the magazine of the accu

www.accu.org

{cvu}

Volume 25 • Issue 6 • January 2014 • £3

{cvu}

Volume 25 Issue 6
January 2014
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Pete Goodliffe, Jez Higgins, Chris Oldwood, Roger Orr, Richard Polton, Mark Radford, Charles Tolman, Adam Tornhill, Vsevolod Vlaskine

ACCU Chair
Alan Griffiths
chair@accu.org

ACCU Secretary
Giovanni Asproni
secretary@accu.org

ACCU Membership
Mick Brooks
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

On Being Ignorant

Abstraction is selective ignorance
~ Andrew Koenig

Last time I briefly opined about interface names, and the importance of good abstractions. The name of a thing is more than just a tag to identify it; well-named things describe their purpose in a simple, succint way. In a complex system, this can be vital to the programmer(s) maintining it. Well-chosen names for roles in a program allow the developers and maintainers to have a high-level – *abstract* – understanding of the relationships between those roles, and can concentrate on having a deep knowledge of all the nuances of just a small bit, under the hood. If those programmers all needed to know everything about every implementation of every interface or type, the volume of information in even quite small systems can easily become overwhelming.

Good names also form a language, in much the same way that software Patterns do, and thereby create a shared vocabulary about a system that the developers can use to communicate with each other about it at a high-level. Of course, this isn't limited to software. Without abstraction, we'd have difficulty communicating with each other about *anything*! Consider the simple task of arranging to meet someone somewhere: the names we give places are themselves abstractions, the time of day is a pretty abstract idea, the act of travelling (Walk? Cycle? Public transport? Swim?) is an implementation detail, and even the idea of 'walk' is an abstraction over the various muscle movements, force exertion, etc. not to mention the molecular level interactions...

This idea of shared abstractions is partly about communication, but also about simplicity. If you had to concentrate on every atom in your brain to think, you'd have little bandwidth to do the thinking that necessitated it. So it is with the names of interfaces, functions, classes, variables and all the other things that go into programming. Being able to *ignore* the fine detail is what makes programming complex systems possible.

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 26.1: 1st February 2014
C Vu 26.2: 1st April 2014

Overload 120: 1st March 2014
Overload 121: 1st May 2014

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 speaks on communication.

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 multi-faceted.

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.

Since we are not talking in the CPU's mother tongue, 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 yourself 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 chose to use, and in how we use them. Are you using a language that allows you to naturally express the concepts you are modelling?

Obviously, the entire team working on a section of code must be coding in the same language. It's not a good idea to add lines of Basic code to a Python script. If your application is written in C++, the whole team must all be using C++ or it wouldn't work. (Yes, someone could be coding C and they might get away with it, but the code will suffer.)

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++ *vs* '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 where the back-end 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

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

another language in play: the language 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 concious choice to write all code in English: all variable names, comments, class/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.

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.

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?

Interpersonal communication

Electric communication will never be a substitute for the face of someone who with their soul encourages another person to be brave and true.

~ Charles Dickens

We don't just communicate by typing code. Programmers work in teams with other programmers. And with the wider organisation.

There's a lot of communication going on here. Because we're doing this all the time, high quality programmers *have* to be high-quality communicators. We write messages to, speak with, even gesticulate at, others all the time.

Ways to converse

There are many communication channels we use for conversations, most notably:

- talking face-to-face
- talking on the phone, one-to-one
- talking on the phone in a 'conference call'
- talking on VOIP channels (which isn't necessarily different from the phone, but is more likely to be hands-free and allow you to send files over the same communication channel)
- email
- instant messaging (e.g. typing in Skype, on IRC channels, in chatrooms, or via SMS)
- video conferencing
- sending written letters via the physical postal system (do you remember that quaint practice?)
- fax (which has largely been replaced by scanners and common sense; however it still has a place in our comms pantheon because it is regarded as useful for sending legally binding documents).

Each of these mechanisms is different, varying in: the locations spanned, the number of people involved at each end of the communication, the facilities available and richness of interaction (can the other person hear your tone of voice, or read your body language?), the typical duration, required urgency and deferrability of a discussion, and the way a conversation is started (e.g. does it need a meeting request to set up, or is it acceptable to interrupt someone with no warning?).

They each have different etiquettes and conventions, and require different skills to use effectively. It is important to select the correct communication channel for the conversation you need to have. How urgent is an answer? How many people should be involved?

Don't send someone an email when you need an urgent answer; email can sit ignored for days. Walk over to them, ring them, Skype them. Conversely, don't phone someone for a non-urgent issue. Their time is precious, and your interruption will disrupt their *flow*, stopping them from working on their current task.

When you next need to ask someone a question, consider whether you are about to use the correct communication mechanism.

Master the different forms of communication. Use the appropriate mechanism for each conversation.

Watch your language

As a project evolves it gains its own dialect: a vocabulary of project and domain specific terms, and the prevalent idioms used to design/think about the shape of the software design. We also settle on terminology for the process used to work together (for example, we talk about *user stories*, *epics*, *sprints*).

Take care to use the right vocabulary with the right people.

Does your customer need to be forced to learn technical terms? Does your CEO need to know about software development terminology?

Body language

You'd be upset if someone sat beside you, sparked up a conversation, but spent the whole time facing in the opposite direction. (Or you could pretend they were from a bad spy movie; "I hear the gooseberries are doing well this year... and so are the mangoes" [1]).

If they pulled rude faces every time you spoke, you'd be offended. If they played with a Rubik's cube throughout the conversation you'd feel less than valued.

It is easy to do exact this when we communicate electronically; to not fully respect the person we're talking with. On a voice-only conversation, it's easy to zone out, read email, surf the web, and not give someone else your full attention.

Having fully embraced our modern, always-connected, broadband age, I now default to selecting a *video-on* communication channel. Often I'll kick off a conversation that might have been via phone or instant message with a VOIP video chat. Even if my conversant will never enable their own video, I like to broadcast a picture so that my face and body language are clearly visible.

This shows I'm not hiding anything, and fosters a more open conversation.

A video chat forces you to concentrate on the conversation. It engages the other person more strongly, and maintains focus.

Parallel communication

Your computer is having many conversations at once: talking to the operating system, to other programs, device drivers, and other computers. It's really quite clever like that. We have to make sure that *our* code communication with it is clear and won't confuse matters whilst it's having conversations with other code.

That's a powerful analogy to our inter-personal communication. With so many communication channels available simultaneously, we could be engaging in office banter, instant messaging a remote worker, and exchanging SMSs with our partner, all whilst participating in several email threads.

And then the telephone rings. Your whole tottering pile of communication falls over.

How do you ensure that each of your conversations is clear enough and well-structured so it won't confuse any other communication you're concurrently engaged in?

I've lost count of the number of times I've typed the wrong response into the wrong Skype window and confused someone. Fortunately, I've never revealed company confidential information that way. Yet.

Effective communication requires focus.

Talking of teams

Communication is the oil that lubricates teamwork. It is simply *impossible* to work with other people and not talk to them.

This, once more, underscores Conway's Law. Your code shapes itself around the structure of your teams. The boundaries of your teams and the effectiveness of their interactions shapes, and is shaped by, the way they communicate.

Good communication fosters good code. The shape of your communications will shape your code.

Healthy communication builds comradery, and makes your workplace an enjoyable place to inhabit. Unhealthy communication rapidly breaks trust and hinders teamwork. To avoid this, you must talk to people with respect, trust, friendship, concern, no hidden motives, and a lack of aggression.

Speak to others transparently, with a healthy attitude, to foster effective teamwork.

Communication within a team must be free-flowing and frequent. It must be normal to share information, and everyone's voice must be heard.

If teams don't talk frequently, if they fail to share their plans and designs, then the inevitable consequences will be duplication of code and effort. We'll see conflicting designs in the codebase. There will be failures when things are integrated.

Many processes encourage specific, structured communication with a set cadence; the more frequent the better. Some teams have a weekly progress meeting, but this really isn't good enough. Short *daily* meetings are far better (often run as *scrums*, or *stand-up* meetings). These meetings help share progress, raise issues, and identify roadblocks without apportioning blame. They make sure that everyone has a clear picture of the current state of the project.

The trick with these meetings is to keep them short and to-the-point; without care they degrade into tedious rambling discussions of off-topic issues. Keeping them running on-time is also important. Otherwise they can become a distractions that interrupt your *flow*.

Talking to the customer

There are many other people we must talk to in order to develop excellent software. One of the most important conversations that we must hold is with the customer.

We have to understand what the customer wants, otherwise we can't build it. So you have to ask them, and work in their language to determine their requirements.

After you've asked them once, it's vital to keep talking to them as you go along to ensure that it's still what they want, and that assumptions you make match their expectations.

Only way to do this is in their language (not yours), using plenty of examples that they understand: demos of the system under construction.

Other communication

And still, the programmer's communication runs deeper than all this. We don't just write code, and we don't just have conversations. The programmer communicates in other ways. For example, by:

- writing code documentation
- writing specifications
- blogging
- writing magazine articles

How many ways are you communicating as a programmer?

Conclusion

First learn the meaning of what you say, and then speak.
~ Epictetus

A good programmer is hallmarked by good communication skills. Effective communication is:

- clear
- frequent
- respectful
- performed at the right levels
- using the right medium

We must be mindful of this, and *practise* communication – seek to constantly improve in written, verbal, and code communication. ■

References

[1] 'Secret Service Dentists', Monty Python's Flying Circus

Questions

1. How does personality type affect your communication skills? How can an introverted programmer communicate most effectively?
2. How formal or causal should our interactions be? Does this depend on the communication medium?
3. How do you keep colleagues abreast of your work without endlessly bugging them about it?
4. How does communication with a manager differ from communication with a fellow coder?
5. What kind of communication is important to ensure that a development project runs successfully?
6. How do you best communicate a code design? They say a picture speaks a thousand words. Is this true?
7. Do distributed teams need to interact and communicate *more* than co-located teams?
8. What are the most common barriers to effective communication (e.g. different assumptions, language barriers)?

Social Networking

Chris Oldwood describes more of the tools of his trade.

Right back at the very beginning of this series of articles [1] I talked about how I had found great utility in chat-style software as an alternative for email to exchange brief messages with my team-mates about day-to-day events. Those messages are generally asynchronous in nature and relate to the events going on around me, such as broken builds, production issues, the forthcoming need for coffee, etc. But when it comes to aspects of software design the need to converse with a human on a less constricted form usually takes over. Depending on who's around at the time it might mean leaning across a desk, walking down the aisle or putting a metaphorical pen-to-paper and banging out a thought-inducing email (largely to self, but with the ability for others to provide feedback).

In the 'good old days' they were pretty much your only choices, but the modern era of smart-phones and ubiquitous internet access means I'm always within 'close' reach of a whole bunch of clever people; some that I already work with but far more that I don't. This means I can tap-in to the 'wisdom of crowds' to some degree and perhaps get feedback in the intervening time before my colleagues are once again around me and able to provide that high-bandwidth communication channel I need to drill down to the finer details.

Of course dial-up forums like CompuServe and Cix were around long before the likes of Twitter and Facebook, as were the Internet-based online forums such as the Usenet. However, although the readership was large, it varied widely and although I made some metaphorical friends the transport mechanism was not really up for the social banter that adds a little more character and background to its users. In fact culturally it was a faux pas to add mindless chatter because it consumed bandwidth and back then bandwidth cost serious money when all you had was a modem measured in kilo-bits per second. In stark contrast, services like Twitter with its so-called 'micro-blogging' format means that the content length is purposefully constrained and probably means that the transport overhead now dominates the message size!

Distilling a problem down to its essence is a tricky problem but also a worthy pursuit ... you may even answer your own question just by going through the motions

It seems ironic after the years of worrying about keeping 'on topic' to purposefully choose to use a service that restricts your freedom. But what I've found with the likes of Twitter is that I can gauge opinion about even very small matters so long I can squeeze it into 140 characters or less. I can on occasion stretch to 280 if I want to try the patience of my audience, but hopefully they'll forgive me if it's interesting.

Distilling a problem down to its essence is a tricky problem, but also a worthy pursuit in its own right as you may even answer your own question just by going through the motions. The fear of public ridicule can be a sobering thought too. Occasionally I've attempted to frame a question as a rant to see if I get a 'me too' style response or just a 'look' of derision. When you work in a very small or very inexperienced team you sometimes need to look outside for validation, or a friendly ear. It probably sounds a little dysfunctional as I should be able to address concerns about my colleagues and their work directly to them, but in the early stages I like to keep 'the air clear' between us and get a reality check from a third party first about whether it's a mountain or a molehill.

Social Networking tools have embraced the whole push model so that you get a stream of collective consciousness delivered directly to your device. Assuming that you're one of those people who doesn't just follow everyone in the world and are interested in keeping the data stream down to a manageable level, there is a continual diet of knowledge and sage advice to be had. That insight may come in the form of a quote from such a luminary as Alan Kay or Sir Tony Hoare. Or it may come in the thinly veiled guise of 'geek humour' such as an XKCD [2] cartoon. Or it might be an off the cuff remark from one of your fellow ACCU colleagues.

My own social networking persona is very much for the purposes of my professional programming career. Although I clearly make out-of-band comments, the group of people I follow and the content I generally consume is of a programming related nature. Personally I'm not interested in trying to maintain a separate 'private' persona, but I know of people that do – so that they can keep their work and personal lives separate as a way of managing the onslaught of data. Despite probably overdoing it myself on occasion I do try and stick with those people that provide the highest signal-to-noise ratio. Tooling has made a big difference here by filtering messages in an intelligent way so you don't get cross-talk.

The links to blog posts and articles I receive are often very useful, but I have to squirrel them away with a bookmark and save them for later consumption. No, what I really enjoy most are the little pearls of wisdom that act as continual reminders to think about what we're doing. Even with the best intentions I lose focus and find myself slipping back into dubious habits or not bothering to think hard enough about what the best name might be for a class or method.

One time I remember lazily naming a class 'EngineManager'. Before I had committed it though a tweet (from Allan Kelly, I think) appeared commenting on the nondescript nature of the term 'manager' and so I renamed it 'EnginePool' as that felt a better description because it followed the Pool pattern from the POSA 3 book [3]. As if by coincidence another tweet from Nat Pryce arrived bemoaning the use of pattern names in classes. Oops.

The practice Nat Pryce appeared to be commenting on is a long-standing joke in Java where there has been a habit of seeing how many Gang of Four pattern names you can squeeze into one class. In my case I felt I was quite justified in naming my class with the 'Pool' suffix as it documented the expected behaviour nicely. The point though is that it made me stop and think.

And that's what I enjoy most about having a stream of one-liners flowing though my consciousness every day, they keep me honest and remind me of the many forces that I need to keep in balance with every line of code I write. ■

CHRIS OLDWOOD

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

References

[1] 'In the Toolbox: Team Chat', *C Vu*, May 2013
[2] http://xkcd.com
[3] *Pattern-Oriented Software Architecture. Volume 3: Patterns for Resource Management* by Michael Kircher and Prashant Jain (2004).

The Soundtrack to Code

Adam Tornhill takes a look at the relationship between music and concentration.

To a programmer, noisy work environments are a devastating killer of job performance. Surprisingly, most workplaces still seem to ignore the problem and leave the root causes untreated. Instead the symptoms of a deep, severe problem are left to the individual programmers to address. A typical treatment involves earphones with music used to shield out the noise. This choice creates a personal auditory backdrop that may relieve the immediate stress but it comes at a potential cost. While music doesn't necessarily have a negative impact on our performance, it's important to understand how the degree and direction of the effect varies with the task.

The perils of noise

A quarter century ago, the seminal Peopleware made the case against dysfunctional office environments. In a comprehensive study on the correlation between noise levels and work performance, Peopleware reported that individuals in quiet working conditions were ⅓ *more* likely to deliver zero-defect work than their unfortunate peers in noisy environments. Even more interesting was that the correlation was stronger for increased levels of noise [1].

The results from Peopleware have been supported by multiple studies. They should be an alarming message to any company that depends upon the creativity, problem solving skills and quality of its employees. Translated to money, a ⅓ increase in zero-defect work should be a no-brainer investment. And that's even when just considering the immediate effects and leave the secondary consequences of a dysfunctional work environment aside. Probably, those secondary effects are even more severe involving long-term stress, health issues and motivation.

Despite these findings, companies continue to squeeze employees into crowded office spaces unsuitable for sustained concentration and novel creative work. And programming is definitely in that category. A failure to realize and adapt the environment to fit the demands of the tasks will have significant costs. Given the quarter century since Peopleware presented their findings I wouldn't hold my breath for an immediate change. There is no simple cure to the problem. In the meantime we need strategies to deal with averse working conditions. We need immediate workarounds and remedies. Music and, to an increasing extent, colored noise proves to be one such approach.

Music as a tool

To an individual in a noisy office environment, active earphones with a self-selected choice of audio is often the only practical and legal way to get rid of undesirable background noise. In situations like this music serves by minimizing external and unpredictable disturbances. The purpose of the music is as a tool that allows us to focus on the task at hand.

But music serves in a second role too. It's in this second role that we sometimes come to chose music as a coding soundtrack even under quiet conditions. The idea is that specific music helps us get into a state of flow. Flow is a state where we are completely absorbed in a specific activity. It's a state characterized by full focus and involvement. Flow is a well-researched state that's vital to the creative and complex work of software development.

Using music as a transition into flow is a double-edged sword. Personally, I often find that specific kinds of music serve as a bridge from previous activities, say a meeting or writing an e-mail, to the desired state of flow. Here music allows me to re-focus my attention. However, I do have to chose my music with care. Just as when we use music to shut-out disturbances, making the wrong choice runs the risk of either preventing increased focus or, even worse, pulling me out of a productive flow. To understand the different factors involved, join me on a brief journey through the research on music in the work environment.

The motivational dimensions

Before there were computers, in the era of blue collar workers, Wyatt and Langdon set out to investigate the effects of music on performance among industrial workers [2]. The workers in the Wyatt and Langdon's studies were engaged in simple and repetitive tasks. To them, work wasn't fulfilling nor did they have any expectations of developing their own potential. Work was a way of paying the bills. A necessary evil.

In this context Wyatt and Langdon found that music did improve performance. There were additional benefits too. With music, the involved workers felt that the hours of repetitive work passed faster. Music helped them reduce the boredom of daily routine work.

Software development projects of today are often radically different. Many programmers enjoy their work. Some of us even invest our spare time in writing software, perhaps by contributing to the ever evolving open-source eco-system. To us it's not about passing time. Rather, the task itself fascinates us. We want to deliver our best possible work. Not just because we get paid but rather because programming itself holds value to us.

Ultimately it's about different sources of motivation. The most important and well-known psychological distinction is between *extrinsic* and *intrinsic* motivation. Extrinsic motivation is fueled by either an expected reward or a punishment for the performance or non-performance of a task. Basically it's old school management by carrots and sticks.

Extrinsic motivation was what got the workers in the Wyatt and Langdon study to spend countless hours in the factory in pursuit of a paycheck. The other type of motivation, intrinsic motivation, is related to something we perform out of free will. It's something we do because we find the task itself interesting. To a passionate programmer, writing code for a new, exciting project often falls in this category. Consequently, the role of music in the Wyatt and Langdon studies doesn't apply. What would then the consequences be of adding music to our workplace? Let's find out by looking into some ideas about our consciousness and attention.

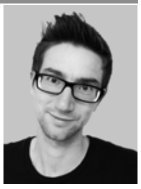
Two-channels, one consciousness

There's a disturbing fact about human consciousness. Despite our frequent attempts in the modern world we just cannot perform two conscious activities in parallel. Human attention doesn't scale; multiple conscious tasks will always compete against each other. And the more conscious involvement in a task, the higher the competition and interference.

