I feel to review it thoroughly exactly because there is a greater chance of something untoward creeping in. Instead what it's taught me is that it's preferable to work on smaller units of change (which fits the modern development process nicely), but also to find an easier way to quickly review changes. The patch-style diff format presented as a single unified diff pays off handsomely here for boiler-plate changes as you can very quickly eyeball the entire set and commit it if turns out to be clean.

Splitting up changes

Even when you are doing continuous integration where you are committing incremental changes every few minutes or hours, it's still possible that a part of your overall change ends up fixing something tangential to the task at hand. I often try and use different tools or techniques when I can to exercise the codebase and tools in different ways to find those less obvious non-functional bugs. For example running a build or test script via a relative path often shows up poor assumptions about the relative path to any dependent scripts or tools. Whilst not essential to the product they are jarring and interrupt your flow.

When these kinds of minor bugs show up, I like to fix them right away as long as they really are trivial. This means that the entire change set will probably include two different logical changes – one for the task I'm doing and the other for the bug I fixed whilst developing it. Logically they are distinct changes and therefore I'd look to commit them separately. However, this extends further than per-file changes: it may be two independent changes within the same source file. Historically you'd either have to manually split these commits out or commit them together and make it clear via the commit message that there were two different motives for the change. Luckily modern tools like Git make it possible to easily to split a single bunch of edits to one workspace into separate coherent commits.

Although cherry picking changes across branches is generally frowned upon (mind you, so is branching itself these days), this has often been down to commits being overly large and failing to adhere to the Separation of Concerns. By ensuring unrelated changes are kept committed separately you make it much easier to surgically insert a fix from one branch into another should the need ever arise.

Integrating upstream changes

Once I'm happy that the changes I've made locally complete the task at hand I then need to see what's changed in the world around me. Working directly on the main integration branch [4] in small increments ensures that whatever has changed is likely to be quite small and therefore the chance for conflict is minimal – any unexpected merge should virtually always be trivial and handled automatically by the VCS tool.

It's imperative that I pull the latest upstream changes locally first, so that I can resolve any syntactic and semantic conflicts before pushing back. The former are fairly easy to spot when working with compiled languages, whereas the latter are somewhat trickier and rely on failing tests for the early warning siren. Whilst I could just push my changes and let a gated build inform me of any problems, I prefer to avoid context switching and so proactively integrate so that when I eventually manage to publish I'm pretty certain it won't come back to me for remediation.

Integration is one area where the centralised and distributed VCS products differ greatly. When using a centralised VCS, there was always the risk that pulling the latest changes to your workspace would invoke a complex merge which, if messed up, could cause you to corrupt your own changes. This reason alone convinced me to go back to single branch development and keep my commits small and focused. On the few occasions where the prospect of losing my work could have been expensive, I would use the 'private branch on demand' feature (e.g. branch from working copy in Subversion) to create a backup in the VCS. Then I'd update to the head and merge my private branch back in, safe in the knowledge that I could repeat it as many times as needed to resolve any conflicts. In the distributed world this kind of behaviour is inherent by the way the tool works (you commit to your private fork first) and so it's safer by default.

Handling merges

This in turn means you have more flexibility about how you handle any merge conflicts directly on an integration branch. In the DVCS world, you can pull the latest changes and do the bare minimum to commit your changes on top of it. Then you can resolve any syntactic and semantic problems in your own time and commit them separately as fixes to the merge. At this point you then have the choice of whether to leave history intact and publish separate commits (merge + fixes) or squash them into a single change and publish it as if nothing went wrong. For me, the decision of which approach to take depends heavily on whether the break was significant or not and therefore may have significance in the future too.

For example, if someone just renamed a method in a refactoring then I squash as it's simply a matter of poor timing, whereas if a design change has caused the rework it means my change was based on a different view of the world and so may hide further semantic problems. Ideally every commit to an integration branch should stand on its own two feet (i.e. the build and tests pass) but when you can publish the fix at the same time as the break the observable outcome is essentially the same as so it feels acceptable to break the rules in the rare case that it happens.

In the CVCS world you do not have this luxury as what you publish to an integration branch is immediately visible, hence I might do a 'what if?' merge first to test the water and if it looks dicey spin up a private branch for safety. This way I can guarantee that I'll only be publishing a consistent change and the private branch contains the gory details for posterity.

Noise reduction

As I mentioned earlier I like to keep my workspace as clean as possible and therefore before integrating I'll definitely do a "deep clean' to ensure that the subsequent build that follows the integration is as close to the build machine's as possible. This means that I'll ensure all transient files in the workspace are removed, e.g. cached NuGet and Node packages, as I want the best chance possible that any problem is strictly down to the integration itself and not some stray environmental issue caused by detritus.

In essence in a Git based project this boils down to the following one-liner:

```
call Clean --all && git pull --rebase && call Build
```

Documenting the changes

With the set of changes reviewed and integrated the last big hurdle is to document them. When committing, I need to provide a message that tells my future readers something about the change that will enlighten them to its purpose. The same is true of any comment, the writer needs to avoid saying what the code (and diff) already says and to instead focus on the rationale – the why.

It's not quite as black & white as that, though, because trawling through the revisions trying to find related changes is much better with some tools than others. For example the 'annotate' feature (or in its more witch-hunt friendly guise – blame) makes it easier to walk the history of a single file using the code itself for navigation. However, if you're using a lesser tool or doing some archaeology around a wider change then the commit message can become more significant in tracking down 'an interesting set of changes'.

Hence it helps if the message is broken into (at least) two sections – a short summary and then a more detailed explanation. The summary should be short, ideally fitting a single line of text, so that when you are scanning the list of recent commits that one line tells you in an instant what it's about. Curiously there appears to be some disagreement about which tense you should write your message in but I've always found the past tense perfectly adequate. Fortunately this two-part format has been brought to prominence by the likes of Git and so it's becoming the norm.

The start of any commit message I write nearly always contains some kind of reference to the feature I'm working on. For example if the user stories are being tracked in an enterprise tool like JIRA then I'll put the ticket number in, e.g. #PRJ-1234. Once again, somewhat fortuitously this practice has been adopted by other tools like GitHub as a way to link an issue with a commit. Linking the commit in this way means that I probably have less to document because much of commentary and rationale will already be in another tool.

Even if the project is not using a formal feature tracking tool, e.g. just a Trello board with simple task descriptions, I'll still use the card number as a message prefix. [5] This is because committing a feature in small chunks makes the set of related commits harder to find. Sometimes the team will review a change after it's already been committed and so the reference number makes it much easier to find all the related changes for the feature. If a change doesn't have a reference then it should be something so trivial as to be easily discernible from the one-line summary alone.

Naturally the body of the commit message will contain further details about why the change was made. If it's an informal bug fix, it might contain the basic problem; if a new feature then a bit about what support was added; and if it's tidying up then why the code is now redundant. Sometimes I'll include other little notes, such as if the commit isn't standalone for some reason. Whilst it's great if you can pick any arbitrary revision and expect it to be self-consistent, there are occasionally times when we have to check-in partial changes or changes that may not strictly work due to some environmental concerns. In these cases, the extra note helps the reader know that there are dragons lurking and therefore they should look further afield to find where they were eventually slain.

Once again, I personally find a UI advantageous for this part of the task, not least because it normally contains a spell checker to ensure that the basic text has no obvious typos. Whilst it's not an everyday occurrence searching the commit messages for a particular word, or phrase, it is useful and so it helps enormously if you do not have to compensate for spelling mistakes. A modern commit dialog also tends to be aware of the folders, files and even the source code that has changed and so can provide auto-completion of terms that might be relevant, such as the name of a method that was edited.

One thing I do miss about working with a CVCS is that you can often still edit a commit message after the fact. With a DVCS the message is part of the immutable chain of revisions and therefore attempting to fix it leads you into the kind of murky waters that you really want to avoid. If the changes only exist as local commits, you still have time to make the correction, but once it's published you pretty much just have to live with it.

Publishing the change-set

Phew! After all those considerations it's finally time to unleash my handiwork on the world at large. If I'm on Git then I'll do the final push, whereas with Subversion or TFS it will have happened when I hit 'OK' on the commit dialog. Either way the change is done, so barring any unforeseen weirdness shown up by the build pipeline, it's time pick another task and go round the development loop once more. ■

References

[1] 'In The Toolbox: Software Archaeology', *C Vu 26-1*, http://www.chrisoldwood.com/articles/in-the-toolbox-software-archaeology.html

[2] 'Cleaning the Workspace', Chris Oldwood, http://chrisoldwood.blogspot.co.uk/2014/01/cleaning-workspace.html

[3] 'In The Toolbox: Pen & Paper', *C Vu 25-4*, http://www.chrisoldwood.com/articles/in-the-toolbox-pen-and-paper.html

[4] 'Branching Strategies', Chris Oldwood, *Overload 121*, http://www.chrisoldwood.com/articles/branching-strategies.html

[5] 'In The Toolbox: Feature Tracking', *C Vu 26-3*, http://www.chrisoldwood.com/articles/in-the-toolbox-feature-tracking.html

On High Rollers

A student investigates the Baron's last puzzle.

In the Baron's most recent wager, he was to roll a twenty-sided die marked with the digits zero to nine twice apiece and place it either upon a space representing tens or upon another representing ones according to his fancy, after which Sir R----- was to do the same. Then the Baron and Sir R----- were to roll a second die each and place them upon their empty spaces. If the number thus made by the Baron was smaller than that made by Sir R-----, then Sir R----- was to have a prize of twenty nine coins from the Baron, otherwise the Baron was to have one of thirty coins from Sir R-----.

The key to figuring the fairness of the wager lies in recognising that there exists an optimal strategy that the Baron should have followed if he were at all desirous of victory and another that Sir R----- should have adopted if he were at all keen to frustrate him.

Indeed, I explained as much to the Baron, but I fear that he may not have entirely grasped its significance.

Specifically, if the Baron's first roll was five or greater then he should have placed the die upon his tens space, with the expectation that he was more likely than not to roll no greater with his second die, otherwise he should have placed it upon the ones, with precisely the opposite expectation.

In the first case, if Sir R----- rolled greater than the Baron then he should have placed his die upon his tens space for assured victory. If he instead rolled lower then he should have placed it upon his ones space to stave off assured defeat. Finally, if he rolled equally then he should have placed it upon the tens space with the same expectation that he was more likely than not to roll no better with his second die.

In the second case, Sir R----- should simply have taken the Baron's strategy and placed the die upon his ones space if he rolled less than five and upon his tens space otherwise.

If we label the Baron's first die b_1 and Sir R-----'s r_1 then we can express these contingencies as

$$b_1 \geq 5 \wedge r_1 > b_1$$
$$b_1 \geq 5 \wedge r_1 = b_1$$
$$b_1 \geq 5 \wedge r_1 < b_1$$
$$b_1 < 5 \wedge r_1 \geq 5$$
$$b_1 < 5 \wedge r_1 < 5$$

where $\wedge$ stands for *and*.

Now Sir R----- is sure to win in the first case, which occurs with a probability of

$$\Pr(b_1 \geq 5 \wedge r_1 > b_1) = \sum_{b_1=5}^{9} \frac{1}{10} \times \frac{9-b_1}{10} = \frac{4+3+2+1+0}{100} = \frac{1}{10}$$

where Σ is the summation sign. Here we're exploiting the facts that each number from zero to nine has one chance in ten of being rolled and that there are $9-b_1$ numbers between zero and nine that are greater than b_1.

there exists an optimal strategy that the Baron should have followed if he were at all desirous of victory

In the second case, Sir R----- must roll higher than the Baron with his second die to secure victory

$$b_1 \geq 5 \wedge r_1 = b_1 \wedge r_2 > b_2$$

an eventuality that has a likelihood of

$$\begin{aligned}\Pr(b_1 \geq 5 \wedge r_1 = b_1 \wedge r_2 > b_2) &= \Pr(b_1 \geq 5 \wedge r_1 = b_1) \times \Pr(r_2 > b_2) \\ &= \left(\frac{1}{2} \times \frac{1}{10}\right) \times \left(\sum_{b_2=0}^{9} \frac{1}{10} \times \frac{9-b_2}{10}\right) \\ &= \frac{1}{20} \times \frac{9+8+7+6+5+4+3+2+1+0}{100} \\ &= \frac{1}{20} \times \frac{45}{100} = \frac{9}{400}\end{aligned}$$

In the third case there are two possible conclusions in which Sir R----- prevails. Firstly, if his second roll is greater than the Baron's first

$$b_1 \geq 5 \wedge r_1 < b_1 \wedge r_2 > b_1$$

and secondly if it is equal to it and the Baron's second roll is less than Sir R-----'s first

$$b_1 \geq 5 \wedge r_1 < b_1 \wedge r_2 = b_1 \wedge b_2 < r_1$$

The chances of these outcomes are

$$\begin{aligned}\Pr(b_1 \geq 5 \wedge r_1 < b_1 \wedge r_2 > b_1) &= \sum_{b_1=5}^{9} \frac{1}{10} \times \frac{b_1}{10} \times \frac{9-b_1}{10} \\ &= \frac{5\times4+6\times3+7\times2+8\times1+9\times0}{1000} \\ &= \frac{20+18+14+8}{1000} = \frac{3}{50}\end{aligned}$$

and

$$\begin{aligned}\Pr(b_1 \geq 5 \wedge r_1 < b_1 \wedge r_2 = b_1 \wedge b_2 < r_1) &= \sum_{b_1=5}^{9} \frac{1}{10} \times \sum_{r_1=0}^{b_1-1} \frac{1}{10} \times \frac{1}{10} \times \frac{r_1}{10} \\ &= \sum_{b_1=5}^{9} \sum_{r_1=0}^{b_1-1} \frac{r_1}{10000}\end{aligned}$$

Now the inner sum here is an arithmetic series and so, by the law that governs them, must satisfy

$$\sum_{i=0}^{n} i \times d = \frac{1}{2} \times n \times (n+1) \times d$$

and consequently

$$\begin{aligned}&\Pr(b_1 \geq 5 \wedge r_1 < b_1 \wedge r_2 = b_1 \wedge b_2 < r_1)\\ &= \sum_{b_1=5}^{9} \frac{\frac{1}{2}\times(b_1-1)\times(b_1-1+1)}{10000}\\ &= \sum_{b_1=5}^{9} \frac{(b_1-1)\times b_1}{20000}\\ &= \frac{4\times5+5\times6+6\times7+7\times8+8\times9}{20000}\\ &= \frac{20+30+42+56+72}{20000}\\ &= \frac{11}{1000}\end{aligned}$$

Similarly, there are two outcomes following from the fourth case in which Sir R----- takes the prize; if the Baron's second roll is less than Sir R-----'s first, or if it equals it and Sir R-----'s second roll is greater than the Baron's first

$$b_1 < 5 \wedge r_1 \geq 5 \wedge b_2 < r_1$$
$$b_1 < 5 \wedge r_1 \geq 5 \wedge b_2 = r_1 \wedge r_2 > b_1$$

We can figure the chances of these with

$$\begin{aligned}\Pr(b_1 < 5 \wedge r_1 \geq 5 \wedge b_2 < r_1) &= \frac{1}{2}\times\sum_{r_1=5}^{9}\frac{1}{10}\times\frac{r_1}{10}\\ &= \frac{1}{2}\times\frac{5+6+7+8+9}{100}\\ &= \frac{1}{2}\times\frac{35}{100} = \frac{7}{40}\end{aligned}$$

and

$$\begin{aligned}\Pr(b_1 < 5 \wedge r_1 \geq 5 \wedge b_2 = r_1 \wedge r_2 > b_1) &= \sum_{b_1=0}^{4}\frac{1}{10}\times\frac{1}{2}\times\frac{1}{10}\times\frac{9-b_1}{10}\\ &= \frac{9+8+7+6+5}{2000} = \frac{7}{400}\end{aligned}$$