If that were bad news regarding our conscious efforts, it may come as a delight that the majority of our cognitive processes are automatic. Automatic processes are unconscious brain activities characterized by their efficiency and effortlessness. Automatic processes are easily

ADAM TORNHILL

With degrees in engineering and psychology, Adam tries to unite these two worlds by making his technical solutions fit the human element. While he gets paid to code in C++, C#, Java and Python, he's more likely to hack Lisp or Erlang in his spare time.



demonstrated using a sophisticated type of test known as two-channel experiments. This type of experiment also illustrates one of the perils with noisy work environments.

In a two-channel experiment, people receive two different streams of information. A typical research procedure is to read two different narratives simultaneously, one to each ear of the participant. Since we can only attend to once conscious stimuli at a time, the two streams just cannot be merged into a single conscious flow. When exposed to these conflicting sources of information, the brain selects one of the streams and masks out the other one. The observable behavior is that information passed to the mentally masked ear isn't encoded into the memory of the participant. As a direct consequence, the participant is unable to recall virtually any of that information. The competing narrative that was attended to is of course remembered well.

The fascinating part starts as we add information of personal meaning to the masked stream. A prime example on information of personal significance is our own name. As soon as that information of personal interest is presented our attention shifts instantly. Our attention now gets re-focused to the previously masked stream of information. The take-away idea is that while we may not be consciously aware of what's happening around us, our brain still scans and interprets the sensations in our environment.

That mechanism had a clear evolutionary advantaged; as our pre-historic relatives were daydreaming on their hunting trips, automatic processes kept them alive by drawing attention to that nasty sabre-toothed tiger that suddenly sneaked up on them. Today the same mechanism keeps its importance for survival (just think about crossing a street while absorbed in deep thoughts). They may also help us get through that cocktail party without missing out on mentions and gossips about our persona. Nonetheless the interruptions triggered by our automatic processes are concentration breakers. They're well capable of destroying our hard-worked focus in front of our favorite editor (Emacs). After all, these processes are about evolutionary survival so who wants to blame them for doing their work? It's not their fault the circumstances have changed drastically.

Today our modern, corporate world presents an environment radically different from the times when our brains evolved. Imagine yourself in an intense programming session in a room shared with friendly, yet chatty co-workers. Those precious automatic processes so vital to our functioning and survival now becomes a disadvantage and source of constant frustration. Similarly, surprising sounds and unexpected stimuli call on our attention. The slam of a coffee mug could well break a deep flow of any knowledge worker. No wonder that we often turn to music as an auditorial shield against the outer world.

The cognitive costs of music

Programming may well be the most cognitively challenging task we can undertake. Not only is the computer ruthless in its execution of our digital creations. We also need to satisfy the needs of a second audience: our fellow team members and maintenance programmers. Since that second group often proves to include our future self, we soon learn about its importance. As programmers, we thus have to switch between different levels of details and constantly balance competing dual goals. We have to constantly learn, re-evaluate and remember large amounts of details and idiosyncrasies about our languages and technical environments.

This chain of cognitively demanding processes is easy to disturb. The resulting consequences may be severe. Think about forgetting an edge case, or perhaps even failing to identify that there is one in the first place. It may boil down to competency issues, but there is often more to it. For experienced programmers ineffective encoding is a potential source of programming errors to come. The typical case is that we fail to remember a vital detail. Not because we somehow lost the memory traces but simply because we never attended to the information in the first place. In fact, failure to remember, the act of forgetting, is often rooted in ineffective encoding.

Conscious encoding is intimately tied to attention. If our attention is directed elsewhere or pending between competing stimulus we fail to commit information to memory. Both noisy environments and self-selected stimulus like listening to music are encoding distractors. Distractors that divide our attention. Without an effective encoding process we never get the information we need to reason about. Our understanding of the problem at hand is incomplete at best.

Now, picture that our encoding was successful. That brings us to our next cognitively sensitive mechanism. Our primary tool, the brain, has limited capacity for the processes programming relies on. One of those core processes involves the conceptual construct called working memory. Working memory is the mental workbench of our mind. It's what we use to hold information in our head and integrate it with existing knowledge. It's a cognitive component vital to high-level reasoning and mental manipulation of information. Our working memory is also a strictly limited resource. Depending on the type of information, our working memory is able to hold a mere 4–7 items simultaneously. And that's under ideal conditions; when listening to music we drain this scarce resource further.

I'd give instrumental music the upper edge since it's associated with other positive side effects

The research on the subject is quite consistent here; when it comes to cognitively challenging tasks, we perform at optimum in quiet conditions. If quiet conditions aren't possible, masking the background noise may increase performance but never to the optimal level reached under quiet conditions (see for example Loewen & Suedfeld [5] or Shih, Huang & Chiang, [6]).

Music or colored noise?

As a way around the cognitive costs of music, some programmers prefer colored noise. Colored noise is computer generated waves based on different mathematical models. The best known of these models is probably white noise, which is a random signal but with equal power in all frequency bands.

There's been a lot of interesting research into the field of colored noise. One frequently quoted study investigated white noise as a tool to increase attention in young students with attention hyperactivity disorder (ADHD). And indeed, white background noise helped the students in the study to reach performance improvements [7]. This sure is promising research. But the main problem when following up on the results is that they don't generalize to the population at large.Worse, the same research team found that white noise significantly hindered the attention of the students who normally paid attention.

Taken to our field of programming, these findings suggest that white noise works well as a noise cancellation technique but, just like music, it cannot compensate for quiet working conditions. Colored noise isn't a panacea either.

The subjective component

Ultimately, the choice between music and colored noise boils down to personal preferences. A recent study found that the influence of background music on concentration had more to do with the listener's fondness for the music than the particular type of music. It's an effect that applies to both ends of the subjective awesomeness spectrum of music preferences: the music that has the most negative impact on your concentration is the one that you either strongly like or dislike [3]. The reasons are probably that the more a certain kind of music affects us emotionally, be it in a positive or negative way, the more of our cognitive resources are allocated to attend to it.

That research points us towards a local optimum. Within your musical preferences there's likely to be music that's more or less suitable to code to. A further guiding principle is found in related neuropsychological

Phenomenal Software Development

Charles Tolman considers the philosophical implications of software development.

Why explore phenomenology?

As we have progressed through the industrial revolution into our current wide ranging use of information technology, there has been a big change in the form of the tools that we use. The massive impact of this transition from external physical tools to internal virtual tools has largely been unconsciously experienced.

Edgar Dykstra back in 1972 was a notable exception when he gave a talk saying:

> Automatic computers have now been with us for a quarter of a century. They have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture, compared with the much more profound influence they will have in their capacity of intellectual challenge without precedent in the cultural history of mankind.

Currently our society is heavily based upon the underlying Cartesian dualistic worldview. Along with this orientation we tend to focus *primarily* on results and though this has been necessary, it has some significant negative consequences. I believe that with the move to virtual tools, the cracks are beginning to show in the Cartesian worldview and its appropriateness for modern times. As computing has progressed along with this has been a questioning of just what it is to be human.

I consider that phenomenology – regardless of whether you can pronounce it or not! – can lead us to a more integrated worldview and I believe the industry needs this more balanced, more human, view if it is to constructively progress.

Overview

I will be starting by providing an overview of my own background. This is important so that you can get a sense of the experiences and thinking that have shaped my conclusions. Only then can you be free to decide what you want to take and what you want to leave.

Then I give some key observations that I've made through my career particularly the one about what I call '*Boundary Crossing*', followed by a short overview of some philosophical ideas. But please note I am not an academic philosopher. Two particular philosophers I highlight are

CHARLES TOLMAN

Charles is a software architect coding mainly in C++. Starting in electronics in the mid 70s, he moved into software after getting an Electronic Engineering degree. He is interested in programmer development as much as in technical competence. Charles blogs at http://charlestolman.com

The Soundtrack to Code (continued)

research. That research suggests that our brain processes lyrical and melodic parts separately. For our conclusions, not only does music draw on our limited cognitive capacity; music with lyrics put an additional load on our resources. The lyrics seem to be a competitor for our attention.

Choosing our soundtrack

These conclusions leave us with a choice between instrumental music or colored noise. Both have the capability to reduce the distractions of random office noise and both imply a slight cognitive performance dip. Which one do we choose when we have to? Again, it comes down to personal preferences. Both alternatives will do. I'd give instrumental music the upper edge since it's associated with other positive side effects. One such effect is the common positive mood induction provided by listening to music. Music may also reduce mental stress and even inspire us on a long-term basis [4]. So, if it's going to affect your performance in a negative way, why not make the best out of it and make the cognitive distractor enjoyable?

Personally I prefer either instrumental music or music of ambient nature where the lyrics are less salient. Music, where the voice serves more like an additional instrument. My preferred choice is atmospheric black metal. It's a kind of music where the main goal is to put the listener in a certain emotional state. It's music of a highly meditative character that I found works well for both transitioning into the zone and maintaining my concentration once there. Similarly I've found that certain electronic music share similar qualities. Your mileage will probably vary.

Summary

All recommendations have to be put in a context. If you're searching for a distraction to get you through a repetitive routine task, music is an excellent choice. It may get you to perform slightly better and make the task more enjoyable in the process. Lyrics or not, choose your favorite acts here.

While music may well boost our mood and performance during routine tasks, it's negative toll on job performance during novel and cognitively demanding tasks is significant. The only thing that's worse with respect to auditory disturbances is a noisy work environment. Until that root problem is addressed, music remains a decent workaround with additional positive benefits on mood and motivation. You personal choice of soundtrack to code to will probably differ from mine. But for an optimal performance I recommend selecting music with similar characteristics. ■

References

[1] DeMarco, T., & Lister, T. (1999). *Peopleware: Productive Projects and Teams (Second Edition)*

[2] Kirkpatrick, F. H. (1943). Music in industry.

[3] Huang, R., & Shih, Y. (2011). *Effects of background music on concentration of workers*.

[4] Jiang, X., & Sengupta, A. K. (2011). *Effect of Music and Induced Mental Load in Word Processing Task* .

[5] Loewen, ,L.J., & Suedfeld, P. (1992). *Cognitive and arousal effects of masking office noise*.

[6] Shih, Y., Huang, R., & Chiang, H. (2009). *Correlation between work concentration level and background music*.

[7] Söderlund, G., Sikström, S., Loftesnes, J. M., & Sonuga-Barke, E. J. (2010). *The effects of background white noise on memory performance in inattentive school children*.

the programmer needs to become more self-aware

Descartes and Goethe as they represent two realms of thought that I consider relevant in their impact on software development. Notable issues here are: Knowledge Generation, Imagination and the Patterns movement.

I then have some conclusions about how we might progress into the future – both with technology development and technology use.

A programmer's background : Novice – the early years

I started out being interested in electronics at 17 back in 1974. Originally I was a shy young adolescent nerd who found comfort in the inner world of thought. Also I was not good at dealing with members of the opposite sex, which I believe could be quite a common phenomenon among younger software developers.

Thereafter I gained entry to Southampton University in order to study electronic engineering gaining my degree in 1979. Even at this stage I realized that I wanted to move from hardware development to software development, although I only had an unconscious sense of this physical to virtual transition.

Early programming tasks were a hobby at the time and were based on programming games in BASIC on computers I had built from a kit. There was an initial foray into trying to do an IT records management application which I messed up completely.

Then came the job in the field of media TV and film editing systems where I was definitely feeling that I was working with 'cool' tech. Definitely a time of being enticed by the faery glamour of the technological toys.

A programmer's background : Journeyman – the dangerous years

It was the next phase of the career that I call the dangerous time. A time characterized by the following traits:

- Wanting to play with more complex and generic structures. (Many of which did not actually get used!)
- A focus on the tools rather than the problem.
- The creation of unnecessarily complex systems, letting the internal idea overshadow the external problem context.
- An arrogance about what could be achieved – soon followed by absolute sheer panic as the system got away from me.
- No realization that the complexity of thought required to debug a system is higher than that required to originally design and code the system.

This phase of a career can last for a long time and highlights the fact that the programmer needs to become more self-aware in order to progress from this stage. In fact some people never do.

This can be a real problem when recruiting experienced programmers. When interviewing I separate the time into two sections. Initially I ensure that the interviewee has the required level of technical competence, and once I feel they are more settled I move on to see just how self-aware they are.

One question I use here is "So tell me about some mistakes?" There are two primary indicators that I am looking for in any response. The first one is the pained facial expression as they recall some past mistakes that they have made in their career and how they have improved in the light of those experiences. The second is the use of the word 'I'.

'I' is an important word for me to hear as it indicates an ownership and awareness of the fact that they make mistakes without externalising or projecting it onto other people or the company. This is important because it will show the degree of openness that the interviewee has to seeing their own mistakes, learning from them, and taking feedback. A programmer who cannot take feedback is not someone I would recruit.

A programmer's background : Grumpy old programmer

This 'Old Grump' phase is possibly a new one that developers go through before reaching Master level. I hesitate to describe myself as Master but am currently definitely at the Old Grump stage! Traits here I have experienced are:

- Awareness of the limitations of one's own thinking, after realizing again and again just how many times one has been wrong in the past. Particularly easy to notice when debugging.
- Realization that maintenance is a priority, leading to a drive to make any solutions as simple and clear and minimalist as possible. Naturally the complexity of the solution will need to match if not exceed the complexity of the problem. Once one has experienced the ease with which it is possible to make mistakes it is always worth spending more time making solutions that are as simple as possible, yet do the job. An Appropriate Minimalism.
- Code ends up looking like novice code, using complexity and 'big guns' when required.
- A wish to find the true essence of a problem, but when implementing using balanced judgement to choose between perfection and pragmatism.
- Most people think that because you are more experienced you are able to do more complex work. The paradox is that the reason you do better is that you drop back to a much more simple way of seeing the problem without layering complexity upon complexity. (This strongly correlates with the phenomenological approach.)

Philosophical considerations

So I hope that gives you some idea of the changes in thinking that can occur throughout a career in programming. In subsequent articles I want to draw out some key observations and link them to philosophical developments that have taken place over the last 300 years or so.

It is not widely recognized that there has been a sea-change in philosophical thought during the 1900s and these major changes are starting to make themselves felt with a vengeance in our society. I intend to show that the practice of software development can provide significant insights into how to deal with these changes and I hope to lead you to a wider vision of what we are doing.

It is not just class diagrams, languages, stand-ups and typing! ■

An Unexpected Journey

Jez Higgins finds treasure in the new Java.

Steve, who you may have noticed at the front of this very magazine, recently published narl, a C++ range library, on GitHub [1]. I read through the code but only caught up with the accompanying *Overload* article [2] a few weeks later. I liked the code, but I really liked the article. I liked it so much, I read it twice. And then I wrote some code of my own.

A few years ago I gave a couple of talks on doing useful things with iterators, specifically in Java, looking at using them for filtering, transformation, and so on. The main idea I was trying to get across was that while iterators, particularly in Java, are mainly used to loop from one end of a collection to another, iterators are a useful thing in their own right. They could iterate over anything, even things that didn't exist. This is a more familiar idea in C++, where stream extractors and insertors exist as part of the standard library, but still not as widely applied as it might be. This is, I'd suggest, primarily because of the pointer-like nature of C++ iterators, which means you have to write a lot of things twice. As a case in point, Listing 1 is a snatch of sample code featuring a C++ filtering iterator [1].

The **filter_iterator** does what its name suggests and its use is straightforward, but my those declarations are clunky. Think about trying to combine it with an another wrapper iterator, perhaps a transformer, and you can see it would all get very messy very quickly.

Coming back to Java, I think what I was saying was good stuff and fundamentally sound (I got quoted, almost verbatim but unconsciously I'm sure, by Kevlin in one of his Conference talks a couple of years back so it must have been ok), but the code was just as unwieldy to work with:

```
Iterator<U> b = new Transformer<T, U>(
  new Filter<T>(list, new Predicate<T> { ... }),
  new Function<T,U> { ... });
  ...
```

There is SO MUCH NOISE – those pointy brackets, the anonymous classes, new this, new that, new the other! Plus it's all in the wrong order. Actually it's worse than the wrong order – it's just in a crazy order. It really isn't obvious what's going on. I did build some quite handy stuff with it, and other people did too, but I was never really happy with it.

Like Steve, I was also in Andrei's ranges talk [5]. The ideas he described and the code he presented were terrific and I got quite excited. However, I wasn't working in C++ at the time and I guess I was waiting for Andrei's range library to appear. Of course, he headed off and wrote it in D. More importantly even though what he was describing as a range was, conceptually, very close to a Java iterator, and it was only a few months since I'd given my own presentation I just didn't make the connection. Basically, I am an idiot.

I get it now though, because Steve pulled me by the nose to it through his article. There's an elegance and clarity to Steve's narl stuff. You read the line and you know what it does (Listing 2).

Listing 1

```
filter_iterator<int> fb(vec.begin(), vec.end(),
   Even());
filter_iterator<int> fe(vec.end(), vec.end(),
   Even());
for( ; fb != fe; ++fb)
{
  ... do something with *fb ...
} // for ...
```

Listing 2

```
auto r = from(src)
  | where([](const item& i) {
    return i.size() > 0; })
  | select([](const item &i) {
    return i.name(); });
```

It's lovely. It's why C++ is so cool sometimes – no base classes, no runtime overheads, clear syntax, extensible, etc, etc. (Steve's written a one-article argument for operator overloading and it's incidental to his main topic.) Reading that code, I had an epiphany.

So I went back to the Java world I mainly live in now, put back in the base classes and the runtime overhead, sacrificed the extensibility, but hopefully managed to retain some of the clarity. Now I'm writing code like

```
Iterator<U> = from(list).
               where(new Predicate<T> {...}).
               select(new Function<T, U> {...});
```

This is does exactly the same thing as the noisy code snippet above except that what it does is much clearer. There're still the **new**s and more pointy brackets than you'd like, but it's much, much better. That's quite a good result.

One of the reasons Steve's code looks so pleasing is we now live now in a previously unimaginable crazy futureworld where C++ got lambdas before Java. Some time next year (and, of course, subject to our corporate overlords and their various contractual relationships) those of us nurdling around in Javaland will be able to replace those clunky anonymous classes with sleek new Java lambdas.

We'll go from this

```
List<U> r = from(list).
   where(new Predicate<T>
   { public boolean test(T t) { ... } ).
     select(new Function<T, U>
     { public U apply(T t) { ... }).
       toList();
```

to this

```
List<U> r = from(list).
            where(t -> ...).
            select(u -> ...).
            toList();
```

Even nicer!

I confess to not having paid a huge amount of attention to Java 8 and its feature set. It's been so long coming, it was starting to feel a little like Perl 6, always just over the horizon. It turns out if we dig a little deeper in Java 8, we can write this

```
List<U> r = list.stream().
            filter(t -> ...).
            map(u -> ...).
            collect(Collectors.toList());
```

and I can chuck out my code completely. Remarkably this stuff, called Streams, is baked into the standard java.util package.

JEZ HIGGINS

Programming for over 30 years, using 20 or so different languages, in a dozen different fields, Jez Higgins is still trying to work out how to do it properly. If you can help, he can be contacted at jez@jezuk.co.uk

I say remarkably because the java.util package, while undeniably Java, hasn't laboured very much on the util. The `Queue` and `Deque` interfaces and their implementations, for instance, didn't arrive until Java 6. The `Objects` utility methods didn't arrive until Java 7. Many of the iterfaces and classes in util try to give the impression of being useful through maximalism – they have lots and lots of methods. Lots of methods must mean more utility, right? Right? Well, no.

Consider the `List` iterface, which has 23 methods. It includes a full set of methods to get, set, insert, and remove items from the list at a specific index, regardless of the fact that you almost certainly only want to perform those operations on a `List` that provides O(1) random access. You almost certainly don't want to be doing on a linked list for instance. But not only will Java let you, its library interfaces mandate that a list, any list, all `List`s, **must** support it.

There's the rather curious `retainAll(Collection<T> coll)` method, which removes everything from the `List` which isn't in the provided collection. I accept there are times when you want to this, but it's not so common you need to provide it as a method on all `List` implementations. Surely the more common case, given that you've gathered up the things you need into a second collection already is simply to discard the first collection and keep the second?