Finally, in the last case Sir R----- wins if his second roll exceeds the Baron's or if it is equal and his first roll was greater than the Baron's

$$b_1 < 5 \wedge r_1 < 5 \wedge r_2 > b_2$$
$$b_1 < 5 \wedge r_1 < 5 \wedge r_2 = b_2 \wedge r_1 > b_1$$

which have likelihoods of

$$\begin{aligned}\Pr(b_1 < 5 \wedge r_1 < 5 \wedge r_2 > b_2) &= \frac{1}{2}\times\frac{1}{2}\times\sum_{b_2=0}^{9}\frac{1}{10}\times\frac{9-b_2}{10}\\ &= \frac{9+8+7+6+5+4+3+2+1+0}{400}\\ &= \frac{9}{80}\end{aligned}$$

and

$$\begin{aligned}&\Pr(b_1 < 5 \wedge r_1 < 5 \wedge r_2 = b_2 \wedge r_1 > b_1)\\ &= \Pr(b_1 < 5 \wedge r_2 = b_2 \wedge r_1 < 5 \wedge r_1 > b_1)\\ &= \sum_{b_1=0}^{4}\frac{1}{10}\times\frac{1}{10}\times\frac{4-b_1}{10}\\ &= \frac{4+3+2+1+0}{1000}\\ &= \frac{1}{100}\end{aligned}$$

Note that, since we're only considering those circumstances in which the Baron's and Sir R-----'s first rolls were less than five, there are but $4-b_1$ chances in ten that r_1 was greater than b_1.

Having enumerated each and every way in which Sir R----- might have defeated the Baron, we need simply add their probabilities to figure the likelihood that he should have done so.

$$\begin{aligned}&\frac{1}{10}+\frac{9}{400}+\frac{3}{50}+\frac{11}{1000}+\frac{7}{40}+\frac{7}{400}+\frac{9}{80}+\frac{1}{100}\\ &= \frac{200}{2000}+\frac{45}{2000}+\frac{120}{2000}+\frac{22}{2000}+\frac{350}{2000}+\frac{35}{2000}+\frac{225}{2000}+\frac{20}{2000}\\ &= \frac{1017}{2000}\end{aligned}$$

Sir R-----'s expected winnings were therefore

$$\frac{1017}{2000}\times29-\frac{983}{2000}\times30 = \frac{29493}{2000}-\frac{29490}{2000} = \frac{3}{2000}$$

and I should have had no compunction whatsoever in suggesting that he take on the Baron's challenge!

But alas, I should have been wrong to do so; the diligent Mister O-- [1] has deduced that the Baron should have been better served had he first cast a five if he had placed it upon his ones space!

Now it is still the case that, should the Baron have first rolled five or less and placed his die upon the ones, Sir R----- should have placed his first upon the tens if it were a five since he would have won the wager if either

$$b_2 < 5$$

or

$$b_2 = 5 \wedge r_2 > b_1$$

which would happen with probabilities

$$\begin{aligned}\Pr(b_2 < 5) &= \frac{1}{2}\\ \Pr(b_2 = 5 \wedge r_2 > b_1) &= \frac{1}{10}\times\sum_{b_1=0}^{5}\frac{1}{6}\times\frac{9-b_1}{10}\\ &= \frac{9+8+7+6+5+4}{600} = \frac{39}{600}\end{aligned}$$

since there are but six such outcomes for the Baron's first die, totalling 339 chances in 600. In contrast, if Sir R----- had placed his five in the ones space then he should have triumphed in the eventualities

$$r_2 > b_2$$
$$r_2 = b_2 \wedge b_1 < 5$$

which have probabilities of

$$\begin{aligned}\Pr(r_2 > b_2) &= \sum_{b_2=0}^{9}\frac{1}{10}\times\frac{9-b_2}{10}\\ &= \frac{9+8+7+6+5+4+3+2+1+0}{100}\\ &= \frac{9}{20} = \frac{270}{600}\\ \Pr(r_2 = b_2 \wedge b_1 < 5) &= \frac{1}{10}\times\frac{5}{6} = \frac{1}{12} = \frac{50}{600}\end{aligned}$$

totalling just 320 chances in 600. Sir R----- should therefore have adopted the former strategy and we must consider the cases

$$b_1 > 5 \wedge r_1 > b_1$$
$$b_1 > 5 \wedge r_1 = b_1$$
$$b_1 > 5 \wedge r_1 < b_1$$
$$b_1 \leq 5 \wedge r_1 \geq 5$$
$$b_1 \leq 5 \wedge r_1 < 5$$

The first of these occurs with probability

$$\Pr(b_1 > 5 \wedge r_1 > b_1) = \sum_{b_1=6}^{9} \frac{1}{10} \times \frac{9-b_1}{10} = \frac{3+2+1+0}{100} = \frac{3}{50}$$

and ensures victory for Sir R-----.

Once again, Sir R----- will emerge victorious in the second case only if his second roll exceeds the Baron's, which happens with a probability of

$$\begin{aligned}
&\Pr(b_1 > 5 \wedge r_1 = b_1 \wedge r_2 > b_2)\\
&\quad = \Pr(b_1 > 5 \wedge r_1 = b_1) \times \Pr(r_2 > b_2)\\
&\quad = \left(\frac{4}{10} \times \frac{1}{10}\right) \times \left(\sum_{b_2=0}^{9} \frac{1}{10} \times \frac{9-b_2}{10}\right)\\
&\quad = \frac{1}{25} \times \frac{9+8+7+6+5+4+3+2+1+0}{100}\\
&\quad = \frac{1}{25} \times \frac{45}{100} = \frac{9}{500}
\end{aligned}$$

In the third case Sir R----- will win if either his second die is greater than the Baron's first or if it is equal and the Baron's second is less than his first, for which we can figure the probabilities

$$\begin{aligned}
\Pr(b_1 > 5 \wedge r_1 < b_1 \wedge r_2 > b_1) &= \sum_{b_1=6}^{9} \frac{1}{10} \times \frac{b_1}{10} \times \frac{9-b_1}{10}\\
&= \frac{6\times3+7\times2+8\times1+9\times0}{1000}\\
&= \frac{18+14+8}{1000} = \frac{1}{25}
\end{aligned}$$

and

$$\begin{aligned}
&\Pr(b_1 > 5 \wedge r_1 < b_1 \wedge r_2 = b_1 \wedge b_2 < r_1)\\
&\quad = \sum_{b_1=6}^{9} \frac{1}{10} \times \sum_{r_1=0}^{b_1-1} \frac{1}{10} \times \frac{1}{10} \times \frac{r_1}{10}\\
&\quad = \sum_{b_1=6}^{9} \sum_{r_1=0}^{b_1-1} \frac{r_1}{10000}\\
&\quad = \sum_{b_1=6}^{9} \frac{\frac{1}{2} \times (b_1-1) \times (b_1-1+1)}{10000}\\
&\quad = \frac{5\times6+6\times7+7\times8+8\times9}{20000}\\
&\quad = \frac{30+42+56+72}{20000} = \frac{1}{100}
\end{aligned}$$

The two winning outcomes for Sir R----- in the fourth case are now

$$b_1 \leq 5 \wedge r_1 \geq 5 \wedge b_2 < r_1$$
$$b_1 \leq 5 \wedge r_1 \geq 5 \wedge b_2 = r_1 \wedge r_2 > b_1$$

having likelihoods of

$$\begin{aligned}
\Pr(b_1 \leq 5 \wedge r_1 \geq 5 \wedge b_2 < r_1) &= \frac{6}{10} \times \sum_{r_1=5}^{9} \frac{1}{10} \times \frac{r_1}{10}\\
&= \frac{3}{5} \times \frac{5+6+7+8+9}{100}\\
&= \frac{3}{5} \times \frac{35}{100} = \frac{21}{100}
\end{aligned}$$

and

$$\begin{aligned}
\Pr(b_1 \leq 5 \wedge r_1 \geq 5 \wedge b_2 = r_1 \wedge r_2 > b_1) &= \sum_{b_1=0}^{5} \frac{1}{10} \times \frac{1}{2} \times \frac{1}{10} \times \frac{9-b_1}{10}\\
&= \frac{9+8+7+6+5+4}{2000}\\
&= \frac{39}{2000}
\end{aligned}$$

In the fifth and final case Sir R----- needs his second die to be greater than the Baron's second or, if it equals it, his first to be greater than the Baron's first, having chances of

$$\begin{aligned}
&\Pr(b_1 \leq 5 \wedge r_1 < 5 \wedge r_2 > b_2)\\
&\quad = \frac{3}{5} \times \frac{1}{2} \times \sum_{b_2=0}^{9} \frac{1}{10} \times \frac{9-b_2}{10}\\
&\quad = \frac{3}{10} \times \frac{9+8+7+6+5+4+3+2+1+0}{100}\\
&\quad = \frac{3}{10} \times \frac{45}{100} = \frac{27}{200}
\end{aligned}$$

in the first eventuality and

$$\begin{aligned}
&\Pr(b_1 \leq 5 \wedge r_1 < 5 \wedge r_2 = b_2 \wedge r_1 > b_1)\\
&\quad = \Pr(b_1 < 5 \wedge r_2 = b_2 \wedge r_1 < 5 \wedge r_1 > b_1)\\
&\quad = \sum_{b_1=0}^{4} \frac{1}{10} \times \frac{1}{10} \times \frac{4-b_1}{10}\\
&\quad = \frac{4+3+2+1+0}{1000} = \frac{1}{100}
\end{aligned}$$

in the second since Sir R-----'s first die could not possibly have exceeded the Baron's if it were a five.

Adding together these probabilities yields

$$\begin{aligned}
&\frac{3}{50} + \frac{9}{500} + \frac{1}{25} + \frac{1}{100} + \frac{21}{100} + \frac{39}{2000} + \frac{27}{200} + \frac{1}{100}\\
&= \frac{120}{2000} + \frac{36}{2000} + \frac{80}{2000} + \frac{20}{2000} + \frac{420}{2000} + \frac{39}{2000} + \frac{270}{2000} + \frac{20}{2000}\\
&= \frac{1005}{2000}
\end{aligned}$$

which unfortunately turns the tide against Sir R----- whose expected outcome was consequently

$$\frac{1005}{2000} \times 29 - \frac{995}{2000} \times 30 = \frac{29145}{2000} - \frac{29850}{2000} = -\frac{705}{2000}$$

and he would have been most ill-served by my advice! □

Acknowledgement

Courtesy of www.thusspakeak.com

[1] With thanks to Roger Orr.

A Commoner’s Response

Roger Orr offers an analysis of the Baron’s last game.

I was intrigued by the description of the game Baron M and Sir R were playing in the last *CVu*.

The strategy for each player is obviously to try and maximise their own score. But the problem is working out how each player should best approach this – it is more subtle than it seems at first sight!

Let’s start with player one and their first roll of the die.

Since half the numbers are 0–4 and half are 5–9 it seems clear that when throwing a number from 0 to 4 first it would be best for them to put this die in the units field and hope that lady luck treats them better on the second throw!

If the first throw is from 6 to 9 the chances are that the second throw will be lower, so it is better to place the first die in the tens.

The hard choice comes if they throw a 5 first. Half the time the next number will be lower, which is bad, but half the time it will be the same or greater. So what’s the best strategy to employ here? It’s not easy to decide, as deferring the choice might improve our chances, so let’s come back to that later.

Player two then faces two different positions depending on whether player one put his first die in the tens spot or the units spot.

In the tens case, it seems simple: play your coin in the tens if it is bigger, and in the units if it is less.

But again the harder decision is what is the best strategy if it is the same – what then?

If the score is from 6 to 9 they’re unlikely to do as well on the second roll, and should play the tens spot, but if they roll a 5 they’re likely to do no worse next time – and might do better. So should they match player one on a 5, or play a 5 in the units and hope to get a higher roll next time? Let’s come back to that question later as well.

If player one has placed his die in the units spot then player two faces the same question as player one did, *mutatis mutandis*.

Again, if they throw from 0 to 4 they should place it in the units spot, hoping to get a higher score on their second roll, and if they throw from 6 to 9 they should place the die in the tens spot.

Once again, rolling 5 presents them with a more difficult problem.

So we have three questions to answer to decide the best strategy to play this game.

The juxtaposition in *CVu* of the Baron’s game with the article on ‘Random confusion’ led me to run a simulation to try and see by experiment what happens, assuming a set of *fair* 20-sided dice (creating these would be no mean feat, I surmise, and would require excellent engineering skills!)

I anticipate it will take experienced players something like 30 seconds to play a single game, counting the coins probably being the slowest part. The gentry may have it differently, but those who work for their living must restrain their dicing habits to the evening or risk ruin. So assuming the two players can manage dice playing continuously from 8pm to midnight we should see what happens after each night’s games, by when about 480 games will have been completed. (Although what with pouring more wine and answering any calls of nature that might occur they might not *quite* achieve that many games each evening.)

There are three binary choices for strategy options so there are eight possible combinations to try out.

- Option 1: player one puts a 5 in the tens spot, rather than in the units.
- Option 2: player two puts a matching die in the tens spot if it matches a 5 in player one’s tens spot, rather than in the units.
- Option 3: player two puts a 5 in the tens spot if player one has played in the units spot, rather than in the units.

(Although they’re not actually independent choices – if player one always puts a 5 in the units spot then player two never has the opportunity to invoke option 2.)

I ran a simulation of eight nights, running through one set of options on each night.

Option 1	Option 2	Option 3	Player one’s winnings (losses)
0	0	0	(822)
1	0	0	476
0	1	0	889
1	1	0	181
0	0	1	(232)
1	0	1	(173)
0	1	1	(114)
1	1	1	(232)

But one simulation is hardly enough, let us see what a second eight nights produces.

Option 1	Option 2	Option 3	Player one’s winnings (losses)
0	0	0	(114)
1	0	0	(232)
0	1	0	358
1	1	0	299
0	0	1	(173)
1	0	1	476
0	1	1	(645)
1	1	1	63

Oh dear – it doesn’t look very consistent, does it? Perhaps a simulation of a single night’s playing for each option isn’t long enough to be certain of our strategy, dice being the unruly objects that they are.

Let’s see what happens at the end of a whole **year** of playing:

Option 1	Option 2	Option 3	Player one’s winnings (losses)
0	0	0	12,865
1	0	0	2,481
0	1	0	9,207
1	1	0	3,543
0	0	1	17,231
1	0	1	(115)
0	1	1	4,782
1	1	1	(232)

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

ConFoo

ACCU member Clint Swigart speaks at a multi-discipline conference in Canada.

ConFoo's Anna Filina provides the following information on the conference's history and importance.

The Montreal-based PHP conference ran successfully by the local community for seven years. After that, the team decided to include other technologies to transform it into a polyglot conference called ConFoo. Since 2016, a second Vancouver event was created. The mission was simple: increase the quality of web applications and the productivity of the developers who write it. ConFoo achieves that by inviting some of the top industry experts from around the globe to share their real-world experience.

Having an international presence is important, because people in close proximity end up thinking alike. When I travel to conferences around the world, I discover that people have different priorities and mindsets that I did not see in two decades of writing software in my own country. Developers understand this, which is why we get attendees from as far as Poland and New Zealand.

Our sponsors help us greatly with the promotion of ConFoo abroad, which is why we get such a diverse audience and speaker lineup. For example, ACCU gets the word out to software developers in the UK, the rest of Europe and beyond. Because ConFoo is non-profit, such support enables us to spend more money and effort on creating a better experience at the event.