A rather striking omission from the `List` interface is `sort`. That's a really, really common operation and one that could reasonably be provided as a member method. `ArrayList.sort`, for instance, could index directly into its underlying array, while `LinkedList.sort` would have to go rather more round the houses. Instead we have the `Collections.sort` static method. It's 'generic' in the sense that it works with all `List` implementations and treats them all in the same way. And that way is to copy the list's contents into an array, sort the array, and then write the array's contents back into the list. It's about as horrible a way to sort a list as you can think of. (`Collections.sort` is called sixteen times in the Java 7 source while `List.retainAll` is not, as far as I can tell, used in the Java 7 source at all.)

As an additional oddity, since Java 1.4 java.util has provided the `RandomAccess` marker interface to indicate that `List`s support fast random access. Presumably it was dropped in because somebody somewhere reviewed `List` and clapped a hand to their forehead while yelling a loud 'DOH'. Backward compatibility ruled out changing `List` too radically and so `RandomAccess` was introduced 'to allow generic algorithms to alter their behaviour to provide good performance when applied to either random or sequential access lists'. If you poke around in the handful of generic algorithms provided in `Collections`, `shuffle` for instance, `rotate`, or `swap`, they do all check for and switch on `RandomAccess` and act accordingly. But not `sort`. Even in Java 8, the single most common `List` operation copies the list into an array, sorts the copy, then does a linear traverse to overwrite the original.

By constrast, the new Streams stuff looks terrific (and I'm not just saying that because it looks virtually identical to the code I wrote) and I hope it introduces a step-change in how we use Java collections.

Another confession – I didn't actually set out to write an article about iterators and ranges. I wrote a number of shorter pieces on my blog which just seemed to fit together. One of those pieces was, I later discovered, was picked up on reddit [7] where quite a healthy discussion ensued. A mildly sarcastic commenter there suggested that this fancy Java 8 Streams business wasn't anything new, because it's existed in Groovy for years. Well, yes and also no.

Groovy (once the shiny new future of development in Java and now possibly suffering a mild existential crisis) has a cunning (and/or dangerous, depending on your outlook) extension (and/or hack, depending on your outlook) mechanism for grafting new methods into existing classes. The prime use for these extensions is the Groovy JDK, which adds any number of methods into the JDK classes to make them, well, make them more Groovy.

Among the methods added to `Iterator` are various `sort`, `take`s, and `sum`s, but nothing directly equivalent to `filter`/`map` (aka `where`/`select`). There is a `grep` method added to `Collection`, but that returns another `Collection`. However, this isn't really what sets Java 8 apart from Groovy – after all you could always write your own extensions to add those methods in yourself. The thing that sets Java 8 Streams apart is the drop dead simple support for parallelism.

Listing 3

```
public class Serial {
  private static void pi() {
    final long startTimeNanos =
       System.nanoTime();
    final int n = 1000000000;
    final double delta = 1.0 / n;
    final double pi =
      4.0 * delta * multiplier(n, delta);
    final double elapseTime =
      (System.nanoTime() - startTimeNanos) / 1e9;
    System.out.println(String.format("%f %f", pi,
      elapseTime));
  }
  static private double multiplier(int count,
     double delta) {
    final int start = 1;
    final int end = count;
    double sum = 0.0;
    for (int i = start; i <= end; ++i) {
      final double x = (i - 0.5) * delta;
      sum += 1.0 / (1.0 + x * x);
    }
    return sum;
  }
  public static void main(final String[] args) {
    pi();
  }
}
```

Java has had support for parallelism since it was launched, and has gradually got to the point where it's actually usable. Streams, to my untutored look really quite remarkably easy. ACCU stalwart Russel Winder, who knows far more about high performance numerical computing than I ever will, often uses quadrature calculation of pi as an example. Listing 3 is a simple serial version, which on my machine outputs 3.141593 12.560689.

Quadrature calculation of pi is a problem Russel describes as 'embarrassingly parallel'. What he means by that is that you can divide the calculation into as many chunks as you like, evaluate them in any order, combine them back together, and the result is the same. Let's chunk it up (Listing 4).

My machine gives

```
3.141593 12.379540 1
3.141593 11.487891 2
3.141593 9.969373 8
3.141593 10.028523 16
```

As you'd expect the same value of Pi, with more or less the same run time. That `for` loop is a bit last century, given my current obsession with ranges and streams. Let's rewrite the `for` loop with an integer range and head into the future with a nice map and sum. (See Listing 5.) Again, as we would expect, the results are as before.

```
3.141593 11.982472 1
3.141593 11.369035 2
3.141593 10.302495 8
3.141593 10.302260 16
```

Now let's go parallel. Look carefully or you'll miss it (Listing 6). As ever, the value of Pi is unchanged:

```
3.141593 12.535362 1
3.141593 6.488914 2
3.141593 4.017939 8
3.141593 1.519504 16
3.141593 1.522862 32
```

Listing 4

```
public class SerialChunk {
  private static void pi(int chunks) {
    final long startTimeNanos =
System.nanoTime();
    final int n = 1000000000;
    final int chunkSize = n / chunks;
    final double delta = 1.0 / n;
    double multiplier = 0.0;
    for (int i = 0; i < chunks; ++i)
      multiplier += multiplierChunk(i,
         chunkSize, delta);
    final double pi = 4.0 * delta * multiplier;
    final double elapseTime =
       (System.nanoTime() -
          startTimeNanos) / 1e9;
    System.out.println(String.format("%f %f %d",
       pi, elapseTime, chunks));
  }
  static private double multiplierChunk
     (int chunk, int chunkSize, double delta) {
    final int start = 1 + (chunk * chunkSize);
    final int end = (chunk + 1) * chunkSize;
    double sum = 0.0;
    for (int i = start; i <= end; ++i) {
      final double x = (i - 0.5) * delta;
      sum += 1.0 / (1.0 + x * x);
    }
    return sum;
  }
  public static void main(final String[] args) {
    pi(1);
    pi(2);
    pi(8);
    pi(16);
  }
}
```

but look at the run time fall! My machine has an i7 with 8 cores and hyperthreading is on, so 16 threads of execution saturates it and you'd expect to see a small degradation in speed as you push the number of chunks beyond that.

To move the calculation from a serial mode to a parallel mode took one line of code. That's pretty lovely.

Russel has a vast number of examples in any number of languages in his Github repository, from where I adapted the code above. I can't claim to have read them all but the Java 8 version is one of the easiest to read and comprehend, ahead even of the Clojure, D, and Go examples. Parallel processing in readable Java – that really is remarkable!

Listing 5

```
import java.util.stream.IntStream;
public class SerialStream {
  private static void pi(int chunks) {
    ...
    double multiplier =
      IntStream.range(0, chunks)
      .mapToDouble(chunk ->
      multiplierChunk(chunk,
      chunkSize, delta)).sum();
    ...
  }
  static private double
     multiplierChunk(int chunk, int chunkSize,
     double delta) { ... }
  public static void main(final String[] args) {
    pi(1);
    pi(2);
    pi(8);
    pi(16);
  }
}
```

Listing 6

```
import java.util.stream.IntStream;
public class ParallelStream {
  private static void pi(int chunks) {
    ...
    double multiplier =
           IntStream.range(0, chunks).
           parallel().
           mapToDouble(chunk -> multiplierChunk
           (chunk, chunkSize, delta)).sum();
    ...
  }
  static private
     double multiplierChunk(int chunk,
     int chunkSize, double delta) { ...  }
  public static void main(final String[] args) {
    pi(1);
    pi(2);
    pi(8);
    pi(16);
    pi(32);
  }
}
```

It's a funny old business software development. Despite my excitement, there's no new computer science anywhere in this article. We can probably trace the core of ranges and the declaratively functional programming style they allow to the discussion over coffee that ensued after McCarthy first delivered his paper on Lisp. There's a more personal narrative in here – and again this is not a new conclusion – it's that software development and getting better as a programmer is a long game. Everything's relevant, but you can never know when until the stars align and the pieces come together. A lecture from Andrei Alexandrescu, an encounter with C#, and the arrival of C++ lambdas – over the course of several years – combined set up Steve to write his library. That proved to be the key that crystallised out ideas I'd been kicking around for at least as long, and my understanding of containers and how to manipulate them and their contents has moved into some kind of higher realm. Trying to put that understanding into code lead me to dig into Java's past and then become quite excited about its future, while learning something new about parallel computing. Achievement unlocked! Level up! So thanks, Steve. Thanks Andrei. Thanks everyone.

So, where next? ■

References and more information

[1] narl - Not Another Range Library, http://github.com/essennell/narl

[2] 'Range and Elevation', *Overload* 117

[3] 'Finding the Utility in a java.util.Iterator', http://www.jezuk.co.uk/accu2007/iterator/

[4] A filtering iterator in C++, http://www.jezuk.co.uk/cgi-bin/view/jez/2008February#3612

[5] Iterators Must Go!, http://accu.org/content/conf2009/AndreiAlexandrescu_iterators-must-go.pdf

[6] java.util.List, http://docs.oracle.com/javase/7/docs/api/java/util/List.html

[7] Nice post on what lambdas will look like in Java 8, http://www.reddit.com/r/programming/comments/1rumev/nice_post_on_what_lambdas_will_look_like_in_java_8/

[8] Creating an extension module, http://docs.codehaus.org/display/GROOVY/Creating+an+extension+module

[9] Pi Quadrature, https://github.com/russel/Pi_Quadrature/

Generating Code from a Unit Test (Part 3)

Richard Polton generates the code to pass his unit tests.

Last time we were talking we had managed to construct some program code which generated a simple interface definition given some unit tests as input and we finished with a few suggestions for future areas of development. Today we are going to create an implementing class from the interface already constructed. Given the interface declaration syntax **iface**, the following code snippet produces a public class, uninspiringly called **className**, which implements the interface. Currently the method bodies do no more than return a default-valued object of the appropriate type, but first steps .... So, given the interface declaration, for each of the function members contained therein construct an implementing function. Then create a class declaration which incorporates these implementing functions (Listing 1 and Listing 2), which results in Listing 3.

Very interesting, I hear you say, but how do we generate a meaningful function body? Clearly, for boolean functions, we can automatically generate the return statement and supply the correct value roughly half of the time. However, in order to supply the correct return statement all of the time we need more information when generating the function itself. Specifically, we need the assertion method that was used to generate the function signature when building the interface. This complicates the data flow slightly as we will need to pass additional information with the syntax tree we have generated.

Having established that we need to pass the assertion information with the function signature, how should we go about doing this? We could create a second parallel data structure which is keyed into the syntax tree that we generated in some manner. For the purposes of this article, though, let us modify the manner in which we create the syntax tree that we generate to describe the interface so that we can store this additional information in the syntax tree itself, as comments or attributes or similar.

If we choose to represent the additional assertion information in comments then we can simply render the text of the original code as the comment. Presumably we could parse the comment into a tree or similar should we choose to do so. Attributes support a tree structure although not necessarily the same tree structure as the original node. The downside with the use of attributes is that we would want the final code to be compilable and therefore the attribute class used must necessarily exist.

So let us create an **AssertionOriginAttribute** that we can utilise in the code generation. If we construct this attribute with the assertion expression which led to the generation of the function under investigation then we should be able to recover the information when we try to generate a compliant function body. This should work particularly well as we already know how to parse the expression having done so in a previous article [1].

When building the **Syntax.MethodDeclaration** sequence in **InterfaceNameAndMethodDeclarations.MethodDecls** we attach an attribute list to each of the interface functions using Listing 4. This change leads to an interface like Listing 5, which gives us the information that we need at the same time as being almost valid C#. I suppose if we made the assertion expression a lambda function then it would be legal C# but let's not worry about that now.

Listing 1

```
private static ClassDeclarationSyntax
   CreateImplementingClass
    (InterfaceDeclarationSyntax iface)
{
  var memberDecls =

iface.Members.OfType<MethodDeclarationSyntax>()
     .Select(ConstructImplementingFunction);
  return Syntax.ClassDeclaration(
    attributeLists:
      new SyntaxList<AttributeListSyntax>{},
    modifiers: Syntax.TokenList(
      Syntax.Token(SyntaxKind.PublicKeyword)),
    identifier: Syntax.Identifier("className"),
    typeParameterList: null,
    baseList: Syntax.BaseList(
      new SeparatedSyntaxList<TypeSyntax>().
      Add(new [] {Syntax.IdentifierName
        (iface.Identifier)})),
    constraintClauses: null,
    members: Syntax.List<MemberDeclarationSyntax>
      (memberDecls));
}
private static MemberDeclarationSyntax
  ConstructImplementingFunction
  (MethodDeclarationSyntax method)
{
  var returnStmt =  Syntax.ReturnStatement(
    Syntax.Token(SyntaxKind.ReturnKeyword),
      Syntax.ParseExpression("new " +
      method.ReturnType.ToString() + "()"),
      Syntax.Token(SyntaxKind.SemicolonToken));
  return Syntax.MethodDeclaration(attributeLists:
    new SyntaxList<AttributeListSyntax> { },
      modifiers: Syntax.TokenList(
        Syntax.Token(SyntaxKind.PublicKeyword)),
      returnType: method.ReturnType,
      explicitInterfaceSpecifier: null,
      identifier: method.Identifier,
      typeParameterList: null,
      parameterList: method.ParameterList,
      constraintClauses: null,
      body: Syntax.Block(statements: new[]
        { returnStmt }));
}
```

Now, when generating the return statement in the class method/function, we can parse the **InvocationExpressionSyntax** object contained within the attribute and determine what the return value is expected to be. Thus, let us change **ConstructImplementingFunction** so that, instead of returning a default **'new T()'**, we return the expected value.

And so now we have Listing 7.

Listing 2

```
var classDeclaration = interfaceProgramCode.
  Select(CreateImplementingClass);
var classDecl_asString = classDeclaration.
  Select(pc =>
    pc.NormalizeWhitespace().ToFullString());
```

RICHARD POLTON

Richard has enjoyed functional programming ever since discovering SICP and feels heartened that programming languages are evolving back to LISP. He likes 'making it better' and enjoys riding his bike when he can't. He can be contacted at richard.polton@shaftesbury.me

Listing 3

```
classDecl_asString.ToList()
Count = 1
  [0]:
    public class className : AnInterface
    {
      public bool DoIt()
      {
        return new bool ();
      }
      public bool Verify(Int32 a)
      {
        return new bool ();
      }
    }
```

Listing 4

```
WithAttributeLists(
  Syntax.List<AttributeListSyntax>(
    Syntax.AttributeList().WithAttributes(
      Syntax.SeparatedList<AttributeSyntax>(
        Syntax.Attribute(
          Syntax.ParseName
            ("AssertionOriginAttribute"),
          Syntax.AttributeArgumentList(
            callingAssertionExpr.
             ToAttributeArgumentSyntaxList() )
          ) ) ) ) )
```

Listing 5

```
interfaceProgramCode_asString.ToList()
Count = 1
  [0]:
    public interface AnInterface
    {
      [AssertionOriginAttribute
        (Assert.IsTrue(a.DoIt()))]
      public bool DoIt();
      [AssertionOriginAttribute
        (Assert.IsTrue(a.Verify(1)))]
      public bool Verify(Int32 a);
    }
```

Listing 6

```
var expectedReturnValue =
    DeriveExpectedReturnValue(
  (InvocationExpressionSyntax)
    assertionAttrs.Attributes
  .First().ArgumentList.Arguments.
   First().Expression));
// DeriveExpectedReturnValue from the method
// attribute list
var returnStmt = Syntax.ReturnStatement(
  Syntax.Token(SyntaxKind.ReturnKeyword),
     expectedReturnValue,
  Syntax.Token(SyntaxKind.SemicolonToken));
```

Listing 7

```
classDecl_asString.ToList()
Count = 1
  [0]:
    public class className : AnInterface
    {
      public bool DoIt()
      {
        return true;
      }
      public bool Verify(Int32 a)
      {
        return true;
      }
    }
```

Listing 8

```
// a.DoIt() or a.Verify(1)
var assertionArgList = expr.DescendantNodes().
  OfType<ArgumentListSyntax>();
// () or (1) in our example
var functionUnderTestArgList =
  assertionArgList.SelectMany(args =>
    args.DescendantNodes().
    OfType<ArgumentListSyntax>());
if (functionUnderTestArgList.
    First().Arguments.Any())
{
  return Syntax.BinaryExpression(
    SyntaxKind.EqualsExpression,
    Syntax.IdentifierName("a"),
    functionUnderTestArgList.First().
      Arguments.First().Expression);
}
```

As we said above, little steps. At this point we can say that we have automatically generated a class that correctly implements the expected interface and satisfies the supplied unit tests. Unfortunately the unit tests were of low quality. Is there anything else we can infer from the tests as they are presented? Notice that **`Verify`** is called with a literal '1'. It seems reasonable to assume that the test was expected to succeed because the parameter was '1' (otherwise what would be the purpose of the parameter?) so let us parse the attribute and extract the function parameter(s). A reasonable first attempt at a conformant function body would be something like

```
{
  return parameter == 1;
}
```

In order to achieve this we need to modify our **`DeriveExpectedReturnValue`** function, let's rename it to **`DeriveExpectedReturnExpression`**, so that it parses the argument list at the call site and extracts the parameter passed. So let us change the function to check for any parameters and, if they exist, parse them. For example, given the **`InvocationExpressionSyntax`** that describes the assertion statement, we can build an expression to be returned using the code in Listing 8.

If the predicate evaluates to false then return the literal expression as before. Now we have Listing 9.

Okay, all well and good. How about we add another test function in which we call **`Verify`** and assert that its return value is false? So, let us add the below to our unit test suite.

```
[Test]
public void ThisIsATest3()
{
  var a = Mock<AnInterface>();
  Assert.IsFalse(a.Verify(2));
}
```

Listing 9

```
classDecl_asString.ToList()
Count = 1
  [0]:
    public class className : AnInterface
    {
      public bool DoIt()
      {
        return true;
      }
      public bool Verify(Int32 a)
      {
        return a == 1;
      }
    }
```

Listing 10

```
classDecl_asString.ToList()
Count = 1
  [0]:
    public class className : AnInterface
    {
      public bool DoIt()
      {
        return true;
      }
      public bool Verify(Int32 a)
      {
        return a == 1;
      }
      public void Verify(Int32 a)
      {
        return a == 2;
      }
    }
```

Given the current implementation of **DeriveExpectedReturnExpression** this should 'Just Work'™. Let's try...(Listing 10).

Ah! Of course! Now we have multiple tests for the same function thus making our code generator a little more complex. The problem we have at the moment is that we create a new assertion attribute are created for each **Assertion** statement. However, with the addition of **ThisIsATest3**, we now have two **Assertion** statements for the single **Verify** function. Looking back at the interface generation code we can see that **Verify** is now twice-declared there too, cf. (See Listing 11.)

So the problem is at least two-fold. The multiple assertions need to be folded into the single interface function declaration and also into a single function body in the implementing class. We need to cast our minds back to the definition of **ConstructMethodDeclarations** because, right now, this is returning one interface having three functions instead of two. We need something more sophisticated than a **GroupBy** expression to build the declarations. Cast your mind back a while. The **ConstructMethodDeclarations** function loops through the assertion statements and, essentially, returns a collection of method declarations, one for each assertion statement. We need to change this so that the assertions are grouped in some manner so that we can construct the **AssertionOriginAttribute** list and populate it with multiple assertions.

It seems, therefore, that we need to loop through the assertions first building a container of interfaces and methods and then loop through the resulting container grouping the methods ..... Phew, okay back to the drawing board.

Having recognised that there may be multiple assertions for any given interface method, we change the function that constructs the interface method declarations so that **interfaceProgramCode_asString** contains (see Listing 12).