Every programmer should be excited to attend ConFoo. With 100 presentations in Vancouver and more than 150 in Montreal, it is one of the largest events of its kind. This conference is aimed specifically at software developers. It is a great place to sharpen those hard tech skills. There is a great range of topics, including many programming languages, databases, security, performance, machine learning and management.

We are excited to have ACCU members both in attendance and among speakers. If you want to be part of this learning adventure, get tickets to the event of your choice at confoo.ca or follow @confooca on Twitter for updates.

CLINT SWIGART

Clint is a Certified Public Accountant enjoying life in Tampa FL. He has been hooked on learning more about programming ever since he was introduced to Ruby and JavaScript classes online. His number one fear in life is not public speaking or being contacted at cds8011@gmail.com

A Commoner's Response (continued)

This is looking quite hopeful for player one, but if we run the results for a second year we again see quite a few changes in the results:

Option 1	Option 2	Option 3	Player one's winnings (losses)
0	0	0	4,841
1	0	0	3,897
0	1	0	(2,652)
1	1	0	(1,944)
0	0	1	8,027
1	0	1	(3,773)
0	1	1	9,797
1	1	1	1,714

Let's assume playing this game does amazing things to the players' life expectancy and they manage to keep playing for one **hundred** years. (They'd better also have a very high boredom threshold, and a very large supply of coins....)

Option 1	Option 2	Option 3	Player one's winnings (losses)
0	0	0	10,411,745
1	0	0	1,316,482
0	1	0	10,105,948
1	1	0	1,114,761
0	0	1	7,711,315
1	0	1	(163,061)
0	1	1	8,102,898
1	1	1	(290,737)

Let's stretch our imagination a bit further and imagine they play for *another* century (it's only a simulation, no actual players were harmed getting these results....):

Option 1	Option 2	Option 3	Player one's winnings (losses)
0	0	0	10,052,022
1	0	0	1,558,441
0	1	0	10,105,830
1	1	0	1,407,755
0	0	1	7,739,426
1	0	1	(293,923)
0	1	1	7,865,541
1	1	1	(158,282)

Now we're getting somewhere that looks a bit more consistent. We're still getting a fair bit of variation, but the picture is becoming clearer.

Player one should *not* take option 1; if they throw a five first they should place it in the **units** spot and hope for a higher roll the second time round.

Given this, player two's best strategy on option 2 is irrelevant but they *should* take option 3 as, although in the long term they still lose money, their loss is slightly less.

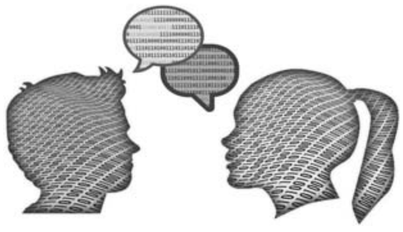
Conclusion

Using a Monte Carlo method to analyse this sort of puzzle is possible, but you have to run a lot of simulations to get consistent results. However, this is like real gambling where as this simulation showed, even a year of nightly gaming doesn't produce consistent results!

A quick search of the web on random walks will give some idea of the likely variation after a given number of rounds, and so an indication of the number of rounds you might need to run to get the desired consistency in the results. ■

Code Critique Competition 102

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'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 1 contains `sort_scores.cpp`.

```
#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';
  }
}
```

Listing 1

Critique

Felix Petriconi <felix@petriconi.net>

The main problem is that `iss.exceptions()` has a parameter of type `std::ios_base::iostate`. The underlying values are implementation defined, as stated here [1]. So it depends on the used library, if the `~iss.exceptions()` or the conversion of `"true"` works as expected.

In [2] Howard Hinnant explained that the correct parameter combination would be:

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

iss.exceptions(std::ios::failbit | std::ios::badbit) and then your program works as expected.

Beside this, here are my further observations:

The type definition of pair as **std::pair<std::string, std::string>** is from my point of view not something that I would do, because on one hand, as soon as one includes **std** into the current namespace, one gets a name collision with **std::pair**. And on the other hand, a type with the name pair says nothing about its purpose. In this case I would use a type-name like **fullname**. The first parameter of the function **read_line** is defined as call by value. There is no need to copy the value for parsing. So a call by const reference would be sufficient and would spare the copy of the complete line. Since we have today move semantics (and most compilers implement even longer return value optimization (RVO)), the readability of the code would be better if the parsed values would be returned as **std::pair<int, fullname>**. Then composing the code in a functional way would be much easier.

The current implementation of the function **read_line** has the problem that partly successful parsing of a line leads to incomplete data records. So if e.g. parsing the score value was successful, but then reading of the first name fails, the map would contain a probably not usable data set. If parsing of the line would be separated from storing the values in the map, one could avoid this kind of error.

The code uses **std::map** for two purposes as far as I can see: The data sets are ordered by the score value and in case of double entries the last entry wins. Here the problem of not separating the parsing from storing the data becomes more prominent. Think about the situation that a first data set for score value 42 was read successfully and then later an other data set with the same score value 42 from the input stream is malformed in a way that the 2nd string could not be read. The record in the map would contain information from the 1st successful read action and the first name of the second read attempt. If only fully read data sets would be inserted into the map, then this could not occur. The **std::map** has another problem: It is a data structure with very bad cache locality which results in large collections in bad performance. Each insert is a candidate for re-balancing the underlying tree which is costly. Alex Stepanov goes into details in his course at A9, that is available under [3]. So I would follow the general advice, use **std::vector** with **std::pair<int, fullname>** as value type and append all values. If you know that all score values appear just once, then simply use **std::sort** with a less predicate on the score value. If score value may appear multiple times, then I would use first a **std::stable_sort**, that keeps the order of equivalent values, and then remove all duplicate values, either keep the first occurrence or the last occurrence of equivalent entries.

The **catch(...)** catches all exceptions. We normally use this 'wildcard' only in **main()** as a fallback. In this case you know that **std::ios_base::failure** is thrown and so I would just catch this one. If any other exception is thrown, e.g. **std::bad_alloc**, I would either handle it separately, if possible or let it go and let the application terminate. If **std::bad_alloc** would be thrown in this example, you would catch it with **...** and then continue parsing, even the probability is high, that the next parsed line throws the same exception again.

In the last range-based **for** loop I would not take **entry** as an **auto&&**. **auto&&** is a forwarding reference, which does not make sense in this context. I would write the loop as **for (const auto& entry : mmap)**.

For better readability I would indent the **try - catch** block below the **while** loop. At the first glance there is the **while** loop and after it, the **try-catch** block. But the **try-catch** block actually is inside the loop.

References

[1] http://en.cppreference.com/w/cpp/io/ios_base/iostate
[2] http://stackoverflow.com/a/16473878
[3] https://www.youtube.com/playlist?list=PLHxtyCq_WDLXryyw91lahwdtpZsmo4BGD

James Holland <James.Holland@babcockinternational.com>

The student was very close to getting the program working. The problem lies with the function that was causing the student so much trouble, namely **exceptions()**. The comment next to the parameter of **exceptions()** would suggest that the student thought that the parameter type was of type **bool**; we either want exceptions to be thrown, or we do not. This is not the case. The parameter type is **iostate** and provides more flexibility in controlling stream exceptions.

The C++ standard does not define the underlying type of **iostate**; it is left to the compiler implementer. This accounts for different behaviours of the various compilers used by the student when attempting to pass **true** to **exceptions()**. The gcc compiler uses an underlying type to which the type **bool** cannot be converted and so an error message is emitted. Some compilers may, conceivably, use an underlying type that permits a **bool** to be used, such as an **unsigned int**. A value of **true**, when converted to an **unsigned int**, will have a value of **1**. This will enable one of the stream states to throw an exception and it may not be the one required.

The student also experimented with passing to **exceptions()** the bit-inverted value of the currently enabled exceptions. As the default state of the stream is to throw no exceptions, the effect of the call will be to enable all exceptions. Although this will compile without error, it is probably not what the student intended.

An exception can be thrown when the end-of-file is encountered, when a read or write operation fails, and when something more serious goes wrong with the file system. A set of constants is provided, of type **iostate**, that can be used as parameters of **exceptions()** to enable or disable the desired exceptions. The constants are listed below.

Constant
std::ios::goodbit
std::ios::eofbit
std::ios::failbit
std::ios::badbit

It is, perhaps, a little confusing that **goodbit** is so named as it does not represent a bit position. It has a value of zero and so can be used to 'represent' the absence of the other bits. The constants can be combined, using the bitwise operators, to enable more than one exception, if required.

The student states that at least one character is required after the second name for the software to work as required. I suspect this is because the student's program has the **eof** exception enabled. When **operator>>()** reads the second name it needs to be sure that it has read all the characters representing the name. It can only do this by attempting to read one character beyond the end of the name. If there are no characters beyond the name, the end of the buffer will be encountered and **eofbit** set. Should the **eof** exception be enabled, as is the case with the student's code, an exception will be thrown. If there is a space, for example, after the second name, **operator>>()** will read the space and conclude that all the characters of the second name have been read. There is no need to read any more characters and so no attempt is made to read beyond the record; **eofbit** will not be set and no exceptions will be thrown. The record will be processed as required.

The student also stated that the program does not correctly handle records with invalid scores. In such cases, **operator>>()** attempts to read a numerical value but fails, setting **failbit** and writing zero into **score**. As a result no exceptions will be thrown because only the **eof** exception has been enabled. An entry in the map will then be made with a key of zero. The program then attempts to read the first and second names from the stream. As the stream is not in a good state, the read operations will not have any effect thus leaving the map entry with null value for the two names.

From this analysis, it can be seen that the program needs to be amended in two ways; one to prevent exceptions being thrown on encountering the **eof**, and the other to handle invalid scores. Both can be achieved by allowing an exception to be thrown only when a formatting error occurs. This is done by passing **std::ios::failbit** to **exceptions()**.

Although this modification results in a working program, I suggest, some areas of the design could be improved. It would be better, for example, to read the score, first name and last name into their respective variables before entering them into the map. This will ensure the map does not contain records that are not properly formed. If the student is concerned that this will introduce inefficiencies, use could be made of the move semantics of **std::string** (since C++11). This is achieved by using **std::move()** in the assignments to the map. After the assignments, the strings **first_name** and **last_name** will be in a valid but unknown state. This is acceptable as the string variables will no longer be used. It should be noted, however, that some string implementations use what is called small string optimisation for strings of 15 characters or less. In such cases, moving short strings will not be any faster than copying them; but I digress.

It is noticed that the student has used an rvalue reference (**&&**) in the **for** loop that prints the content of the map. This is legal but not necessary for non-generic code. A simple reference (**&**) will do.

Placing the incrementing of **line** within the **try** block is not required as incrementing an **int** will not throw an exception. Simply moving the incrementing statement outside the **try** block will alter the program logic as the **while** loop does not have separate scope defining braces and, instead, relies on the fact that the **try** and **catch** blocks are, in effect, one compound statement. I have chosen to add braces to the **while** statement and take the incrementing outside the **try** block. I think this makes things a little clearer.

It might be worth making variable names a little more descriptive. It is not absolutely clear what **lbufr** means, for example. Perhaps something like **line_buffer** would be more appropriate. Also, I am not entirely sure what **mmap** stands for.

Finally, I include my version of the program.

```
#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(std::ios::failbit);
  int score;
  std::string first_name;
  std::string second_name;
  iss >> score;
  iss >> first_name;
  iss >> second_name;
  auto & name = mmap[score];
  name.first = std::move(first_name);
  name.second = std::move(second_name);
}
int main()
{
  std::map<int, pair> mmap;
  std::string lbufr;
  int line = 0;
  while (std::getline(std::cin, lbufr))
  {
    ++line;
    try
    {
      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';
  }
}
```

Simon Sebright <simonsebright@hotmail.com>

It's been a while since I had a go at the critique and a longer while since I have done any C++. The subtleties of the standard and the language will have to be sacrificed!

I compiled the code using the free Visual Studio edition. I got slightly different results from the student, but similar in form. My value for the score of XX was -858993460. This suggests we are in the realms of undefined behaviour. The first three lines all gave me an exception. Note that I pasted the `scores.txt` from the website and tidied up the result a bit – perhaps there were deliberately some extra characters hanging around there. I had each line end with the last character of the second name.

There are many levels to critique such a short piece of code; I'll spew it out in the order it comes to me...

The first thing that strikes me is the lack of overall algorithm structure. I would have expected:

```
instantiate some collection to store results
for(each line in input)
  add to the results
Sort Results
Output Results
```

The bad naming of the **read_line** function contributes to this – it doesn't read a line, it converts a line to a result and adds it to the collection. **add_to_results(...)** would be a better option. Using a map keyed on integer to sort the results automatically is neat, but it is not explicitly mentioned in the code, that this is sorted. **sorted_results** would be a better name than **mmap**, which says nothing.

So, having dealt with the overall structure, let's look at some more detail. First, the catch-all exception handling is a bad idea. The first thing I did to change the code was to add **#include <exception>** to the top and to catch **exception& e**, then output **e.what()** in the result:

```
Line 1 ignored because of ios_base::eofbit set:
iostream stream error
```

So, now we have some more information – it threw because it ran off the end of the stream whilst reading the second name (I debugged it a bit to see that, and also we note that the second names were gathered, so it must have got to the end of them). OK, well that's not surprising because the string that was passed to **read_line()** did indeed end abruptly with the last character of the second name. Odd behaviour that it sent the result back, but maybe that's the standard (out of scope here).

So why did that occur? Well, the student attempted to get the input stream to throw exceptions when something went wrong – presumably because they wanted to trap the case where a non-number was passed as the score. Calling **basic_ios::exceptions()** is a way to accomplish this, but what about that parameter – **true**? If you look at the documentation you will see that the function is expecting an **int** which is an ORed set of flags for the various exceptions to be thrown. But we are passing **true**, which isn't an **int**, but can be implicitly converted to one. On my compiler, it is 1. This corresponds to the **std::ios::eofbit** value, so in fact we are only asking the stream to throw when it runs off the end of the stream, which is not what we want. The values of this enumeration are: **badbit**, **eofbit**, **failbit**, **goodbit**.

If we change the **iss.exceptions()** call to take **ios::failbit** instead, then we get:

```
Line 4 ignored because ofios_base::failbit set:
iostream stream error
32: Roger Orr
34: Alison Day
45: John Smith
```

This is the desired result, as it did indeed trap the case of **XX** not being an integer.

So what can we learn here? First, structure and naming is important. Second, make sure you pass suitable parameters to functions avoiding implicit type conversion (I am out of date with compiler settings – perhaps this is a warning one can induce and prevent by building with warnings as errors set). Third, avoid catch-all exception handling.

As an aside, the decision to take a name as two strings – first and second names – is bad. Firstly, consider Mary Chapin Carpenter or Frank Lloyd Wright. Secondly, not all cultures have the 'given name' first – in China for example, this comes after the 'family name'. A lot of work has been done to establish suitable standards for such concepts. In this case, a single string comprising the full name would have sufficed, but then of course the string parsing might have been trickier.

Indeed it should be said that piggy-backing the stream's string parsing functionality was a good idea, but perhaps not as plain sailing as one hoped!

Paul Floyd <paulf@free.fr>

This is a small piece of code, barely over 50 lines long. What can I say about it? First a minor nit, as a matter of course (or habit) I would pass the string argument to **read_line** as a const reference. Clearly the grist of the matter is the **std::istringstream** exception specification. I don't remember by heart what the parameters to the setter are, but it certainly looks like the author of the code was guessing at what to use.