Listing 11

```
interfaceProgramCode_asString.ToList()
Count = 1
  [0]:
    public interface AnInterface
    {
      [AssertionOriginAttribute
         (Assert.IsTrue(a.DoIt()))]
      public bool DoIt();
      [AssertionOriginAttribute
         (Assert.IsTrue(a.Verify(1)))]
      public bool Verify(Int32 a);
      [AssertionOriginAttribute
         (Assert.IsFalse(a.Verify(2)))]
      public void Verify(Int32 a);
    }
```

Listing 12

```
public interface AnInterface
{
  [AssertionOriginAttribute
     (Assert.IsTrue(a.DoIt()))]
  public bool DoIt();
  [AssertionOriginAttribute
     (Assert.IsTrue(a.Verify(1))),
   AssertionOriginAttribute(
     Assert.IsFalse(a.Verify(2)))]
  public bool Verify(Int32 a);
}
```

The revised **ConstructMethodDeclarations** function is now as in Listing 13.

Listing 13

```
public static Ienumerable<IGrouping<
  string, InterfaceNameAndMethodDeclarations>>
ConstructMethodDeclarations(SemanticModel model,
  IEnumerable<MethodDeclarationSyntax>
    methodsHavingTestAttributes)
{
  var assertedInterfaces =
    new List<AssertedInterface>();
  foreach (var testMethodDeclaration in
     methodsHavingTestAttributes)
  {
    // find the mocks
    var mocks =
      testMethodDeclaration.Body.Statements.
      SelectMany(stmt => stmt.DescendantNodes().
        OfType<GenericNameSyntax>()).
        Where(nm => nm.Identifier.ValueText ==
          "Mock");
    Func<GenericNameSyntax,IEnumerable<string>>
     mockedInterfaceTypes =
      mock => mock.TypeArgumentList.Arguments.
      OfType<IdentifierNameSyntax>().
      Select(ident => ident.Identifier.ValueText);
      // "AnInterface"

    // Find the variable declarations => get the
    // names. Recurse up through the tree from the
    // mocks to find the variable declarations.
    var varDecls = mocks.Select(mock =>
      new { Mock = mock, Decl = mock.Ancestors().
        OfType<VariableDeclarationSyntax>().
           First() });
    var varIdents = varDecls.Select(decl =>
      new { Mock = decl.Mock, Identifier =
         decl.Decl.DescendantNodes().
        OfType<VariableDeclaratorSyntax>().
           First().Identifier });
    var varNames = varIdents.Select
      (decl=>decl.Identifier.ValueText).ToList();

    // find the asserts
    var asserts =
       testMethodDeclaration.Body.Statements.
      SelectMany
         (stmt => stmt.DescendantNodesAndSelf().
        OfType<ExpressionStatementSyntax>()).
      Where(expr => expr.DescendantNodes().
        OfType<IdentifierNameSyntax>().
        Any(node => node.Identifier.ValueText ==
             "Assert"));

    // find the mocks' accesses -> fn names and
    // parameters, return values. We don't care
    // about functions which are not 'assert'ed
```

Listing 13 (Cont'd)

```
  // so search down from 'asserts' instead of
  // 'testMethodDeclaration.Body.Statements'
  var assertedMocks = asserts.Where
     (stmt => stmt.DescendantNodes().
    OfType<IdentifierNameSyntax>().
      Select(id => id.Identifier.ValueText).
      Intersect(varNames).Any());
      // Assert.IsTrue(a.DoIt());
  Func<ExpressionStatementSyntax,
    IEnumerable<InvocationExpressionSyntax>>
       extractInvocations =
    expr => expr.DescendantNodes().
      OfType<ArgumentListSyntax>().
      SelectMany(node =>
         node.DescendantNodes().
        OfType<InvocationExpressionSyntax>());
  var joinedMocksAndAssertedMocks =
    varIdents.Join(assertedMocks,
    ident => ident.Identifier.ValueText,
    assertedMock =>
      extractInvocations(assertedMock).First().
      DescendantNodes().
         OfType<IdentifierNameSyntax>().
      First().Identifier.ValueText,
    (ident, assertedMock) =>
       new AssertedInterface
    {
      Interface =
         mockedInterfaceTypes
         (ident.Mock).First(),
      AssertedMock = assertedMock,
      Invocation =
         extractInvocations(assertedMock)
        .First()
    });
  assertedInterfaces.AddRange
     (joinedMocksAndAssertedMocks);
}
// perform some sort of grouping - we need a
// single instance of each function but there
// could be multiple assertions for each
// function. For each assertedInterface, group
// all the assertions for each function
// together
var groupedInterfaces =
   assertedInterfaces.GroupBy
   (aif => aif.Interface);
var methodDecls = new
   List<InterfaceNameAndMethodDeclarations>();
foreach (var groupedInterface in
   groupedInterfaces)
{
  var groupByFn =  groupedInterface.GroupBy(
    grp=>grp.Invocation.DescendantNodes().
    OfType<IdentifierNameSyntax>().Skip(1).
      First().ToString());
  foreach (var gf in groupByFn)
  {
    var methodCalls = gf.ToList();
    var firstInvocation = methodCalls[0];
    var returnType = ExtractAssertionReturnType
       (firstInvocation.AssertedMock);
    // Could call
    // model.GetTypeInfo(invocationExpr) but
    // none of the names involved are defined
    // so the function returns ErrorType, ie
    // unknown
    var invocationArgList =
       firstInvocation.Invocation.ArgumentList
       as ArgumentListSyntax;
```

```
      var invocationArgs =
         invocationArgList.Arguments;
      var functionParameterTypes =
         invocationArgs.Select(argSyntax =>
      {
        var argType = model.
          GetTypeInfo(argSyntax.Expression).Type;
        return Syntax.Parameter
          (Syntax.Identifier(@"a")).
          WithType(Syntax.ParseTypeName
            (argType.Name));
      });
      var fpt = functionParameterTypes.ToList();

      var method =
        new InterfaceNameAndMethodDeclarations
      {
        Name = groupedInterface.Key,
        MethodDecls = new[]{
          Syntax.MethodDeclaration(returnType,
          gf.Key).
          WithModifiers(Syntax.TokenList(
            Syntax.Token
            (SyntaxKind.PublicKeyword))).
          WithTfmIfTrue(
            t => t.WithParameterList
               (Syntax.ParameterList(
              Syntax.SeparatedList
                 <ParameterSyntax>(fpt,
                 Enumerable.Repeat(
                Syntax.Token
                   (SyntaxKind.CommaToken),
                   fpt.Count - 1)))),
            () => functionParameterTypes.Any()).
          WithSemicolonToken(Syntax.Token
             (SyntaxKind.SemicolonToken)).
          WithAttributeLists(
            Syntax.List<AttributeListSyntax>(
            Syntax.AttributeList()
               .WithAttributes(
              methodCalls.Select(invocation=>
                Syntax.Attribute(
                  Syntax.ParseName
                    ("AssertionOriginAttribute"),
                  Syntax.AttributeArgumentList(
                    invocation.AssertedMock.
                  ToAttributeArgumentSyntaxList()
                )))).
            ToSeparatedSyntaxList())))
        }
      };
      methodDecls.Add(method);
    }
  }
return methodDecls.GroupBy
   (methodDecl => methodDecl.Name);
}
```

Listing 13 (Cont'd)

However, we still need to modify the class generation code because it does not currently recognise multiple assertions for a single function. With a little tweak to **`ConstructImplementingFunction`** to incorporate the separate equalities into a single logical expression, we have **`classDecl_asString`** (see Listing 14, overleaf).

And that, good reader, is where I will leave it for this instalment. ■

Reference

[1] *CVu* 25.4 and 25.5 'Generating code from a unit test'parts 1 and 2.

Architectureless Software Design

Vsevolod Vlaskine explores some ideas of simplicity in software design.

Complexity of system architecture

When one designs software, what comes out of it? An architecture, is not it? Or is it? The word 'architecture' applied to software has so many distorting connotations that I am trying to consciously expel it from my vocabulary.

The architectural approach assumes a vision of the system where its components connect, interact, build on top of each other, to form the whole edifice functioning as one organism. Such a vision maybe important in a project, but only as, well, a vision rather than the master blueprint from which the system eventually materializes.

Once we admit that we are building 'a software system' the next inevitable design step would be to split it into parts, establish their relationships as subordination, aggregation, connections, interfaces, data flows, etc. Then we further analyse the parts in the same vein, etc. Even when we look at the system's aspects or views (functional aspect, security aspect, deployment aspect, testing aspect, etc), we consider those in their turn as systems, just like various systems of the human body.

This approach seems to have served us well for a few centuries of industrial manufacturing. It has been used in software for decades, too. Is there a problem at all?

It is commonplace that once an upfront software architectural blueprint is eventually implemented, the result often bears little semblance to the original design, due to all the implementation details, adjustments, special cases, undocumented, but necessary features, etc.

The agile approach deals with this inconsistency, suggesting that we should elaborate and beef up the software architecture of our system in incremental steps in parallel with its actual implementation, adapting it according to the feedback from the on-going implementation and changing requirements and therefore architectural design and implementation happen in parallel in a tight incremental loop. This makes the system design more adequate, but very tedious to maintain, overburdened with details, and frankly, from my observations, even those engineers who complain about the lack of documentation are slack at doing their own bit, when it comes to updating the design diagrams.

Thus, since the primacy of the architectural approach does not seem to match our daily practice, the question to ask is:

> When we design software, do we really work on some sort of architecture first, or *are we essentially doing something else*, and the architecture comes only as a convenient afterthought or supporting act?

To get closer to the answer, let us ask ourselves why in the first place we even need to design a 'system architecture'.

Because it represents the system that we are trying to build? But from all the experience, it does not. I wrote on some of the conceptual reasons for it in 'On Software Design, Space, and Visuality' [1] and 'Two Sides of the Code' [2] and as we discussed just now, practice shows that the final implementation most of the time only vaguely and inadequately is reflected in the upfront or even incremental architectural design.

It seems that the motivation of the software design is not so much accuracy, but simplicity, the need to grasp a complex system in simple terms, before it is even built, so that the customers, management, and project team could understand and share various aspects of the product as a whole in the course of the project. Therefore, the simplicity of expression is perhaps the primary goal of the architectural design.

Some of the tools to achieve this simplification are various types of UML diagrams, state machine diagrams, database diagrams, etc. Their usage is demonstrated with elegant pictures in textbooks. However, in most real-life designs, the number of states, tables, or connections renders the diagram an illegible dog breakfast, especially if you would like to maintain the consistency of the documentation and the code. The alternative is to leave less essential connections out of picture, but that usually makes the diagram too generic to communicate much beyond creating in the upper management the impression that the system is 'designed' and 'planned'.

Therefore, we need simplicity to be able to work on a software system, but representing it as an 'architecture' does not quite scale: it either has too many entities and connections there, or, when some of them are hidden, it does not tell you much.

Let us re-formulate what we are trying to achieve: we need a simple adequate way to work on a software system.

Programming essentially is a pragmatic language activity, as I tried to show in 'Two Sides of the Code' [2]: every statement we write in C++ or

VSEVOLOD VLASKINE

Vsevolod Vlaskine has over 15 years of programming experience. Currently, he leads a software team at the Australian Centre for Field Robotics, University of Sydney. He can be contacted at vsevolod.vlaskine@gmail.com

Generating Code from a Unit Test (Part 3) (continued)

Listing 14

```
public class className : AnInterface
{
  public bool DoIt()
  {
    return true;
  }
  public bool Verify(Int32 a)
  {
    return a == 1 && a != 2;
  }
}
```

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

Python, being executed by the computer, *does* something. Thus, tracing how the design methods plug in into language (both natural and artificial) may help us to understand how to achieve a simpler, more structured, and more adequate view of a software system.

Simplicity of language. Laplace and the Centipede

Laplace's demon is a 19th century hypothesis: if an omniscient demon knew the positions and speeds of all the particles in the universe at a given moment, he would be able to calculate the exact state of the universe at any time in the past and future.

This hypothesis does not hold because of the laws of thermodynamics and quantum mechanics.

However, the software version of Laplace's demon [3] still persists: if we have the full source code of a software system, it seems that we should be able to tell exactly what the system does. Without going into theory, from practice we see most of the time how the projects that started with an architectural design end up with the convoluted code base that requires a score of engineers – a bunch of full-time Laplacian demons – and even they are not able to answer how exactly the product would behave in any given situation. There also is no way to thoroughly document its endless state space.

The software version of Laplace's demon fails due to the delusion that a system can be accurately represented by its description. The full documentation trivially – and uselessly – equates to the full code. It happens because the language elements just do not behave like physical objects [2]. The complexity that we are trying to limit in our design effort is not the complexity of things (e.g. of a car, or of the human body), but the complexity of language.

Unlike physical objects, breaking down into parts does not necessarily simplify things in language. It is the Centipede's predicament: when asked how she walks with so many legs, the centipede became so confused, while reflecting upon her movement, that she could not walk anymore. It is a well-known psychological effect [4], but it is equally true for the software projects that get stifled from the analysis paralysis of the top-down design.

Without the risk of being misunderstood, we say 'walk', instead of 'contract such and such muscles in such and such sequence'. And we say 'go to the pharmacy' instead of 'walk 100 metres, turn left, walk another 200 metres, etc...'. The language actually is very simple: just with a few words it helps us efficiently act in immensely complex contexts.

Capability deployment. Brandom

For a good design, we need to be able to speak about the system in simple meaningful terms.

What we need is the right vocabulary. The difference between a system architecture and a vocabulary is that the architecture essentially captures parts of the system wired up by the relations between them, whereas the vocabulary is like a toolbox: there is no need to draw relationships or connections between a hammer and saw, but when used together they let one do all types of carpentry.

At the design level, the vocabularies should be sufficient for the client to tell what he wants to do with the system, i.e. tell user stories. The design task is to express what needs to be done to fulfil a collection of user stories.

If we are asked to implement an online clothing store, perhaps with a virtual fitting room, we will need a web interface, some image processing algorithms, a database, a payment system, etc. We would go through each user story, making sure these items would certainly cover the required functionality, once they are are cobbled together somehow. Rather than drawing a system diagram, it is sufficient for us to know that we can compose those items with each other in various ways to cover user stories.

Now, we have an adequate bullet-point list, the implementation vocabulary (website, database, etc) for the user story vocabulary (find, display, fit, purchase, etc).

It is important that there is not a single user vocabulary, but a number of them. In our example, the payment-related user stories perhaps will have nothing to do with the fitting-room user stories. Those two will form separate vocabularies or mini-languages. In the same way, the implementation vocabularies also are many: choice of the front end is pretty much decoupled from the database, etc.

As the next step of working on our product, we write user stories regarding website, database, etc necessary to be able to accomplish the online-store user stories. What was an 'implementation' vocabulary, becomes a 'user' vocabulary.

Say, for the user story 'fit a shirt', the 'implementation' stories might be: 'let user choose her/his figure type', 'upload user photo', 'morph the shirt image', etc. These 'implementation' stories become user stories of the next level. We need to devise a list of things that would be sufficient to satisfy them. This list will become the next 'implementation' vocabulary, etc, until at a certain step we realize that we can express our vocabularies as library entries or collection of applications, rather than todo lists in the natural language. Importantly, it happens *not* when the vocabularies and stories become 'simple enough' and without further analysis we can jump to their implementation, but when the stories simply can be written down in the programming language rather than in the natural language. This design process is two-way: not only top-down, departing from client user stories, but also from the point of view of building clear-cut capabilities (of the organization, the team, and the codebase).

Robert Brandom in his book *Between Saying and Doing* [5] makes this process more formal. Although his book is generally about language and meaning, it is very relevant to software engineering.

One of the main concerns of the language theory addressed in Brandom's book is the question of meaning: how do the words mean anything? Following Wittgenstein, Quine, and Sellars, Brandom says that semantics, the meaning of the words in a vocabulary, has to be established through their pragmatics, i.e. how those word are actually used in the language, 'the *practices of deploying* various vocabularies rather than the *meanings* they express' [5, p.3].

Since software engineering is a practical thing, it is simpler and more radical, in a way. It is simpler, because, unlike a theoretical discipline, whatever works better represents a sufficient validation of its method. It is more radical, since 'use' in software means not just 'use in language': all statements and expressions in a programming language *do* something (they are performative in Austin's sense [6], [2]). Pragmatics of programming are much stronger than that of generic phrases like 'it rains', since in software development everything needs to be not only meaningful, but also effectual.

Further in his book, Brandom explores the relationships between the meanings of various vocabularies. He maintains that the meaning is not self-contained in language, but is in its usage, its pragmatics, showing that various vocabularies are pragmatically mediated, calling it 'pragmatic expressive bootstrapping'.

In 'Two Sides of the Code' [2], I wrote about two sides of software code: it tells us something as well as being executable and therefore *does* something, calling these two sides 'expressive' and 'performative', which is practically synonymous to Brandom's terms.

If we have a vocabulary that represents certain capabilities (e.g. various software concepts: database, website, algorithms, etc) and want to express a new vocabulary of meanings (e.g. elements of our online store), we need to deploy the former vocabulary, i.e. in software terms, simply by using the former to implement the elements of the latter. As we saw above, the first vocabularies can be implemented by deploying yet another implementation vocabulary, etc.

While not so obvious in the philosophy of language, this step seems almost trivial in the context of software engineering.

Some conclusions

Since the isomorphism of the software feature-implementation relationship with the meaning-making structure is so trivial, **design and implementation of software systems conforms with the structure of**

Standards Report

Mark Radford reports on the latest from C++14 standardisation.

Hello, welcome to this edition of my standards report, and a happy new year to all the readers.

Since my first report as Standards Officer, a year and a half ago, my reports have focused on the C++ standards process. In that first report I wrote: "would anyone reading this, who is involved in a relevant standards process, please make themselves known to me?". Unfortunately no one has been in touch, hence the focus remains on the C++ standards process – because I'm involved in it, and therefore I can write about it. Once again though, if you have knowledge of any other standards processes that should be reported on in this column, would you please contact me? I can be emailed at: standards@accu.org.

As some readers may be aware, the BSI C Panel has been not been active in quite some time. I'm not sure how long it's been, but I think it amounts to years rather than months. However, that is all about to change because that panel is now reforming. I got in touch with the reformed panel's convenor and obtained some information on what is going on in WG14 (http://open-std.org/JTC1/SC22/WG14), the international C standards committee (the C equivalent of C++'s WG21).

Things currently being worked on include:

- Processing defect reports for C11
- C IEC 60559:2011 (IEEE 754-2008) floating point bindings. There are five parts, two of which have reached the ballot stage. Part 1 (binary floating point arithmetic) is at the Draft Technical Specification (DTS) ballot stage, while Part 2 (decimal floating point arithmetic) is at the Preliminary Daft Technical Specification (PDTS) ballot stage
- C language extensions to support parallelism. There is a study group working on this, but currently the work has not reached the level of an actual proposal to WG14
- A secure coding rules for C specification (TS 17961:2013) was recently published. This is a specification for static analysers.

Currently no work is being done on the next major revision of the C standard. The convenor did comment that WG14 will no doubt start that work at some point. The floating point and parallelism work currently being done is aimed at Technical Specifications (see my May 2013 standards report for more information on what that means). Once the Panel has their mail reflector up and running (they have asked ACCU to host it for them), they will be working on UK positions for the ballots on the floating point bindings.

Unfortunately this edition of my report will be short but, before I finish, I'll briefly return to the C++ standards process...

In my previous report I mentioned that the post-Chicago mailing was (at the time of writing) yet to be published: it is now available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/#mailing2013-10. Readers may remember that in my post-Bristol report, I wrote about Andrew Sutton's 'Concepts Lite' presentation I had seen. I also mentioned that the Concepts study group (SG8) were working towards a TS rather than getting Concepts Lite into the C++14 standard. The first (early) draft of this TS (N3819) is now available in this mailing.

That's it for this report. However there is an ISO C++ meeting coming up in mid February 2014, so hopefully production schedules will permit me to write something about this in my report for the March 2014 *CVu*.