When I compiled the code I got:

```
CC -std=c++14 +w2 -g -o cc101 cc101.cpp
"cc101.cpp", line 17: Error: Could not find a
match for std::ios::exceptions(std::istringstream,
bool) needed in read_line(std::string,
std::map<int,std::pair<std::string, std::string>
>&).
1 Error(s) detected.
```

So I hit the context lookup of my IDE and I got the following 3 definitions:

```
void
  exceptions(iostate __except)
  {
    _M_exception = __except;
    this->clear(_M_streambuf_state);
  }

typedef _Ios_Iostate iostate;

enum _Ios_Iostate
{
  _S_goodbit              = 0,
  _S_badbit               = 1L << 0,
  _S_eofbit               = 1L << 1,
  _S_failbit              = 1L << 2,
  _S_ios_iostate_end = 1L << 16
};
```

So it looks like the first compiler accepts a conversion from **true** (**bool**) to **iostate**, presumably to the value **1** or **badbit**. And it also looks like GCC accepts the bitwise **not** which I expect sets all of the bits to **1**.

I guessed that the intention was to turn on all exceptions, so I tried:

```
iss.exceptions(std::ios_base::badbit |
  std::ios_base::eofbit |
  std::ios_base::failbit);
```

This then compiled and gave the behaviour described in the description. At first I didn't pay attention to the trailing dot on first line of the input, so I couldn't see why the first line wasn't throwing and the others were. The moral is to always read carefully.

Then I had a go at debugging. I generally had a hard time after the first exception was thrown. Either the code would run to completion (**dbx**) or would seem to get stuck reading lines (**lldb**). To try to make things a bit clearer I decided to make the catch a bit more explicit:

```
{
std::cout << "Exception: " << ex.what()
  << " line:" << line << " ignored\n";
}
catch (...)
{
  std::cout << "Other exception, line:"
    << line << " ignored\n";
}
```

This helped a bit, but not as much as I'd hoped:

```
Exception: basic_ios::clear line:4 ignored
```

OK, but I was kind of hoping that it would be more explicit about the cause of the exception.

I also had a look online at http://en.cppreference.com/w/cpp/io/basic_ios/exceptions

Though it doesn't seem to be the case here, the note on how different implementations set the state was quite interesting.

By now I'd figured out that the problem is that even for the 'well-formed' lines, an exception is being thrown when the end of the **istringstream** buffer is reached. I don't consider reading to the end of a buffer to be an exceptional circumstance, at least not in this case, so I removed **eofbit** from the call to set the exceptions flags, and then everything worked as expected.

As a final check, I tried wrapping the body of **read_line** in a **try**/**catch** to see the **rdstate()**, here just the **catch** clause:

```
catch (...)
{
  std::cout << "iss mask " << iss.rdstate()
    << "\n";
}
```

This confirmed that the first two exceptions were **eofbit** and the third a **failbit.**

Commentary

There seem to be few people who use exceptions with iostreams; perhaps this code critique helps to explain why this is so.

The first problem is with the definition of **iostate** which the C++ standard defines to be a 'bitmask type'. These can be implemented as

- an enumerated type that overloads certain operators,
- an integer type, or
- a bitset

It is important to restrict portable code to operations that are valid for all three types of implementations. In the original code, passing **true** to **iss.exceptions()** only compiles when **iostate** is an integer type (as it is for MSVC, but not for g++). It sets the flag corresponding to **1** which will be **eofbit**.

Secondly there are the semantics. Many people do not consider end of file to be unexpected (most files have an end!) and so do not want to enable exceptions with **eofbit**. While the standard does attempt to specify the different behaviour of **badbit** and **failbit** the distinction seems quite subtle and the interaction between the two is messy. Typically, as Felix noted, you would set both these bits together.

Also note that **goodbit** is not actual a bit value, as James pointed out. It is **zero**, so code testing for a 'good' state cannot use bitwise and, as the expression **`(ios.rdstate() & iostate::goodbit)`** is always **`false`**!

Finally, while four values are defined, the actual *implementation* of the bitmask type might contain other values with non-portable sematics, so it may be important to make sure you don't accidentally set the extra bits. The original example did this in the code selected with **`__GNUC__`** as the expression **`~iss.exceptions()`** may modify bits other than those defined by the standard values.

As the critique also shows, another problem with using exceptions with iostreams is that the operations do not provide the strong exception guarantee: if an exception is thrown when reading into **`name.second`** the string value might already have been modified. In this case though the original code is already not strongly exception safe (as has already been observed in the critiques) as the entry is added to the map even if the streaming operations throw an exception.

One aspect of the code that no-one fixed was that the program is trying to sort people based on a score but if two people have the *same* score the second entry *overwrites* the first one. I had hoped the opening sentence implied that all the scores were important and should be retained! The easiest change to resolve this is to replace **`std::map`** with **`std::multimap`** throughout; the variable name **`mmap`** was a bit of a hint (!) (Other than that it's a poor name for the variable.)

The winner of CC 101

I think the four critiques covered the ground well – both the low level syntactic and semantic problems and also some discussion about the higher level design principles involved in this problem.

Felix and James both gave good explanations of the portability problems with the program's use of **`iostate`** and all the critiques fixed the main presenting problem.

Simon's critique addresses some additional design issues explicitly, such as the poor naming of both methods and variables, as well as covering some of the syntactic problems. So, although other critiques may have covered the low level issues in slightly more detail, I have decided the award the prize for this critique to Simon.

Code critique 101

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

> I am trying to parse a simple text document by treating it as a list of sentences, each of which is a list of words. I'm getting a stray period when I try to write the document out:
>
> ```
> -- sample.txt --
> ```
>
> This is an example.
>
> It contains two sentences.
>
> ```
> $ parse < sample.txt
> This is an example.
> It contains two sentences.
> .
> ```
>
> Can you help fix this?

Listing 2 contains `parse.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 2

```
#include <iostream>
#include <sstream>
#include <memory>
#include <string>
// pointer type
template<typename t> using ptr =
  std::shared_ptr<t>;

// forward declarations
struct document;
struct sentence;
struct word;
document read_document(std::istream & is);
sentence read_sentence(std::istream & is);
void write_document(std::ostream & os,
  document const & d);
void write_sentence(std::ostream & os,
  sentence const & s);

// A document is a list of sentences
struct document
{
  ptr<sentence> first_sentence;
};
// A sentence is a list of words
struct sentence
{
  ptr<sentence> next_sentence;
  ptr<word> first_word;
};
struct word
{
  ptr<word> next_word;
  std::string contents;
};

// read a document a sentence at a time
document read_document(std::istream & is)
{
  sentence head;
  auto next = &head;
  std::string str;
  while (std::getline(is, str, '.'))
  {
    std::istringstream is(str);
    ptr<sentence> s(new sentence(
      read_sentence(is)));
    next->next_sentence = s;
    next = s.get();
  }
  document d;
  d.first_sentence = head.next_sentence;
  return d;
}
// read a sentence a word at a time
sentence read_sentence(std::istream & is)
{
  word head;
  auto next = &head;
  std::string str;
  while (is >> str)
  {
    ptr<word> w(new word{nullptr,str});
    next->next_word = w;
    next = w.get();
  }
  sentence s;
  s.first_word = head.next_word;
  return s;
}
```

ACCU Information

Membership news and committee reports

View from the (no longer Acting) Chair

Bob Schmidt
chair@accu.org

This is my third View, which means we're almost halfway through the 2016–2017 term. You may have noticed that my Views have concentrated on the people who make ACCU work. We are a group run by volunteers, and our volunteers deserve to get full credit for their efforts. One of the few regular responsibilities I have is to write this article every two months; the day-to-day running of ACCU is done by the other members of the committee, and they do it well.

Special General Meeting

The Special General Meeting was held September 28th at St. Aldates Tavern in Oxford, in combination with the ACCU Oxford local group. The purpose of the SGM was to formally elect a Chair and a Secretary, so those positions would no longer be held by caretakers. I'm pleased to announce that Malcolm Noyes was elected Secretary, and I was elected Chair.