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

Architectureless Software Design (continued)

meaning-making in language. Once again confirming that software design is essentially a language activity and therefore the linguistic laws and concepts are highly applicable here.

The purpose of the software design, rather than drawing 'relationships' between the entities like database, website, etc, is deploying a collection of vocabularies or mini-languages on top of each other that eventually would be sufficient to build a vocabulary fulfilling the product user stories. Libraries and collections of utilities are good examples of such mini-languages: **software design is not about building system architecture, but about defining a collection of mini-languages**. Those mini-languages (closely related to Pattern Languages) express not only the user features of a software product, but all its implementation details.

One mini-language is expressed through deploying another one. However it is not their hierarchy that matters, but the ability to produce new statements. Capturing a limited number of **relations and connections** in a system architecture is an unnecessary semantic coupling: too much too early. Instead, a good designer keeps the number of predefined relationships in the system to the absolute minimum and deploys decoupled, semantically sufficient capabilities – libraries, concepts, naming conventions, domains of natural language, etc – as performative phrases, statements, or user stories.

(Another interesting corollary is that the whole idea of ontologies as the formal representation of 'knowledge as a set of concepts within a domain, and the relationships between those concepts' [7] very likely may be a dead end.)

Lots of things in this article may look trivial, but over years I noticed that far too many software engineers seem to think that if something is trivial, it is not worth doing (because after all they are here to excel in the complex stuff), whereas to me, one of the main virtues in software design is the ability to speak simply about complex things. ■

References

[1] V. Vlaskine, 'On Software Design, Space, and Visuality' *CVu*, vol.25, issue 3, July 2013

[2] V. Vlaskine, 'Two Sides of the Code' *CVu*, vol.25, issue 4, September 2013

[3] http://en.wikipedia.org/wiki/Laplace's_demon

[4] http://en.wikipedia.org/wiki/The_Centipede's_Dilemma

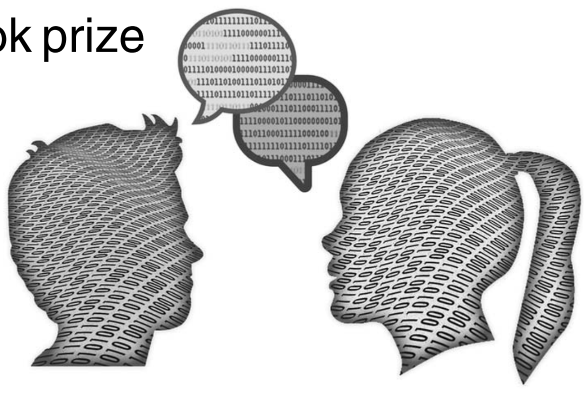
[5] R. Brandom, *Between Saying and Doing: Towards an Analytic Pragmatism*.

[6] J. L. Austin, *How to Do Things with Words*.

[7] http://en.wikipedia.org/wiki/Ontology_(information_science)

Code Critique Competition 84

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.

Last issue's code

I want to use a data structure as a key in a map, but it isn't working as I expect. I've simplified the code down to a shorter example that shows the problem – can you help explain why?

I'm expecting to get output of:

```
Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} => p57x
```

But I (sometimes!) get output like this instead:

```
Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} =>
```

but I don't understand why. I've added some extra output at the end but I still don't see it.

The code is in Listing 1.

Listing 1

```
#include <iostream>
#include <map>
#include <memory.h>
#include <sstream>
#include <string>

// This is the Key -- it is "plain ole data"
struct Key
{
  char ch;
  int i;
  char ch2;
};

typedef std::map<Key, std::string> Map;

std::ostream & operator<<(std::ostream & os,
  Key const & key)
{
  std::cout << "{" << key.ch << ","
    << key.i << "," << key.ch2 << "}";
  return os;
}

bool operator<(Key lhs, Key rhs)
{
  return memcmp(&lhs, &rhs, sizeof(Key)) < 0;
}

// Add an item to the map
void add(Map &map, Key key)
{
  std::ostringstream oss;
  oss << key.ch << key.i << key.ch2;
  map[key] = oss.str();
}
```

Listing 1 (cont'd)

```
// Add an item to the map
void add(Map &map, char ch, int i, char ch2)
{
  Key key;
  key.ch = ch;
  key.i = i;
  key.ch2 = ch2;

  add(map, key);
}

int main()
{
  Map map;

  Key k1 = {'c', 42, 'y'};
  add(map, k1);

  add(map, 'p', 57, 'x');
  Key k2 = {'p', 57, 'x'};

  std::cout << "Testing k1 " << k1 <<
    " => " << map[k1] << std::endl;
  std::cout << "Testing k2 " << k2 <<
    " => " << map[k2] << std::endl;

  // Testing the first element
  Key k = map.begin()->first;
  std::cout << "First: " << k <<
    " => " << map[k] << std::endl;

  k = map.rbegin()->first;
  std::cout << "Last: " << k <<
    " => " << map[k] << std::endl;

  std::cout << "Contents of the map:\n";
  for (Map::iterator it = map.begin();
    it != map.end(); ++it)
  {
    std::cout << it->first <<
      " => " << it->second << std::endl;
  }
}
```

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

Critiques

Paul Floyd <paulf@free.fr>

On my first scan over the code, I saw **`struct Key`** and I thought '**`char int char`**, that won't pack nicely'. Then I saw the **`operator<`** used for the map and I thought '**`memcmp`** on a **`struct`** that is not nicely packed, looks bad'.

This is the crux of the problem. **`struct Key`**, on 32bit systems with 4byte alignment has a memory layout that looks like

```
char ch; <1 byte>
<3 bytes padding>
int i; <4 bytes>
char ch2; <1 byte>
<3 bytes padding>
```

for a total size of 12 bytes.

I'll get back to the subject of member data ordering at the end.

`std::memcmp` is going to compare all of the **`struct Key`** memory, including the 6 padding bytes. So the big question is, what goes into those bytes? This of course depends on how the key is created.

In the 2nd overload of the function **`add`** there is

```
Key key;
key.ch = ch;
key.i = i;
key.ch2 = ch2;
```

so **`key`** is created on the stack and its fields are default initialized (Key has a trivial compiler synthesized constructor which default initializes the POD members, which in practice means that they are left uninitialized). This means the padding is junk. Then the fields are assigned values.

Getting down to some assembler [line numbers probably different from the original source], I see

```
/ Line 57
  movsbl  12(%ebp),%eax
  movb  %al,-16(%ebp)
.L71:
/ Line 58
  movl  16(%ebp), %eax
  movl  %eax, -12(%ebp)
.L72:
/ Line 59
  movsbl  20(%ebp),%eax
  movb  %al,-8(%ebp)
.L73:
/ Line 61
  pushl  $0
  pushl  $0
.L_y38:
  pushl  -8(%ebp)
.L_y39:
  pushl  -12(%ebp)
.L_y40:
  pushl  -16(%ebp)
  movl  8(%ebp), %eax
  pushl  %eax
  call
__1cDadd6FrnDstdDmap4nDKey_n0AMbasic_string4Ccn0ALc
har_traits4Cc__n0AJallocator4Cc____n0AEless4n0B___n
0AJallocator4n0AEpair4Ckn0B_n0E_______1_v_
```

So, 2 1-byte moves for the 2 **`char`**s and a long move (32bit) for the **`int`**. It looks like the struct is copied by value onto the stack for the call to map using 3 long (32bit) pushes, so the padding gets copied as-is.

Then there is the **`Key`** that is passed to the 1st overload of **`add`**. This gets passed a **`Key`** that is initialized as

```
Key k1 = {'c', 42, 'y'};
```

This time the members are value initialized.

```
/ Line 69
.L_y45:
.L_y60:
  movl  .LI203, %eax
  movl  %eax, -28(%ebp)
.L_y46:
.L_y61:
  movl  .LI203+4, %eax
  movl  %eax, -24(%ebp)
.L_y47:
.L_y62:
  movl  .LI203+8, %eax
  movl  %eax, -20(%ebp)
```

3 long moves (32bit).

Looking at the data associated with that

```
.LI203:
  .byte  0x63
  .zero  3
  .4byte  0x2a
  .byte  0x79
  .set  .,.+3
  .type  .LI203,@object
  .size  .LI203,12
  .align  4
```

This looks to me like the padding is explicitly set to zero (though I'm not so certain for the **`.set`** that corresponds to the trailing padding).

Getting back to the C++ code, in this extract

```
add(map, 'p', 57, 'x');
Key k2 = {'p', 57, 'x'};

std::cout << "Testing k2 " << k2 <<
  " => " << map[k2] << std::endl;
```

the **`Key`** added to the map and the **`Key k2`** are initialized differently , so when **`operator[]`** is used with **`k2`**, **`memcmp`** compares differently (junk padding bytes) and a new entry is added to the map.

If by chance the padding bytes of **`k2`** and the **`Key`** used for the map match, then the output will be as 'expected'.

Just to confirm the issue, I ran the program with dbx and memory access checking on (**`access -check`**) and I got:

```
Read from uninitialized (rui):
Attempting to read 1 byte at address 0xfeffe5b8
    which is at top of stack
stopped in std::_Rb_tree<Key,std::pair<const
Key,std::basic_string<char,std::char_traits<char>,s
td::allocator<char> >
>,std::_Select1st<std::pair<const
Key,std::basic_string<char,std::char_traits<char>,s
td::allocator<char> > >
>,std::less<Key>,std::allocator<std::pair<const
Key,std::basic_string<char,std::char_traits<char>,s
td::allocator<char> > > > >::key_comp at line 355
in file "_tree.h"
  355     _Compare key_comp() const { return
_M_key_compare; }
```

How to fix the problem? Well, there are several possibilities. One might be to make sure that the padding is always initialized to the same value, e.g., by allocating with **`std::calloc`** or explicitly setting all bytes in **`Key`** with **`std::memset`**. I don't like this as it is more a reactive cure than pro-active prevention. Slightly better would be to add explicit padding members like **`char pad1[3];`** and to write a user defined constructor(s) which always initialize the padding to a known state. Still, this is a maintenance headache.

Another possibility would be to get rid of the padding. This is easier said than done, particularly for padding at the end of data structures. If you do use something like `#pragma pack`, this will have a performance overhead as most CPUs work best at their natural word size (which is why the padding was added in the first place).

Lastly, the best fix is not to use `std::memcmp` for the `Key operator<`.

The 'canonical' way of supplying an `operator<` for a data structure is to perform memberwise `operator<`, in this sort of fashion:

```
if (lhs.ch == rhs.ch)
{
  if (lhs.i == rhs.i)
  {
    return lhs.ch2 < rhs.ch2;
  }
  else
  {
    return lhs.i < rhs.i;
  }
}
else
{
  return lhs.ch < rhs.ch;
}
```

I noticed that the function calls use pass by value for `Key`. Pass by reference would be more efficient, using a `const` reference.

To wrap up, I'll get back to the question of ordering of data members. To my mind, you have 3 reasons for choosing your ordering:

1. Readability – putting logically related members together to aid understanding.
2. Optimize for speed. Place members such that they play nice with the cache.
3. Optimize for space. Roughly speaking, try to group members of the same size together to minimize the amount of padding inserted. E.g., in this case, `Key` would be smaller if the two char members were grouped together:

   ```
   struct Key
   {
     char ch;
     char ch2;
     int i;
   };
   ```

This would give a size of 8 bytes:

```
char ch; <1 byte>
char ch2; <1 byte>
<2 bytes padding>
int i; <4 bytes>
```

This is better, but it still doesn't make the padding problem go away.

Björn Fahller <baccu@fahller.se>

The key to understanding the problems with this code, is to understand how the compiler generates the data layouts, and how Map's `operator[]` works. Assuming 32-bit integers and 8-bit chars the type

```
struct Key
{
  char ch;
  int i;
  char ch2;
};
```

will be represented in memory as:

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| ch| / | / | / |       i       |ch2| / | / | / |
+---+---+---+---+---+---+---+---+---+---+---+---+
```

`i` takes up 4 bytes, `ch` and `ch2` one byte each, but there are unused gaps marked `/`. The gaps are padding to ensure correct alignment of members, in this case the integer member `i`. These gaps are not required to be initialized to anything, and are unlikely to hold any predictable values.

There are several problems with this, but the obvious one for this example is the comparison:

```
bool operator<(Key lhs, Key rhs)
{
  return memcmp(&lhs, &rhs, sizeof(Key)) < 0;
}
```

`memcmp()` will compare the memory range, byte by byte, until a difference is detected. Comparing two `Key` objects initialized with the same values are likely to have differences in the padding mentioned above.

This becomes important with `std::map<Key, std::string>`, since map uses `operator<` on the key for all lookup and insertion operations, and this is probably what goes wrong in the program. `Map::operator[](Key)` will create a new element if none is found. Due to the uninitialized gap-bytes, it is likely that this happens, and this would then return a newly default constructed, and thus empty, string.

Depending on the requirements of the user, it may or may not be enough to ensure that the gaps have predictable values. If the sort order is important, and if the program is to be portable, the sort order of the integer field will depend on the byte order of words (endianness) of the CPU.

Writing a correct `operator<` for multiple-fields is not difficult, but it is tedious:

```
bool operator<(Key lhs, Key rhs)
{
  if (lhs.ch < rhs.ch) return true;
  if (lhs.ch > rhs.ch) return false;
  if (lhs.i < rhs.i) return true;
  if (lhs.i > rhs.i> return false;
  return lhs.ch2 < rhs.ch2;
}
```

If a c++11 compliant compiler is used, it is better to use the `tie()` function template to generate tuples, for lexicographical comparisons are generated:

```
bool operator<(Key lhs, Key rhs)
{
  return std::tie(lhs.ch, lhs.i, lhs.ch2) <
         std::tie(rhs.ch, rhs.i, rhs.ch2);
}
```

There are more things to say about the memory layout, though. Half of the 12 bytes required by `Key` are wasted. Changing the order of the members in `Key` will reduce the memory consumption:

```
struct Key {
  int i;
  char ch;
  char ch2;
};
```

will be laid out as (again assuming 32-bit integers)

```
+---+---+---+---+---+---+---+---+
|       i       | ch|ch2| / | / |
+---+---+---+---+---+---+---+---+
```

Note that there are still two uninitialized gap-bytes, making the structure require 8 bytes instead of 6 as one might have guessed. The reason for this is again to get the correct alignment for the integer i. This is not obvious when looking at one `Key` in isolation, but look what happens when several are used in an array:

```
Key keys[2]

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|       i       | ch|ch2| / | / |       i       | ch|ch2| / | / |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 \          keys[0]           / \           keys[1]           /
```

Still, the wasted space is reduced from 6 to 2 bytes per object.

The existence of these gaps still renders `memcmp()` unsuitable for `operator<()`, although it is possible if you insist, by ensuring that only the memory occupied by initialized data is compared. This can be done on POD types using the `offsetof` macro.

```
bool operator<(Key lhs, Key rhs)
{
  return memcmp(&lhs, &rhs,
    offsetof(Key, ch2) + 1) < 0;
}
```

`offsetof(Key, ch2)` will evaluate to 5, since `ch2` is at that offset in the memory layout, and then add 1 for the size of `ch2`. I strongly advise against using this technique, though, since it is horrendously brittle in the face of change, by no means obvious to the reader, and still not portable across CPU-architectures if the sort order of the integer field is important. Experiment with the technique to learn, and then go with `std::tie()` instead.

Ken Duffill <kenduffill@gmail.com>

Let's start with the output you see (sometimes!):

```
Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} =>
```

What does this tell us?

Looking for the code that generates this output we find:

```
std::cout << "Testing k1 " << k1 << " => " <<
  map[k1] << std::endl;
std::cout << "Testing k2 " << k2 << " => " <<
  map[k2] << std::endl;
```

Clearly, the output tells us that `k1` and `map[k1]` are both OK.

And we can see that `k2` is fine, but `map[k2]` is not. Why would that be?

Let's first try to find out what `map[k2]` really means.

`std::map` is being used as an associative array here. The index of the subscript operator is the key that is used to identify the element. But there is a gotcha when using `std::map` as an associative array.

When using the key as the index for which no element yet exists, a new element is inserted in the map automatically. This can be very useful, as it is when the method `add` is called to add an entry into the map; which it does with the following line of code:

```
map[key] = oss.str();
```

What this actually does is tries to overwrite the element of `map[key]` with the new value. As this key probably doesn't exist `std::map` creates one with the default `std::string` (ie an empty string) as the element.

Then the code assigns the string `oss.str()` (which was created from the members of `struct Key`) to the element of the newly created map entry.

This may be a little inefficient, using the `insert` method of `std::map` may be more efficient, but you may have deliberately chosen to do it this way, because `insert` will fail if the key already existed whereas what you are doing will not, it will just overwrite the element of the existing entry with the newly created element. Your choice.

But in the line of code

```
std::cout << "Testing k2 " << k2 << " => " <<
  map[k2] << std::endl;
```

`std::map` would create an entry if we are accessing a non existent key and the element again would be created using the default constructor for type `std::string` (an empty string). This is what we see with the output

```
Testing k2 {p,57,x} =>
```

`map[k2]` returns an empty string, because `k2` did not exist in the map. But why does `map[k2]` not exist?

Looking for the declaration of `k2`, that looks OK, but wait, we are not adding `k2` to the map! We are adding a temporary `struct Key` made from the same parameters, which looks to be identical to `k2`, but is not `k2`, to the map.

So how would `std::map` find an entry in the map, and why does it not consider the temporary `struct Key` to be the same as `k2`?

The answer is that `std::map` considers two entries to be equal if neither entry is less than the other. That is why we have to declare the `operator<` method.

Now we get to the coding error that is causing all our problems. And, incidentally, is a problem which I actually encountered in my real work very recently (somebody else's code, of course).

This implementation of `operator<` makes the fundamental error of assuming that `struct Key` contains memory only for the declared items. This is wrong for two reasons: Alignment and padding. Most processors expect and therefore the compiler will ensure, that `int` data items are aligned on an `int` address boundary. That is to say that on a 32-bit machine the address of an `int` will always be a multiple of 4. They will also start a `struct` on the same boundary. Because you declare an `int` after the `char` which is the first element in the `struct`, the `int` cannot start on the next memory address as that would not at be an address that is a multiple of four. Therefore there will be some unused space between `ch` and `i` in `struct Key`.

Similarly at the end of the struct, there may be some padding. Often the compiler will round up the size to a multiple of four bytes, for its own convenience. In general you do not know this and therefore should not rely on there being no unused space at the end of your declarations in a structure.

Now, if there are padding and alignment issues in your structure then that extra memory will not be initialised, and therefore may or may not compare correctly when comparing with another `struct Key`. The contents will be entirely unpredictable. This is where the '(sometimes!)' comes in in your unexplained output.

There are usually `#pragmas` to tell the compiler not to do these things, they cost (quite a lot) in processing, but are useful if it matters to you that there is no unused space. For example, when constructing complex data to send down some data channel where bandwidth is an important consideration. And, you can also help the compiler by reordering the declarations so that big items come first and the smaller ones are at the end.

But a better solution is to change the implementation of the `operator<` so method that instead of using `memcmp` to compare all the memory (including the uninitialised memory) in `struct Key` it just compares each element within one structure with its corresponding element in the other. For example:

```
bool operator<(Key lhs, Key rhs)
{
  if (lhs.ch != rhs.ch)
  {
    return lhs.ch < rhs.ch;
  }
  if (lhs.i != rhs.i)
  {
    return lhs.i < rhs.i;
  }
  return lhs.ch2 < rhs.ch;
}
```

Hey presto, your problem goes away.

So, you thought I had finished, did you? No, there is more. I have a hankering to do some refactoring.

Ideally we should have some unit tests in place that we would run between each refactoring in order to prove that the behaviour hasn't changed, but as I have neither the time nor the space here to create those unit tests here I will leave that as an exercise for the reader.

Refactoring 1

The new `operator<` method takes two parameter, both are `Key`s, and both are passed by value. There will be a small processing overhead in passing by value, as this means that the contents will be copied every time the method gets called.

A small change to improve performance, just a little, is to pass the parameters by reference. Further, as the contents of both keys are not modified, they are only looked at, then they should be `const` references. Like this:

```
bool operator<(Key const &lhs, Key const &rhs)
{
  if (lhs.ch != rhs.ch)
  {
    return lhs.ch < rhs.ch;
  }
  if (lhs.i != rhs.i)
  {
    return lhs.i < rhs.i;
  }
  return lhs.ch2 < rhs.ch;
}
```

Compile and run the code (and your unit tests if you have created them), and see that the behaviour hasn't changed.

Refactoring 2

Now we find that this new `operator<` method can be made a member of `struct Key`. To do this we lose the `lhs` parameter, but we must remember to make the method itself `const` so as to tell the compiler that running this method doesn't change the internals of the object.

It is always good practice to make methods that don't change the internal state of a class or struct `const`, it is a good habit to get into, but in this case it is actually essential otherwise `std::map` will not recognise it as the comparison function it needs. Move its declaration into the struct declaration and `operator<` now looks like this:

```
bool operator<(Key const &rhs) const
{
  if (ch != rhs.ch)
  {
    return ch < rhs.ch;
  }
  if (i != rhs.i)
  {
    return i < rhs.i;
  }
  return ch2 < rhs.ch;
}
```

Compile and run the code (and your unit tests if you have created them), and see the behaviour hasn't changed.

Now there is a large part of me that wants to keep refactoring until I can get the data members of `Key` to be private. BUT, I realise that that is actually quite a lot of work and, more importantly, most of that work is 'fiddling about with' the strings that are used to create or output not the `Key` itself, but the element part of the map that is generated from the `Key`.

It is a good idea at this point to keep in mind the single responsibility principle (SRP).

`Key`'s single responsibility is to maintain the `Key` that will go into the map. In order to do that it must be able to construct a `Key` and compare two keys. That is all it needs to do. The fact that in this test case the parts of the `Key` are also used in the element, is an implementation detail of this particular application, and probably only temporary during the development of the app. So there is no profit in wasting time making the data items private (not right now, anyway).

The only other refactoring I would like to do is to give `Key` a constructor. You will see, shortly, how this really benefits the code structure for the rest of the app.

Refactoring 3

Add a constructor for `struct Key` like this:

```
Key(char chParam, int iParam, char ch2Param)
{
  ch = chParam;
  i = iParam;
  ch2 = ch2Param;
}
```

In order for app to still work we must change the declarations of `k1` and `k2` thus:

```
Key k1('c', 42, 'y');
Key k2('p', 57, 'x');
```

Finally, we need to change the creation of the temporary `struct Key` in the method `add(Map &map, char ch, int i, char ch2)` to `Key key(ch, i, ch2);`

Now the app code is starting to look much simpler, isn't it?

Compile and run the code (and your unit tests if you have created them), and see the behaviour hasn't changed.

Refactoring 4

Now we don't even need the variable `key` to hold the temporary `struct key` in the method `add(Map &map, char ch, int i, char ch2)`, we can create that inline thus: `add(map, Key(ch, i, ch2));`

Finally you can now see that the method `add(Map &map, char ch, int i, char ch2)` isn't even necessary by changing

```
add(map, 'p', 57, 'x');
```

to

```
add(map, Key('p', 57, 'x'));
```

The second `add` method can be removed and the app code becomes even simpler, cleaner, and easier to maintain.

Oh, and once again, to avoid the unnecessary copy in the `add` method, we should make the `Key` parameter passed by `const` reference instead of by value.

Compile and run the code (and your unit tests if you have created them), and see the behaviour hasn't changed.

Now I feel better, I will stop.

James Byatt <grumpyjames@gmail.com>

Ah, maps. A celebrated source of C++ esoteria. I first came across this CC on a train, and, as such, only had my own personal prejudices to help me out. Here's the JamesLINT output. Do be warned though – JamesLINT is a bit rusty, and has also been known to be somewhat jaded by prior experience, rather than standard led.

Problem on line 3: 'When in Rome...'

Importing C headers is almost always a bad sign; not least because I have little idea what they do, or the motivation to find out. A whole bundle of people far smarter than I have spent many man years of work wrapping up this `memcpy` nonsense in more pleasant abstractions, it seems wholly rude to reintroduce these crudities without first attempting to find the appropriate C++ idiom.

Problem on line 8: `Key` should be immutable

Const the fields, provide a constructor. What is this, the 90s?

Problem on line 25: Why aren't those `Key` inputs references to `const`?

Problem on line 27: See problem #1

Ugh. Would it have been that hard to delegate to the relevant operators for `char` and `int`? I seem to remember there being a library function for doing comparisons of arbitrary numbers of fields, as well.

I wouldn't be surprised if this were the problem (possibly because of some alignment bits in the binary representation of `Key`).

Problem on line 35: Using `[]` to insert into a map.

The mixed semantics of the operator make it a minefield. It is used for both entry and retrieval, and, when used for insert, default constructs a string, and then copies the value string over it. Woe betide you should your value type not be default constructable.

Recommended: use `insert` with `std::make_pair` instead (or whatever fancy new tricks C++11 offers to avoid copying (emplace?))

Problems in method `main()`:

Lots of using `[]` for map lookup – this will work just fine for strings, but might not in general (I'm willing to bet no-one will remember to migrate all these usages when the value type changes). Use `find()`, please.

Compiler output

Quite a lot of output from JamesLINT, there. What does the compiler have to say for itself?

```
g++ -Wall -pedantic Map.cpp
```

Nothing at all in the way of warnings – now, that is a first.

The usual suspects

My suspicion here is that the various pass by value semantics and that use of **memcmp** is going to be the problem. Let's verify a small part of that suspicion: that **Key** is being padded with some alignment bits:

```
int main()
{
+    std::cout << "Key has size " << sizeof(Key)
     << std::endl;
+    std::cout << "int has size " << sizeof(int)
     << std::endl;
+    std::cout << "char has size "
     << sizeof(char) << std::endl;
```

One might expect that **sizeof(Key) == sizeof(char) + sizeof(int) + sizeof(char)**, however:

```
Key has size 12
int has size 4
char has size 1
```

It is not the case; each field is aligned at a four byte boundary. I am guessing those extra alignment bits are not always the same when we get to that call to **memcmp**; a problem perhaps exacerbated by all the pass by value semantics. Let's see if the problem goes away if we amend that.

```
-bool operator<(Key lhs, Key rhs)
+bool operator<(Key const &lhs, Key const &rhs)
-void add(Map &map, Key key)
+void add(Map &map, Key const &key)

Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} =>
```

Heh – no – we're in no better shape. Without getting the debugger out, let's just try rewriting that operator without the **memcmp**.

Removing memcmp

```
bool operator<(Key const & lhs, Key const & rhs)
{
-    return memcmp(&lhs, &rhs, sizeof(Key)) < 0;
+    if (lhs.ch < rhs.ch) {
+       return true;
+    } else if (lhs.ch > rhs.ch) {
+       return false;
+    } else if (lhs.i < rhs.i) {
+       return true;
+    } else if (lhs.i > rhs.i) {
+       return false;
+    } else if (lhs.ch2 < rhs.ch2) {
+       return true;
+    } else if (lhs.ch2 > rhs.ch2) {
+       return false;
+    }
+    return false;
}

Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} => p57x
```

Aha. Ugly, but correct. If we were in C++11 land, we could make our lives easier by making **Key** a **std::tuple**, and then we wouldn't need to drudge through all that boilerplate. This is left as an exercise for the reader.

So, without really understanding what was broken, we've got a working implementation merely by fixing the more glaring JamesLINT issues. Is that good enough? Of course it isn't. Let's figure out what was going on with that memcmp.

Understanding the **memcmp** problem

```
add(map, 'p', 57, 'x');
Key k2 = {'p', 57, 'x'};
```

Heh. I totally missed that we were building the key objects twice in the main method. Let's use the old crappy **operator<** and pass **k2** in by reference rather than attempting to build it again.

```
-    add(map, 'p', 57, 'x');
     Key k2 = {'p', 57, 'x'};
+    add(map, k2);

Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} => p57x
```

...and that fixes it. Excellent. So the problem was actually creating two completely different instances of the plain old data, and expecting their alignment bits to be the same. Someone with a copy of the standard could probably point you at a place where this guarantee is explicitly not given...

...perennial short cut taker that I am, though, I'm just going to write a quick test program to see what happens on my machine.

Alignment

```
Key k2 = {'p', 57, 'x'};
Key k3 = {'p', 57, 'x'};
print_bytes(&k2, sizeof(Key));
print_bytes(&k3, sizeof(Key));
print_key(k3);
```

Here, **print_bytes** prints a hex representation of each byte in the given pointer, and **print_key** copies the key before passing it to **print_bytes** – implementations elided for being too dull (and, also, because the code I used was copy pasted from stack overflow, and used **printf**, for shame).

```
[ 70 00 00 00 39 00 00 00 78 2e 40 00 ]
[ 70 09 c7 c4 39 00 00 00 78 2f 40 00 ]
[ 70 09 c7 c4 39 00 00 00 78 2f 40 00 ]
```

Here, at least, it looks like alignment bits are preserved on copy, but not zeroed on object creation. That seems sensible enough, although I now feel vaguely silly for thinking that copying **Key** by value would introduce differences given the object layout.

So, what did we learn?

Don't mix abstraction levels! **Key** is a C++ **struct**, an object, while **memcmp** is a function for comparing two completely arbitrary pointers. Implementing **operator<** using something from `<memory.h>` was a poor choice.

Using the standard library correctly is fun and profitable, if occasionally requiring of you to read the manual a wee bit.

Judicious use of **const &** can hide a multitude of sins. JamesLINT will still judge you, though.

It's a good idea to read the whole of the source code before diving in and trying to fix it (oops).

Carl Gibbs <carl@carlgibbs.co.uk>

Why doesn't it work? Because using **memcmp()** to compare the contents of a structure is neither good practice nor correct. By trying to compare raw bytes of memory you're circumventing the C++ language and making assumptions about how the compiler will translate your program.

It's not correct because C++ can add padding to the structure **Key** in memory in order to align fields on machine-word boundaries for the fastest possible addressing and this padding may not be initialised. So on a 32-bit compiler you're likely to really get:

```
struct Key
{
  char ch;
  int rubbish[3];// uninitialised, random values
```

```
    int i;
    // no padding cos int already size of a
    // machine-word
    char ch2;
    int moreRubbish[3]; // uninitialised, random
                        // values
  };
```

memcmp() will compare these random padding bytes and because it's not just comparing the data you expected, can give a different answer than you expect.

Note: Some compilers will always set the same values in the padding bytes, especially in a debug build so the application might work. The release build from the same compiler is still likely to save time by not initialising the padding and so fail.

A second reason is that byte ordering on your machine may make it never correct. An x86 CPU is little-endian so **int i = 256** will put the following in memory (still a 32-bit compiler):

```
char i0 = 0; // least-significant byte first,
             // so memcmp() will not compare
             // integers as expected
char i1 = 1;
char i2 = 0;
char i3 = 0;
```

But a Motorola 68000 is big-endian so **int i = 42** will put the following in memory:

```
char i3 = 0;
char i2 = 0;
char i1 = 1;
char i0 = 0; // least-significant byte last,
             // so memcmp() will compare
             // integers as expected
```

This will leave the **std::map** in the wrong order.

std::map relies on **operator<** working correctly. If 2 Keys that you think are the same don't compare the same then **std::map** won't understand they are the same.

This explains why **k1** behaves as expected even with this bug while **k2** does not. Because we are using a bitwise copy of **k1** in the map, even the bits in the padding, **k1** and the key in the map are identical when compared with **memcmp()**.

```
Key k1 = { 'c', 42, 'y' };
add( map, k1 ); // bitwise copy of k1 in the
        // map with exactly the same padding
```

But **k2** is created separately from the equivalent key in the map, so has different random padding, so **k2** and the key in the map are not identical when compared with **memcmp()**.

```
add( map, 'p', 57, 'x' ); // map with key
// entry {p,57,x} with some random padding
Key k2 = { 'p', 57, 'x' }; // k2 with visible
// values {p,57,x} but with some different
// random padding
```

While most compilers allow you to override whether to put padding bytes in, this approach definitely would be considered 'unobvious magic' or 'reason to hate you' by the next developer working on this code.

A better approach is to ensure you only compare the fields you know about. One easily extended way to write this is:

```
bool operator<( Key lhs, Key rhs )
{
  if ( lhs.ch != rhs.ch ) {
    return lhs.ch < rhs.ch; }
  if ( lhs.i != rhs.i ) {
    return lhs.i < rhs.i; }
  if ( lhs.ch2 != rhs.ch2 ) {
    return lhs.ch2 < rhs.ch2; }
  return false;
}
```

This makes it both correct and clearly shows precedence of the fields to later developers. On a modern optimising, inlining compiler it's also likely to be as fast as the **memcmp()**. Work with the language, don't try to circumvent it.

Brian Ravnsgaard Riis <brian@ravnsgaard.net>

There is one basic misunderstanding that prevents this code from producing the expected result. Aside from that, though, there are some stylistic issues that immediately caught my eye when typing the code to test run it.

To begin with, I am very much at a loss to understand which problem using this convoluted **Key** in a **std::map** is supposed to solve. However, as stated, the code is simplified, so I may not have the whole picture. And I digress.

Typing the entire code into cc84.cpp and throwing g++ 4.8.2 at it gives a clean compile, even with **-Wall** specified. As Bjarne Stroustrup states in *Programming: Principles and Practice*: "If your program has no compile-time errors and no link-time errors, it'll run. This is where the fun really starts."

Indeed. Running it gives this output:

```
Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} =>
First: {c,42,y} => c42y
Last: {p,57,x} => p57x
Contents of the map:
{c,42,y} => c42y
{p,57,x} =>
{p,57,x} => p57x
```

Not quite what we expected. The coder has added some helpful debugging output at the end but failed to gain further insights. However, it should be immediately obvious that there are three elements in the map, two of which have the same key! In other words, the map is corrupted! **std::map** maintains a strict weak ordering on its contents, sorted on the key using **std::less**, which defaults to **operator<()**. The coder knows this, since he has provided an **operator<(Key, Key)**, but still somehow a duplicate has crept in. Let's take a look at this function.

It is a simple call to **memcmp** on copies of the provided **Key** arguments. I have a sneaking suspicion that just comparing those two objects' raw memory (including padding!) is too simplistic, so I try to code it out instead, and immediately come up with a simple, short and wrong answer:

```
bool operator<(Key lhs, Key rhs)
{
  return
    lhs.ch < rhs.ch &&
    lhs.i < rhs.i &&
    lhs.ch2 < rhs.ch2;
}
```

This gives the following output:

```
Testing k1 {c,42,y} => p57x
Testing k2 {p,57,x} => p57x
First: {c,42,y} => p57x
Last: {c,42,y} => p57x
Contents of the map:
{c,42,y} => p57x
```

Well, that wasn't exactly an improvement! Granted, the map is not corrupt any longer, but it certainly does not contain the objects I thought I put in there. Of course, I have now specified that **lhs** is less than **rhs** only if all its members are less than all of **rhs**' members. This is not right at all!

What I want is to compare each member, one by one. If the first is less we're done. If it's greater we're also done and can return false. Otherwise we check the next member. The resulting function looks like this:

```
bool operator(Key const& lhs, Key const& rhs)
{
  return
    lhs.ch < rhs.ch ||
        (lhs.ch == rhs.ch &&
          (lhs.i < rhs.i ||
           (lhs.i == rhs.i &&
            lhs.ch2 < rhs.ch2)));
}
```

Notice I changed one of the stylistic issues as well and passed the arguments by reference-to-const instead of by value. The program now outputs:

```
Testing k1 {c,42,y} => c42y
Testing k2 {p,57,x} => p57x
First: {c,42,y} => c42y
Last: {p,57,x} => p57x
Contents of the map:
{c,42,y} => c42y
{p,57,x} => p57x
```

Bingo. The resulting **operator<** is a bit verbose, but it's correct!

So, let's clean up the rest of the code a bit.

In **operator<<()** we take a reference to some **std::ostream** and then use **std::cout** regardless. Why? We may want to stream the **Key** object to a **stringstream** or a file instead. We should use the supplied **ostream** object. Recompiling confirms that we have broken nothing; we get the same output.

As stated above there is no reason for **operator<** to take **Key** objects by value; reference-to-const will do. And, since we're no longer using **memcmp**, which gave the wrong result anyhow, we can remove the **memory.h #include**.

Finally I would note that the comparison function can be made significantly less verbose if you can use C++11. **std::less** for **tuple**s already does the right thing, so we can just create **tuple**s-of-references to **Key**'s members and use **operator<** directly:

```
bool operator<(Key const& lhs, Key const& rhs)
{
  return
    std::tie(lhs.ch, lhs.i, lhs.ch2) <
    std::tie(rhs.ch, rhs.i, rhs.ch2);
}
```

You need to **#include <tuple>** for this to work portably.

As I stated first, the entire scenario makes me wonder a bit, but at least a correct **operator<** for **Key** objects makes **Key** usable in associative containers. **std::set**, **std::multiset**, and **std::multimap** are all broken for **Key** objects if using the original **operator<**.

Emil Nordén <emilnorden@yahoo.se>

The problem here is a combination of structure padding and the use of **memcmp**.

By using **memcmp** to compare the **Key** structs, a byte-by-byte comparison of the memory is made. Given the current layout of the **Key** struct there will be some padding added to the struct after the fields **ch** and **ch2**, and since the values stored in padded memory are undefined, chances are high that two seemingly identical **Key** instances will not be considered equal when comparing them using **memcmp** because of garbage values in the padded memory.

This also explains why it would appear to work at random times, you were just lucky enough to get the same garbage values in the padded memory!

The solution to this problem is either to rearrange the fields of the **Key** struct:

```
struct Key
{
  int i;
  char ch;
  char ch2;
}
```

or to tell your compiler not to pad the struct (using GCC in this example. For other compilers, consult your manual!):

```
struct __attribute__ (packed) Key
{
  char ch;
  int i;
  char ch2;
}
```

or finally, to implement your own **Key** comparison function without using **memcmp**:

```
bool operator<(Key lhs, Key rhs)
{
  if(lhs.ch != rhs.ch)
    return lhs.ch < rhs.ch;
  else if(lhs.i != rhs.i)
    return lhs.i < rhs.i;
  else
    return lhs.ch2 < rhs.ch2;
}
```

Joe Wood <joew60@yahoo.com>

When I first read the supplied code and accompanying output, my first reaction was 'Ah map', because using a new key with the **operator[]** causes a new entry in the map. However it is not obvious where the new key was being introduced.

However, before getting into the bowels of map, lets do a little re-factoring and tidying up.

Personally, I like local functions to be declared static, just to stop namespace pollution. It might also improve optimisation, but I do not think that is our primary goal here. So prefix the two **Key** functions (**operator<** and **operator<<**) and both **add**s with `static`.

main contains several output lines which are excellent candidates for re-factoring into separate functions. So let's create two new functions called **showElement** to display a map element. The initial code will be

```
  static void showElement(
    const std::string & text,
  const Key & k,
  const std::string & v) {
    std::cout << text << k << " => "
              << v << std::endl;
}
```

and

```
static void showElement(
  const std::string & text,
  Map & map,
  const Key & k) {
    const std::string value = map[k];
    showElement(text, k, value);
}
```

With these two functions in place, most calls to **iostream** can be replaced.[1] Notice in the second overloading of **showElement** we must declare map without **const** because the **operator[]** can update the map.

Having factored out the call to **map[k]**, we can replace it with a call to **map.at(k)**, which is new in C++11 [2]. However, **at** throws an exception if **k** is not found rather than silently adding a default constructed new pair to the map with a key of `k`. Hence for some unknown **k'**, **map[k']** is basically equivalent to **map.insert(std::make_pair (k', std::string())** [3].

Since **at** throws an exception for unknown **k**, we need to catch this since we are not expecting any unknown keys in our map. Hence rewrite the second **showElement** as