Malcolm reported that he received 133 ballots out of the 591 sent. Malcolm received 121 votes for secretary, with 12 abstentions, and I received 119 votes for chair, with 14 abstentions. Thank you to all who voted.

Twelve members attended the meeting, including officer Matt Jones and committee member Ralph McArdell, giving the SGM the needed quorum. I had planned to attend the meeting in person – it seemed like a great excuse to play hooky from work, enjoy an evening with colleagues, and take another holiday in Europe. Unfortunately, reality intruded in the form of the dreaded Jury Duty, which put an end to those plans. I was able to 'attend' via a Google Hangout, and from what I could see and hear it sounds like a good time was had by all.

Nigel Lester coordinated the SGM with the local group meeting in Oxford. He also was able to arrange sponsorship for the evening, provided by Oxford Computer Consultants. Robert Bentall arranged additional sponsorship for the meeting from Martin-Baker Aircraft Company, Ltd. In addition to sponsoring the evening's drinks, Martin-Baker provided an ejection seat for 'testing' by the attendees.

Please join me in thanking the sponsors, Robert, and Nigel for an enjoyable evening.

Diversity statement

We published the draft Diversity Statement in the last issue of *CVu*. For those of you following along, here it is again:

> 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 absence 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.

Please note that based on a comment received after the last *CVu* was published, I have changed one word in the statement. The clause 'religion (or lack thereof)' has been changed to 'religion (or absence thereof)'. The comment

Code Critique Competition (continued)

Listing 2 (cont'd)

```
// write document a sentence at a time
void write_document(std::ostream & os,
  document const & d)
{
  for (auto s = d.first_sentence; s;
       s = s->next_sentence)
  {
    write_sentence(os, *s);
  }
}

// write sentence a word at a time
void write_sentence(std::ostream & os, sentence
const & s)
{
  std::string delim;
  for (auto w = s.first_word; w;
       w = w->next_word)
  {
    std::cout << delim << w->contents;
    delim = ' ';
  }
  std::cout << '.' << std::endl;
}

int main()
{
  document d(read_document(std::cin));
  write_document(std::cout, d);
}
```

noted that ‘lack’ could be seen as a pejorative, and that ‘absence’ is more neutral.

This draft is still open for comments. We will keep the draft open for comments for another publishing cycle, and target the end of the year for approval by the committee.

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.

In moving the site, rather than just taking a backup of the existing host and restore onto the new host, he took the opportunity to do a little spring cleaning. The new host is a virtual host courtesy of Bytemark (www.bytemark.co.uk), and runs Debian Stable. Jim adapted the configurations to Debian from the previous CentOS+Plesk setup, and archived a lot of files that didn’t seem to be used any more.

One notable change is that the website is now accessed with HTTPS only, fixing a long-standing security flaw that user login credentials were visible on the wire. For the mailing lists, there has been a very slight minor version downgrade of the mailing list manager, Mailman, as Jim moved to the Debian packaged version. He also moved some lesser known older functions such as an old wiki to *.accu.org subdomains, and ensured they too are secure.

The domain registration for accu.org was also close to expiry, so Jim moved the domain hosting to Bytemark and renewed it. It’s now easy to set up new subdomain websites. As an example, Russel Winder is doing great work with conference.accu.org, which is hosted on the same system but uses a completely different web framework. (More on Russel’s work below.)

After running the new site under test.accu.org for a few weeks, on the morning of the IETF 96 Hackathon Jim diverted himself by updating the database from the old host, copying over any recently updated files from the website and the (private) mailing list archives, and changing the DNS to point to the new host. He was expecting a torrent of breakage, but thankfully only a few minor issues came up.

Jim is still working on some minor details, but the new hosting seems to be running well. ACCU now has an excellent hosting basis for further changes.

This is a condensed version of the work Jim performed on the migration. I have it on good authority that he is working on a more complete, first-person version for publication in the near future. In the interim, please join me once again in thanking Jim for all his hard work.

Conference web site improvements

Our conference chair, Russel Winder, has been busy working on a new web site for the conference.

The experimental site is located at https://conference.accu.org. It is intended that this will be a Python 3/Flask/SQLAlchemy/SQLite site handling session proposals submission, review selection, scheduling, and program publishing. A blog and static pages will be managed by Nikola. In his words:

> We are going to be extraordinarily agile here: release early, release often. So whilst the Flask stuff is being developed the Nikola stuff will be published, indeed already is. We need volunteers to help with styling, imagery, and indeed content. The current display is the default Bootstrap generated by Nikola, we need to create a styling that is obviously the ACCU brand, but is definitely the conference and not the main website. I am sure the AYA folk will chip in, but so should ACCU members. The conference is a joint venture so everyone has a vested interested in making the website as stonkingly brilliant as possible.
>
> The source for the Nikola managed static material (yes, blogs are static material!) is on GitHub at https://github.com/ACCUConf/ACCUConfWebsite_Static. The master branch is all the general material that is the same for all years. For each year there is a branch, this year accu2017, where all the year specific material goes.

I have no idea what stonkingly means (there’s that whole ‘UK and US are two countries separated by a common language’ issue cropping up again) [1], but I’m sure the new conference web site will, indeed, be brilliant. Contact Russel if you would like to help.

Committee spotlight

Since Jim Hague volunteered to perform the website migration, he has been working as a co-opted member of the committee. Jim first learned C as a postgrad back in 1984. He bought edition 1 of *The C++ Programming Language* when it was first published, and has followed the C and C++ world ever since. He has enjoyed a 25-plus year career as a jobbing programmer, during which he’s worked on projects from an embedded JVM to air traffic control systems. He discovered the ACCU conference in 2002, was bowled over to be able to learn about C++ (and lots more besides) from the actual people doing the steering, and gradually got sucked in. He founded the Oxford local group and ran it for the first few years, and will tell anyone interested in setting up a local group that it’s not that difficult and they should definitely do it.

Call for volunteers

I'm happy to report that Russel Winder has filled out his conference committee. Thank you to everyone who volunteered.

We still have several open positions:

- We still need an additional auditor to fill out my term. This is a low commitment position that should take only a few hours of your time in late March or early April. Please send me an email if you are willing to join Guy Davidson on the auditing team.
- Jim Hague reports that the ACCU web site uses Xaraya, a PHP framework that has been moribund for the last 4 years at least, and a replacement is overdue. He regrets he has neither the time nor the skills necessary for that, and hopes one or more volunteers with the appropriate skills will step forward.
- We are hoping to recruit someone to assist Martin Moene with the web site sys admin duties. Please contact me if you are interested.

Journals

In the past two months *CVu* and *Overload* have published articles by Silas Brown (three!), Thaddaeus Frogley, Pete Goodliffe, Sergey Ignatchenko, Baron M (a nom de plume, perhaps?), Chris Oldwood (two), Adam Tornhill, Jonathan Wakely (two), and Russel Winder. In addition, Roger Orr set and moderated the Code Critique Competition; and our intrepid editors, Fran and Steve, provided their editorials and produced top-notch publications. Please consider joining this august group of authors by contributing an article.

Note

[1] Yes – I know – Google is my friend: *stonk* – to bombard with concentrated artillery fire; *stonking* – used to emphasize something remarkable, exciting, or very large; *stonkingly* – something that is exceptionally good. One of these is not like the others.

You've developed some great software.

You've worked with various stakeholders, and are sure it does what the business wants and needs... which means your application isn't going to be exactly like the one they had before.

Good instructions – whether a manual, online help or training materials – are a helping hand while people get comfortable with changes, offering time-saving tips and providing a feeling of support and security.

Clearly stated...

It's not what we do that's important...
it's what we can help your customers do.

Get in touch for an informal discussion on how we can help you:

T 0115 8492271

E info@clearly-stated.co.uk

W www.clearly-stated.co.uk

GET MOORE

intel Software Elite Reseller

TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER

Take your results to the next level with screaming-fast code.

Intel® Parallel Studio XE

www.qbssoftware.com/parallelstudio
020 8733 7101 | sales@qbssoftware.com

qbs SOFTWARE