```
static void showElement(
    const std::string & text,
    const Map & map,
    const Key & k) {
  try {
      const std::string value = map.at(k);
      showElement(text, k, value);
    }
    catch(const std::out_of_range& e) {
      std::cerr << "Error: the key '" << k
                << "' is not defined in "
                << "the supplied map"
                << std::endl;
      throw;
    }
  }
```

Notice that we have changed the map parameter to be **const**, since we do not want to write into the map. On an exception we have displayed an error message and re-raised the exception. In a production system we might want to do something more appropriate to the underlying system.

Finally we find the first (but not the only bug). When run, the second (if you have been following along with the code changes) call to **showElement** throws an exception. Whoa, what, why? It looks as if we are looking up the newly inserted **Key k2**, but **map** thinks we are adding a new entry. Time to get the debugger out. We know that **map** uses **operator<** to determine where to place new entries and for internal searches.

So a breakpoint on **operator<** and a look at the passed in **Key**s. As expected, we soon get a comparison of what appears to be **k2** versus **k2**. However, looking at the code in **operator<**, it uses **memcmp** and **sizeof(Key)** to compare keys. Now, **sizeof(Key)** (on my system) is 12 bytes, so there is some extra padding hidden inside **Key**, as we would expect **sizeof(ch)+sizeof(i)+sizeof(ch2)** to be (again on my system) 6 bytes. Sure enough, the language specification allows for extra bytes to pad data onto suitable boundaries.

A naïve fix is to change the declaration of **Key** to

```
  struct __attribute__((__packed__)) Key
  {
    char ch;
    int i;
    char ch2;
  }
```

This may work on some machines, but it suffers from three principal problems, viz.

- It is compiler specific.
- It assumes knowledge of memory layout, and little-endian integers.
- It is brittle and not obviously related to the contents of **Key**, adding a new field would (potentially) invalidate existing runtime behaviour.

Let's just pause for a moment and think about **struct Key**. The original code explicitly wanted **Key** to be Plain Old Data (POD). Why? We are not told. Perhaps because it interfaces with some other code, and the layout is important. This is another reason for rejecting the naïve approach. Basically POD must only use C-style datatypes, so that it can statically initialised, and not have virtual functions [4].

Returning to the map lookup problem. Before getting stuck into the comparison function, **operator<**, it is worth reminding ourselves that this must produce a strict weak ordering, in particular if the same **Key** is supplied to both parameters, it must return **false**, since Map determines key equality by

```
  !(k1<k2) && !(k2<k1) => k1==k2
```

memcmp performs a byte by byte lexicographical comparison between the specified memory areas, up to the specified maximum number of bytes, returning zero if both blocks are equal, a negative number if the first is less than the second, and a positive number otherwise. Which is why the naïve padding seems to work.

We want [5] a lexicographical comparison of **ch** followed by **i** and finally **ch2**. Which can be done by

```
  return
    this->ch < k.ch ||
    (this->ch == k.ch && this->i < k.i) ||
    (this->ch == k.ch && this->i == k.i &&
       this->ch2 < k.ch2);
```

That's a bit of a mouth full. Fortunately, the new C++11 tuple library offers some much needed simplification. Given two tuples **t1** and **t2**, we can perform a lexicographical comparison (element by element) just by writing **t1 < t2**. So all we need to do is convert the needed fields of **Key** into a **tuple**, enter **make_tuple**. (We need to add **#include <tuple>** at the top of the code).

So our **Key**'s comparison **operator<** function becomes

```
  static bool operator<(const Key &lhs,
                        const Key &rhs) {
    return std::std::make_tuple(lhs.ch, lhs.i,
                                lhs.ch2)
              < std::make_tuple(rhs.ch, rhs.i,
                                rhs.ch2);
  }
```

Well it's better, but I do not like spelling out **Key**'s fields in two different places. We can write a helper function in C++11, as

```
  #define keyTuple(k) \
    (std::make_tuple(k.ch, k.i, k.ch2))
  auto mktuple(const Key & k) ->
      decltype(keyTuple(k)) {
    return keyTuple(k);
    }
  #undef keyTuple
```

And change **operator<** to be

```
  static bool operator<(const Key & lhs,
                        const Key & rhs) {
      return mktuple(lhs) < mktuple(rhs);
  }
```

The first **add** function can have a reference to **Key** rather than passing it by value, hence

```
  static void add(Map &map, const Key &key) {
  // as before
```

We can make a few minor changes to **main**, with C++11 support. Before declaring map, add the line

```
  static_assert(std::is_pod<Key>::value,
      "Key must be a POD type");
```

This will cause a rejection at compilation time, if **Key** is not a POD. Change the various **begin**/**end**s to **cbegin**/**cend** as we don't want to modify the map. We might as well take advantage of C++ **foreach** loop, e.g.

```
  for (const auto & entry : map) {
    showElement("", entry.first, entry.second);
  }
```

Finally running the resultant code, we get the expected output

```
  Testing k1 {c,42,y} => c42y
  Testing k2 {p,57,x} => p57x
  First: {c,42,y} => c42y
  Last: {p,57,x} => p57x
  Contents of the map
  {c,42,y} => c42y
  {p,57,x} => p57x
```

Notes

[1] Not shown because it is straight forward and lengthy, for example the line **std::cout << "Testing k1 " << k1 << " => " << map[k1] << std::endl;** can be replaced with **showElement("Testing k1 ", map, k1);**.

[2] **map.at** can be simulated with **map.find** in older C++ with some additional logic. Left as an exercise.

[3] There is a semantic difference between **operator[]** and **insert** if the key is already in the map. **insert** will not overwrite the

previously stored value whilst **operator[]** does update the stored value.

[4] Prior to C++11 t he requirements on POD datatypes were even stricter.

[5] Well presumably we want, its not actual explicitly stated in the problem.

Balog Pal <pasa@lib.hu>

It's so much joy to meet a problem intro having words 'I simplified', and presentation of expected and actual behavior. It made it definitely worth taking the trouble to look for the cause. :)

'Map' is also mentioned, for that I usually ask questions in advance of even looking at the code for the usual suspects:

- Did you modify a key sitting inside the set/map?
- Did you use a function that breaks the rules (irreflexive, antisymmetric, transitive for both < and equivalence)?
- Did you invalidate some of the iterators?

In my experience that trio smokes out most of the problematic cases and gained me some 'seer' points, so we will pay special attention to those while scanning the code. This is just a raw run, we'll go deep later.

- In the include section there's an odd **<memory.h>**
- **struct Key** raises 'suboptimal layout' warning placing **int** between **chars**
- **typedef** to name '**Map**' in global namespace
- **operator<<** that takes the stream to write to but writes to **cout** really
- **operator<** that takes arguments by value
- **operator<** uses **memcmp** in implementation without the usual page-full of comments explaining why it will be okay
- strange pair of **add** functions that are hopefully not for production but just helpers to write the test/demo.
- First **add** takes **Key** by value without a good reason
- Second **add** shouts for a **Key** constructor or a **make_Key** function
- aggregate init of **Key** in **main** that's not incorrect but creates fragility
- pasted lines in **main**, all **cout <<** could better be a function taking **k** and one string
- a **for** loop that could use **auto** or even better be replaced by **for_each** or **copy**

That's twelve notes for 80 lines of code, something to work with. Looking back to initial questions, we can put iterators and mutation to rest, but the **<** function looks like the hit, and is one likely source of undefined behavior. A good and hopefully frightening summary of **memcmp** is found here: http://www.codepolice.org/c/memcmp.html with all references, I will just summarize what's relevant for our case:

- **memcmp** compares raw bytes of memory (as **unsigned char**)
- The contents of 'holes' used as padding for purposes of alignment within structure objects are indeterminate.

If used in the compare function for map, the last is a showstopper. As having the indeterminate values in padding sends transitivity down the drain. To have a fighting chance we must make sure to not have any padding holes and all bytes and bits in the memory must be part of our **Key** object. We can't do that just with the standard features, but real implementations may provide tools. For example using MSVC targeting WIN32/x86 we can use **#pragma pack(1)** or a compiler switch /Zp1. With that the **Key** structure will take 6 bytes of memory and fully predictable with **memcmp**. But it's good to know that if we use that same pragma pack with gcc (supported with same semantics) and a SPARC V9 target, we will still get a 3 byte hole between ch and i, and i must be aligned on n*4 address.

Suppose we are on a friendly target and can force a hole-less layout, is it okay to use **memcmp** for the case? It still depends. Now the function will actually provide the required strict weak order and the insert+retrieve cases will work as expected. If our **map** is just used for the associations and was picked over **unordered_map** for unrelated reasons, it's okay. But if we are also interested in the actual order, we're open to more surprises. As **memcmp** works on raw bytes it will give us an order dependent on memory patterns, that may significantly differ from the semantics of **<** on the original elements. Starting with **char ch**, if **char** is signed on our platform it will compare differently in **memcmp** that will reinterpret it as **unsigned char**. The **int** will also face the sign-related problem in x86, where at least **unsigned int** would get us the same results. But on a big-endian platform the ordering would differ massively due to swapped bytes. On other platforms we might meet some different representation, say signed 0 that compares equal natively but not so in **memcmp**. Or having some unused bits in the memory pattern.

So the best advice is to just not use **memcmp** for this purpose in the first place. And in the rare case it fits, make sure to add a big deal of **static_assert**s, unit tests and documentation that guards all those fragile assumptions. Oh yeah, and if we do use it, make sure to **#include <string.h>** where it lives according to the standard for some time – MSDN still lists that odd **<memory.h>** but it's better to keep the code portable.

What to do instead? Well, unfortunately the code that compares elements is not elegant, especially as all the proposals to add a 3-way compare operator to the language got nowhere. There's the lazy man's approach that uses **std::pair**, chain of those or **std::tuple**, sacrificing the descriptive names to have access to the stock, well-implemented operators. A less intrusive alternative is to assemble the pair/tuple only in the compare function, which is subject to runtime cost. However we have a new, promising facility that uses **std::tie**:

```
#include <tuple>
struct Key
{
  int i;    // put int to front for best
            // potential layout
  char ch;
  char ch2;
  bool operator <(Key const& rhs) const
  {
    return std::tie(ch, i, ch2)
         < std::tie(rhs.ch, rhs.i, rhs.ch2);
  }
};
```

This reads fine, we only need to make sure to match the order on the two sides, and hopefully the optimizer will figure out the same code as our manual chain of comparing the elements.

Fixing this and using **os** instead of **std::cout** in the **<< operator** should fix the behavior and we can look at the minor issues.

As baseline I'd give this structure a constructor like:

```
explicit Key(char ch=0, int i=0, char ch2=0) :
  ch(ch), i(i), ch2(ch2) {}
```

or this without the defaults and another without parameters. (I could accept leaving the state uninitialized if there were some extremely good reason; however I fail to recall an example from the last decade... :) Certainly using healthy names for parameters that I'm sure were just anonymised as part of the simplification process. It's a little less fragile compared to raw aggregates. Unfortunately having three integral arguments it's still too easy to pass them in the wrong order even to a **ctor**.

If we decide to work with aggregates, we had better tighten the reviews and make initialization mandatory. The form in the second **add** function has no good reason, and should just be

```
add(key, Key{ch, i, ch2});
```

or with old compilers we would use the same form as inside **main()**. Certainly the whole function is better scrapped, most likely it is there, along with all those by-value arguments to create more chance to have different noise in the padding and produce the misbehavior.

I put the op< in the **`struct`** as member rather than hanging it out of the class. The difference is not big (especially if we don't mess up the params), but if the compare is the natural 'one way' for this **`struct`** it looks better in the class. While if it was outside because one module used it with an arbitrary content, that is the wrong approach. In that case it should not be the op<, rather a function that has the word Less somewhere and tells us the specifics. And that shall be used with the map definition (an unfair hassle), to avoid confusion and possible ODR violation when another guy creates the operator with the other arbitrary content.

We have a comment around key that it's supposed POD, good to know that adding nonvirtual functions, including in-class op< will not change that – unlike adding a ctor. Much of the reasoning to stick with PODs is worth a review – I have seen too many times it was not really a requirement. But it might be, say because the **`struct`** is used in a public C interface. (Well, those cases in my practice were even more enforced with packing and other instrumentation that made sure the layout is the one as we think...). It's still no reason to drop handy C++ features, we can use that bare POD struct as base class (or member) of a proper C++ one with ctors, operators and whatever other utility. And happy to provide the raw data to the picky clients.

I must notice that this time we have a sample with perfect whitespace format, probably supported from a tool. I normally prefer keeping the **`&`** with the type rather than spaced on both sides, but that is up to local taste.

Commentary

This problem obviously piqued people's interest and many of the issues with the code were well covered by the critiques offered.

However, the reason I originally picked the code wasn't really highlighted; which was that the program demonstrated the dreaded 'undefined behaviour' as it makes control flow decisions based on the uninitialised padding bytes. There's also no guarantee that reading the uninitialised data will give the same result each time. Where optimisers come across uninitialised values they are at liberty to assume the value is whatever makes their job easiest: hence when adding an uninitialised value it may be assumed to be zero and when multiplying assumed to be one.

Also note that since compilers know that the padding is not required it is not necessarily copied when the object is – I believe clang does this for the trailing alignment padding, whereas at least one other deliberately doesn't do this as it causes too many surprises for users.

Tools that perform static or dynamic analysis of the program and notify of uninitialised memory access are extremely useful when finding this sort of problem: only one of the critiques mentioned this.

Finally you may notice that although many of the entrants implemented an **`operator<`** there were several different forms for this – which makes it harder to check quickly for correctness. It may well be that the form using **`std::tie`** will eventually become the de facto standard once C++11 adoption is more widespread.

The Winner of CC 84

This critique had many good entries – thank you to all who took time to put metaphorical pen to paper. I have decided to award the prize to Björn – his clear explanation, with line drawings, of the reason for the trailing padding helped me make my decision.

Code Critique 85

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

> I'm trying to write a function to read an integer from a string but I always seem to get zero. Can you advise me?
>
> The code is in Listing 2.

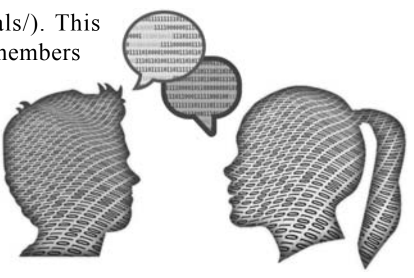
You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
#include <algorithm>
#include <iostream>
#include <string>

// Parse integer with optional commas
int readInt(std::string s, int v = 0)
{
  if (s.empty()) return v;
  if (s[0] == ',')
    std::remove(s.begin(), s.end(), ',');
  int digit = s[0] - '\0';
  if (digit < 0 || digit > 9) return v;
  return readInt(s.substr(1), v * 10 + digit);
}

int main(int argc, char **argv)
{
  for ( int i = 1; i != argc; ++i)
  {
    int const v = readInt(argv[i]);
    std::cout << argv[i] << ':' << v << '\n';
  }
}
```



Bookcase

The latest roundup of book reviews.

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

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

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.
Jez Higgins (jez@jezuk.co.uk)

Android

Android App Development

By Wallace Jackson, published by Apress

Reviewed by Paul Johnson

Not recommended

I have quite a number of issues with this rather weighty tome. It's not that it's badly written; it isn't.

It's not that it's too short; at 507 pages that accusation can't be levelled against it. The problem is that it doesn't actually do what it says it does.

The pace is slow and there is remarkably very little in the way of code in there. Sure it goes into XML and spends chapter 1 in setting up the IDE (which you know is a personal bug bear of mine), but it then starts spending a stack load of time on using GIMP, Audacity and a pile of other applications that while they are good for chopping audio and editing images, really hasn't anything to do with app development.

If you chop out the sections that are peripheral to development or really don't have anything to do with development, you end up with a fairly good book that gives a reasonable coverage to app development – nothing too in-depth, but not too scant.

There are *much* better books out there for app development. This one is a set aside and leave.

50 Android Hacks

By Carlos Sessa, published by Manning, ISBN 978-1-617290-56-5

Reviewed by Paul Johnson

Highly recommended

I've been developing for Android now for almost 2 years, always in C# so I was half expecting this to be another blah-blah-java-blah-eclipse-blah book. It isn't. It's actually full of useful information on how to get the best out of Android with the right amount of technical detail and well documented examples.

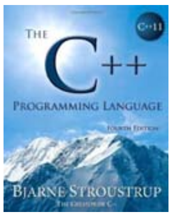
Each example has the version of Android it works from and each example line is commented with additional bullet points for each line, so even if you're like me and only use the Xamarin offering with .NET, the code is accessible and structured.

The author also has a sense of humour which is essential in a book of this type which could have quickly become a dry tome.

If I had a spare couple of hours, I'd port the code over to C# for a wider audience, but when your schedule is a hectic one at best, I'll just say that this is a cracker and deserves the shelf space.

The C++ Programming Language

The C++ Programming Language (4th Edition)



By Bjarne Stroustrup. published by Addison-Wesley, ISBN 978-0-321-56384-2

Reviewed by Alan Lenton

This is not a book for novice programmers. It's also not a book about the differences between C++98 and C++11. Neither is it a traditional style tutorial or just reference book, though it has an index good enough to make it usable as such.

So what is it then?

Its avowed purpose is to provide intermediate and advanced C++ programmers with a thorough grounding in modern C++ defined as being post 2011 ISO standard. The book makes few concessions to how things were done in C++98, its purpose is to show you how they should be done in C++11.

The book is divided into four main parts – 'A Tour of C++', 'Basic Facilities', 'Abstraction Mechanisms', and 'The Standard Library'. I'll look at each of them in turn.

The first section is, at first sight, a bit odd. It's a 100-page rapid look at how things fit together in C++ without going into too much detail at any point. I wasn't sure at first, but after a while I realized that I could start to see how the new facilities would be used, even though the setting was relatively simple.

You can do this sort of thing when you write for developers who already use the language, because you don't have to worry about using common facilities that haven't yet been formally introduced. Some people may not like it, but if it's not your cup of tea it can be skipped without causing too many problems later on.

In the second part we start to cover the basics in more detail. I found the section on references particularly useful, covering, as it does, both lvalue and rvalue references. As readers probably know rvalue references were introduced in the latest standard, but their treatment in this book is typical of the treatment all the way through – as part of a whole, not something bolted on afterwards.

One thing this section has that I haven't seen in most books is a chapter on source files and programs which covers not only linkage, but headers, ODR, and initialization.

The third part covers abstraction mechanisms – broadly speaking classes, templates, generic programming and metaprogramming. Much of the material in this section is hard work. That's not the fault of the author. He is dealing with complex, abstract, concepts which require concentration to understand. You can't simplify them, or you lose the essence of the ideas. Be prepared to give the material your undivided attention, or you will get lost.

The fourth and final part of the book covers the Standard Library. It's only about 400 pages long (though I have whole books shorter than that!) but it's packed with useful material ranging over the whole library. The problem is that the library is big, and this is perhaps the one place where you will find it necessary to have some more specialist books on your shelf in addition to this one.

It's not that there is anything wrong with the section. Quite to the contrary, there is much in it that is excellent, but it just doesn't have the space to cover everything with enough examples. The most obvious need is in the concurrency chapters. The library concurrency material is all there, but there simply isn't space to deal in depth with how to use it safely. I think that the part of my programming shelf dealing specifically with C++ will not only have this book on it but also *The C++ Standard Library* by Nico Josuttis and *C++ Concurrency in Action* by Anthony Williams.

Overall there are a couple of things which I particularly liked. One is the 'Advice' sections at the end of each chapter, one or two liners which make some suggestions about the best way to go about doing the things covered in the chapter. They aren't proscriptive but they represent good advice to bear in mind.

Second, I, for one, found particularly useful the brief examples given in the book. The way they are constructed makes no concessions to pre-C++11 code, and shows how one of the minds behind the standard intended the new material to be used. I'm sure that some of those who follow the work of the standards bodies closely will recognize echoes of arguments in some of the book's explanations of various features!

I got a lot out of this book. More than I expected, and I suspect I'm a better programmer for that. I would be careful who I recommend it to, because, as I said at the start of this review, it's not for beginners.

Coda: This book is physically *heavy*. It's 1,300+ pages, including the index (which as I said earlier, is good enough to make it useful as a reference). I have the paperback edition, I imagine the hardback is even heavier. There have been reviews suggesting that the book is not well constructed. I carried it back and forth to work on the tube (subway) and train for a month, and it's still fine, a little battered, perhaps, but certainly not coming apart. I think that any early problems there may have been must have been fixed.

If you are considering purchasing the Kindle edition you should be aware that there are both tables and diagrams in the book, something I've found are often not handled all that well in electronic readers.

Reviewed by Paul Floyd

Though the 2nd edition came out about the time that I started writing C++ code, for some reason I never read it, and I started off with 'D&E' followed by the 3rd edition of this book. And frankly, I did not enjoy the 3rd edition, to the extent that I relegated it to a far shelf at home.

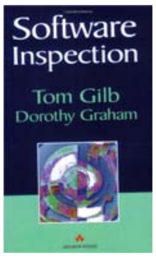
The 4th edition is much improved. I feel that the author has benefited greatly from his other writing and teaching experiences. This makes the book flow better, from the code to the descriptions and the associated advice.

It almost goes without saying that the coverage of C++ is compendious, but where I felt this book is really strong is not just the description of the nuts and bolts but the explanations of the design choices and how various C++ features work together. It's a big book, yet rather dense. I suspect that I'll have to read it again before long in order to digest it well.

I have two slight criticisms. Firstly, I thought that there was a bit too much about concepts, clearly something close to the author's heart. Secondly, there is a bit of a negative tone on the coverage of threads (and a preference for processes, reflecting the author's experience perhaps?).

Review and Testing

Software Inspection



By Tom Gilb and Dorothy Graham, published by Addison Wesley (1993), ISBN: 0-201-63181-4

Reviewed by Paul Floyd

This is one of two books that I read recently on software reviews. I bought it partly because in my current job we do 'code reviews' and I wanted to get some ideas for participating in and running such reviews. None of my previous employers had ever done any sort of organized reviews. The other factor in my purchasing this book was my generally positive experience of reading two other books by Tom Gilb (namely *Competitive Engineering* and *Principles of Software Engineering Management*).

This book is fairly high up on the prescriptive scale. It describes Software Inspection (note the capital letters) as invented by Michael Fagan at IBM in the 70s. Other software inspection methods (no capitals) are described, but mostly in terms of how less efficient they are than the real thing.

The parts of the book are introduction (chapters 1–3), the grist of Software Inspection (chapters 4–7), more details on running the process and solving problems in the process (chapters 8–12). The next two parts of the book cover case studies (chapters 12–17) and the appendices, five of them, mostly templates for plans and reports to use in the inspection process.

Well, I wasn't entirely convinced. Though there are a few success stories, I imagine that it is difficult to get buy-in for such a heavy method from management and engineering.

Peer Reviews in Software

By Karl E. Wegers, published by Addison Wesley (2001), ISBN: 0-201-73485-0

Reviewed by Paul Floyd

I bought this for much the same reasons as *Software Inspection*. I also quite liked two books by the author (namely *Software Requirements* and *More About Software Requirements*). Of the two books this is the slighter tome. Due to its more recent publishing date, it was possible to include web links to 'extras' like report templates.

Inspection isn't the only method covered, covering the gamut from formal inspection down to desk checking and ad-hoc checks. Having said that, inspection does get the lion's share of the coverage. The tone of the writing is also far more relaxed, being less prescriptive and more informative. I felt that my thinking was very much on the same wavelength, in particular chapter 1 (quality motivation for using reviews) and chapter 9 (measuring the results, and in particular the little bit on measurement dysfunction illustrated with a Dilbert cartoon).

One thing that was mildly annoying was the chapter introductions that use presumably fictional little scenes to illustrate the point of the coming chapter. That might just be me, as I've never liked this sort of artifice in non-fiction.

How Google Tests Software

By James Whittaker, Jason Arbon, Jeff Carollo, published by Addison Wesley (2012), ISBN: 0-321-80302-7

Reviewed by Paul Floyd

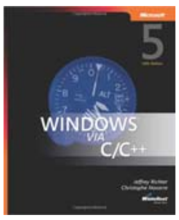
There are some technical books that read a bit like a novel. Then there are some that read like an encyclopaedia. And again there are some that read like a collection of short stories. This book is a collection of bits and bobs that clearly falls into the last category. There's a lot about Google and the Google culture – in part the book reads like an advert for Google hiring and required reading for newly inducted Google testers.

The book gives an overview of the people (SETs, basically a developer/tester role and TEs, test engineers) that are behind the testing. Tools are covered, without going into too much detail. I must say that I envy the Google CI system. Selenium/WebDriver gets quite a few mentions, which I suppose makes sense as it is close to their core business.

Whilst I wasn't expecting the book to literally fit its title and explain the Google systems in detail, I do think that a bit more structure would have helped. There are bits that feel like someone took their Dictaphone to the coffee machine and left it on while a gang of people were chatting casually. Then again, perhaps that's how things are inside Google.

Miscellaneous

Windows via C/C++



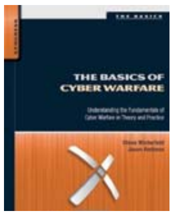
By Jeffrey Richter and Christophe Nasarre, published by Microsoft Press, ISBN: 978-0-7356-6377-0

Reviewed by Paul Floyd

It may be C and C++, but it has a strange Windows API dialect from my UNIX developer perspective, with lots of VeryLongCamelCaseFunctionNames with even longer argument lists (often with MultipleVersionsEx). Not to mention the odd relic from 61 bit Windows like DWORDs. The good news is that this book gives concise coverage of the Windows API in a native manner. The concise aspect is not to be undervalued. The book isn't small at about 770 pages, but I dread to think how it would have turned out had the authors aimed for comprehensiveness.

The parts that I found most interesting were the sections on I/O completion ports, thread pools ans structured exception handling. I also appreciated the tips and pointers to tools like DumpBin.exe and Rebase.exe.

The Basics of Cyber Warfare



By Steve Winterfeld and Jason Andress, published by Synrgress ISBN 978-0-12-404737-2

Reviewed by Alan Lenton

Cyber Warfare is one of the two really hot topics in the US Military-Industrial establishment (the other is drone aircraft, in case you are wondering), and this book is for those who wish to get in on the ground floor. With the reduction in budgets (actually, budget increases), all the armed forces are casting around for new justifications for larger shares of the pie, and all have set up their own 'cyber-commands'. This book is firmly rooted in that milieu.

Needless to say, you won't find a reasoned analysis of the subject, or even a justification for it, in this book. The section headed 'Cyber War – Hype or Reality' occupies less than one page in a 150 page tract. I'm sure I don't have to tell you what its conclusion was! What this book does do, and does very well, is to provide the senior management of companies wishing to become part of this highly lucrative business with the jargon and enough of a basic understanding to not make fools of themselves.

Along the way it provides the largest selection of military bureaucratic acronyms I've ever come across – in just one page it introduces the reader to TTPs, InfoSec, Net Centric Warfare, IA, CNO, CNE, CNA, CND (Computer Network Defense – not Campaign for Nuclear Disarmament!), and IO... And that's just the start. I read the book with a kind of warped fascination. This stuff would make a great basis for a game about cyber-warfare, but would provide little of use for most people in IT, or even IT security.

Oh, and one comment for the publisher, the days when it was acceptable to use bad photocopies of leaflets (in this case an old Verisign leaflet) as an aid to understanding are long since past!

Computer Systems

By Randall E. Bryant, David R. O'Halloran, published by Pearson (2011), ISBN: 0-13-713336-7

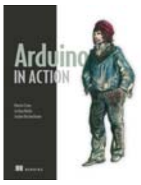
Reviewed by Paul Floyd

There aren't many books like this about, and their scarcity increases their value. Clearly this is aimed at the undergraduate, and I felt that I missed out a lot by not doing the exercises (my excuse is that I do most of my reading over breakfast or on public transport).

The authors try to cover computer systems from the ground up. After the basics, they go into lengthy coverage of machine code. Next up is a cover the basics of processor design, developing a simplified x66 basic processor (which they call y86). This is then enhanced by adding pipelining, out-of-order execution and an explanation of hazards. The next chapter is on performance from the perspective of the compiler and the code that it has to deal with. The last chapter in the first section covers caches and the memory hierarchy.

Section 2 kicks off with a chapter on linking, a topic that generally gets little coverage. After this, as the topics get to higher levels of computer systems, I felt that the ground was better trodden and the scarcity value decreased. The last few chapters cover Unix systems programming in the same vein as W. Richard Stevens. Some of the low level C stuff was lacking government health warnings – setjmp/longjmp are described without saying anything about C++ exceptions and threads. I was a bit disappointed that the chapter on threading didn't get down to instruction reordering and its impact on thread safety.

Arduino In Action



By Martin Evans, Joshua Noble, Jordan Hochenbaum, published by Manning (2013), ISBN: 9781617290244

Reviewed by Andrew Marlow

Recommended but with reservations.

The Arduino is a credit card sized single-board microcontroller for use in various home electronics projects. The Arduino has an development IDE and several open source libraries to help interface the Arduino to other devices.

This book is a practical guide to using and controlling the Arduino and connected devices via its programming language and by all sorts of electronic components, typically connected using breadboard. The components include LEDs, LCDs, speakers, sensors and various kinds of motor.

The book says it is aimed at beginners, but in my opinion the book is much more suited to a more experienced developer and someone who is quite familiar with electronics. The projects start simple but after the first third of the book the pace gathers exponentially with briefer and briefer explanations with more and more material simply glossed over. It makes an excellent start as a beginners book but the programming language is not covered or introduced properly (it is covered in an appendix) so its use becomes more mysterious as the projects become more complex. The use of electronics terminology is also a bit loose and has a tendency to slip into jargon (trimpot, pull-down resistor etc). The more complex projects start to bandy terms about with no explanation at all. For example, the face tracker project mentions a Haar Cascade without explaining what it is or how it is relevant. The beginners part is excellent which is the main reason I recommend the book despite the problems.

The book structure is not clear from the table of contents. There is an introduction and brief history of the Arduino, some simple projects to control things like LEDs and speakers, then projects that are more complex. There is section on wearables, creating your own shield (an interfacing circuit board), and integration with other environments and languages for more even complex projects. These include face tracker, a graphic equalizer and temperature monitor.

There is a large amount of material to cover. The organisation and presentation of the book as a whole does not help with this. The divisions appear to be quite arbitrary and the section and chapter titles are vague. For example, the book is supposedly in two sections, getting started and putting the Arduino to work. This is not helpful. Divisions based on project complexity and use of additional circuitry and devices would have been more useful.

Arduino programs, known as sketches, start off quite useful, but become less so as the book progresses. This is for several reasons: the language is not introduced properly and starts off looking like C, but C++ aspects such as classes are used later on with no introduction or explanation. The code has various clashing styles that overall give the impression of multiple authors with different backgrounds, experiences and coding conventions.

The balance between prose and code fragments is about right but as the code becomes more involved the explanations become shorter. Some aspects, such as why certain baud rates are chosen for the serial line, are barely mentioned at all. Also the circuit diagrams vary a little in style and layout conventions. The diagrams in the wearables chapter are noticably different in style. The shield chapter in particular has some departures from the conventions of previous chapters. The diagrams show the shield as a picture rather than a rectangle, grounded resistors are shown using the ground symbol rather than being connected to the GND line and there are no photographs to illustrate how the circuit diagrams work out in practise (there are helpful photographs in most of the rest of the book). Some of the circuit diagrams are made using fritzing (the diagrams have a small note to this effect) but fritzing is not mentioned in the text (There is a one line mention on the inside cover).

Shields are mentioned several times in the early parts of the book without saying what they are. There is a dedicated chapter on shields that even coveres how to make your own. Surely the first reference could have explained what they are with a forward reference to the detailed chapter.

Later chapters cover external interfaces such as the ethernet port, USB port, using other circuitry such as shields, breakout boards and special leads to use WiFi, Bluetooth and connect to Nintendo Wii, the XBox and Apple phones. There is nothing on how to interface the Arduino to an Android phone, which is a bit suprising.

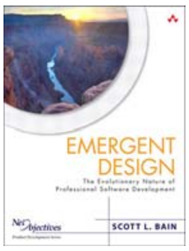
The book is dated 2013 but unfortunately does not mention the more recent Arduino boards. The Leonardo looks to take over the position of the Uno but the book uses the Uno most of the time. The graphic equalizer project uses the java audio library Tritonus. This is a suprising choice since it is linux only (not mentioned in the book) and the last news item on the website is dated 2003 (i.e. it looks to have been neglected for ten years).

The synthesiser project towards the end of the book uses PureData (Pd) which is a visual programming language. The visual programming aspect of Pd does not get the proper mention that it should. There is even a text listing of the generated code, which surely defeats the point. This is compounded by the fact that the Pd diagrams are small and of low quality.

The book covers a huge range of electronic projects and does start with some very simple ones. This makes the book appealing, despite the problems referred to. There are lots of photographs and many of these show assembled breadboards corresponding to the circuit diagrams. This will be of particular help to the beginner. The experienced programmer and electronics hobbyist will probably find the later projects very interesting and fun to do.

My overall impression of the book is that is started with a good emphasis for the beginner but lost this as time went on during the writing, during which many aspects got insufficient treatment, creating a rushed feel, compounded by the clashing styles and conventions. Still, the breadth is impressive and serves as a useful starting point for the more advanced projects.

Emergent Design



By Scott L. Bain, published by Addison Wesley. ISBN: 978-0-321-88906-5

Reviewed by Gail Ollis

I really struggled to see what this book is for. When in a charitable mood I think that it might have been intended as an introductory text to give beginners an introduction to all the practices they should be aware of as a professional software developer – a broad overview to get them asking the right questions and to prepare the ground for reading some important books later. This is not what the book claims to be, though it could indeed be a less daunting prospect than a bookshelf-crushing collection of essential reading presented all at once. On less charitable days, I think the author must have chosen to publish this as a book because you can't sell a one-page bibliography.

The rather harsh judgement stems from my question about the purpose of the book. This volume of 400 or so pages is not (obviously!) a comprehensive guide to patterns, testing, refactoring, coding style and object oriented design. Instead it takes samples of good advice from a range of classic works on these topics. These are perhaps a good start in explaining the importance and usefulness of knowing and understanding concepts that the profession finds useful, but it's not enough to go very far with applying them. A reader not already familiar with the referenced works will need to read those too, so why not give them just enough to whet their appetite for that? This book is a lot longer than it needs to be and would be better as a thin volume of *Practices every programmer should know* – giving the flavour of these, illustrating why they are important and helping readers to identify which ideas warrant the most immediate attention in their own circumstances.

A reader who is already familiar with the cited texts will not learn anything new from this book. The author seems to lack of a clear understanding of just who his audience is. Sometimes he refers to books and their authors as though they are already known to the reader and only later, if at all, pauses to introduce them. The referenced titles and names are in fact almost certainly already well-known to ACCU members. The back-cover claim that the book 'provides developers, project leads and testers powerful *new* ways to collaborate, achieve immediate goals, and build systems that improve in quality with each iteration' (my emphasis) is not accurate. They are new only in the sense that they are not universally known and adopted ways. That they are not universal is evident from the author's banging of a professionalism drum, which gets increasingly irritating with every repetition. There seems little point in labouring the value of adopting professional practices to a reader who has already shown some regard for their own development by picking up a book they hope to learn from.

Clarity of purpose is lacking not just in writing for a particular audience but also in expounding the theme set out in the book's title. The term 'Emergent Design' is used in two different contexts: how software development processes have changed over time, and how a software system can be designed to evolve smoothly over time rather than inevitably decay as changes come along. The former does not really warrant space in this book; how we came to reach our current understanding of what constitutes good software development practice is of little use, even for those with experience of how things once were, in a book which will be read to learn how to do things better within a very different paradigm. The latter theme is more apposite; the author should have stuck to this and kept the book more focused on the value of important current practices. With a couple of small exceptions the book's advice on these is not poor advice – but it is a rather random rehashing of subset of ideas from a good, solid collection of other, well-respected works. My recommendation is to cut to the chase and read the books in the bibliography instead.

View from the Chair

Alan Griffiths
chair@accu.org

We're approaching the deadline for nominations for committee posts. At the time of writing there are a number of posts without candidates standing. In particular both the current Chair and Secretary are standing down and there have been no candidates proposed.

Under the constitution, nominations of candidates for these posts are needed by the Secretary before the 11th February. Naturally, candidates can be nominated for other positions too – even those for which there is a sitting candidate.

The committee is in the process of putting descriptions of the current posts and roles onto the website – by the time you read this they should be available in the members' area.

If no-one stands for election to either the Chair or Secretary posts before the 11th February then the incoming committee will have an interesting situation to deal with. I don't know what they will do but the constitution does allow them to appoint someone to fill roles.

The duties of The Chair should not terrify anyone. They fall into two categories: presiding over meetings and representing either the committee or the organization. Neither is particularly onerous or time consuming.

There are two types of meetings the chair presides at: committee meetings and general meetings. There are perhaps a half dozen committee meetings during the year and one Annual General Meeting. The people involved in these sometimes need prompting to move on to the next agenda item or a vote, but proceedings are generally relaxed and not difficult to run.

This bi-monthly *C Vu* report 'From The Chair' is probably the most persistent task – as every couple of months it is necessary to submit a few hundred words to the *C Vu* editor. The occasional emails from outside the organization that require a response are rare and seldom need much consideration.

I'm not standing down because of the burden of these duties, but because I've achieved what I set out to do when I stood for the post:

- There is more transparency in committee proceedings so members can see who on the committee is (or is not) dealing with the work.
- Committee meetings are now able to accommodate remote attendance both by committee members and by interested ordinary members.
- There is a new constitution that recognizes the global aspirations of the ACCU.
- The website reflects more of the activity happening within the ACCU.

Of course, there is still work to be done. Most importantly, the membership numbers are falling.

I don't have a plan for the future of ACCU so I'm stepping down to make room for someone who does.

Is that you?

Getting Ready for the 2014 AGM

Officers Elections and Motion Proposals

The next AGM will be the first where full remote voting will take effect. The changes from previous years have deep implications for the way officers are elected and motions are proposed.

The deadlines for the next AGM are:

12 January 2014	Announce Deadline
11 February 2014	**Proposal Deadline**
1 March 2014	Draft Agenda Deadline
15 March 2014	Agenda Freeze
22 March 2014	Voting Opens
12 April 2014	**AGM**

Members interested in standing for any committee post must notify the current Secretary in writing (letter or email), including the names of a proposer and a seconder, on or before the **Proposal Deadline**. Please note that the same person cannot stand for more than one role in the same election.

In the same way, members interested in proposing a motion have to do so on or before the **Proposal Deadline**. The only exception is:

> Motions that don't affect the running of the organisation can be accepted from the floor at the presiding member's discretion
>
> (section 7.6.2 of the constitution) .

I encourage you to look at the constitution here http://accu.org/index.php/constitution, especially sections 5 and 7, for more details. You can seek early feedback on ideas for motions by writing to the accu-members@accu.org mailing list (joined from your subscription profile on the website).

On a final note, I want to remind you that Alan Griffiths, the Chair, and I (the Secretary) will not be standing for election again. These are arguably the two most important roles in the committee. The committee needs new people with new ideas. Perhaps you should consider putting your name forward...

For any questions and clarifications, feel free to email me at secretary@accu.org.

Giovanni