The magazine of the ACCU

www.accu.org

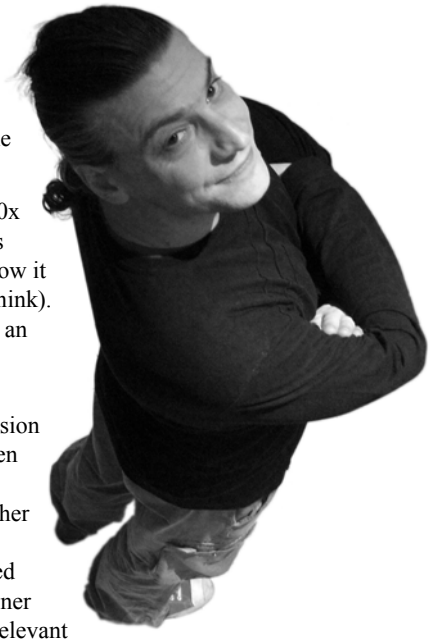
{cvu}

Volume 21 Issue 3 July 2009 £3

Features

{cvu}
Volume 21 Issue 3
July 2009
ISSN 1354-3164
www.accu.org

Shiny New Things



There are some exciting developments in the programming world at the moment. C++0x should perhaps be top of the list, for this publication anyway, but the ACCU is about much more than C and C++ these days. Python 3.0 is still quite a recent release, and newer languages like F#, D and Groovy are gaining popularity, to name but four of the armies of programming languages in use.

It's interesting to note that Bjarne Stroustrup considers C++0x to be a 'new language' to some extent, although its design is deliberately setting out to be compatible with C++ as we know it today (http://www.research.att.com/~bs/C++0xFAQ.html#think). With such an enormous back-catalogue of code, that's quite an undertaking.

It's been almost a decade since C# was first released to the development community, and work is now underway for version 4.0. Some years ago I attended a conference at which the then new .Net and C# were the stars of the stage. I overheard someone talking to Don Box, of COM renown (and many other things, of course, but I hope he won't mind me forever associating his name with 'Essential COM' – which is indeed essential to any developer working with COM). The questioner wondered if .Net had just made all COM and native C++ irrelevant (quite pleased at the prospect I think). Don was definitely sceptical of the idea that all the COM and C++ would vanish overnight.

The interesting thing about so-called 'legacy' languages – and I include C and C++ in that collection, simply because that is a commonly-held perception – is the depth and breadth of experience a large number of people have with using them. They are 'legacy' because they are successful, and being successful they have lessons to teach us, even if we're using shiny and (relatively) new languages and technologies today.

The state-of-the-art in software development is at least in part about applying the lessons already learned in the 'old' to the 'new'.

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 21.4: 1st August 2009
C Vu 21.5: 1st October 2009

Overload 93: 1st September 2009
Overload 94: 1st November 2009

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

Hunting the Snark (Part 3)

Alan Lenton looks at the process from the hot seat.

The interview

The room was square, with no windows, and a whiteboard on one wall. The air conditioner made a distracting thrumming noise, but failed to noticeably cool the atmosphere. In the middle was a table with two men sitting on the far side from the door. Facing them was a single, empty, hard and uncomfortable looking chair. As the candidate entered the room the older of the two men rose and extended his hand. 'I'd like to welcome you here today', he said, 'My name is Grey, and this is my colleague, Mr Brown. We will be assessing your technical skills.' Mr Brown smiled a sardonic smile.

The candidate shook the proffered hand and sat gingerly on the edge of the chair, wondering why it was de rigueur to wear uncomfortable suits at interviews. The interviewer smiled a plastic smile and carefully placed his hands flat on the table. 'Well I think we might as well start', he said, 'my apologies for keeping you waiting.'

There was a moment's silence, but the candidate failed to say that it wasn't a problem. Mr Brown made a note in the 'teamwork' section of his note pad. 'Now', he began, 'for the first part of this practical driving test, I'd like you to explain exactly how you would perform a three point turn. We would like you to be sure to cover all aspects of multi-tasking your eyes, both arms and your feet. If you feel the need to describe a parallelogram of forces acting on the car then please feel free to use the white board. The marker is a bit dried up, but I'm sure you can overcome that small problem.' His companion smiled wolfishly.

The candidate cleared his throat nervously and started, 'Well, first of all I would move my eyes to look in the rear view mirror...'

As the interview proceeded the questions became more difficult. It became clear from his interjections that Mr Brown was a would-be Stirling Moss, probably the test centre's guru. All his little additions to Mr Grey's questions seem designed to show how clever Mr Brown was, and conversely, how little the candidate knew. Every time the candidate stumbled over his description of the action taken, Mr Brown made a note on his pad, most of which was, by this time, covered with psychologically revealing doodles.

'At this stage, I would move my left hand onto the gear lever, and shift it back into reverse by pressing down hard and moving it all the way over to the right before pulling it toward the back. At the same time my right foot...' The candidate was going through a description of parallel parking when Mr Brown said sharply, 'Stop!'. The candidate stopped abruptly, his train of thought interrupted.

Mr Brown leaned on to the table, giving a very good impression of a predator moving in for the kill. 'Your passenger chooses this moment to reach across and press the cigar lighter in, in preparation for lighting up a cigarette. What would you do?' The candidate thought rapidly, recognising this as a variant on the teamwork test question in his 'Driving Test for Dummies' book. Fortunately, he had mugged up on the question the previous night, and was able to give the politically correct answer to the effect that smoking was bad for the health, the dangers of secondary (and, for a bonus point, tertiary) smoking, the cost to the Health Service (figures to an accuracy of 0.1 of a penny), and its effect on average life expectancy. This, noted the candidate, should be explained to the passenger.

The interrogator looked faintly disappointed and leaned back in his chair. As he did so he asked, in a suspiciously casual voice, 'By the way, as you are parking you notice a manhole in the road. Do you happen to know why manhole covers are round?' The candidate blinked. 'Where I come from they're square.'

Figure 1

Mr Brown looked fazed for the first time in the interview, and the candidate mentally chalked himself up a much needed point. Mr Grey smiled faintly at his colleague's discomfort and indicated that the candidate should continue with his description of parallel parking.

The interview wound on, seeming interminable. Eventually, Mr Grey said, 'Well that seems to cover most of the issues, including exceptions, which my colleague added into a previous question by having a child run out in front of you. We only have one item left on the list. As you have probably realized, most of the questions so far can be considered to be high level driving problems, so we would now like to get an idea of whether you understand what is happening when you 'hit the metal' at we say in the trade.' He gestures to Mr Brown, whose resulting smile would have done Torquemada proud (Figure 1).

the candidate mentally chalked himself up a much needed point

With a flourish Mr Brown produces a featureless cylinder of greyish-silvery metal, about a foot long, and perhaps three inches in diameter. He places it upright in the exact middle of the table. There is a pause. A drop of sweat trickles down the candidate's face, but to brush it away would indicate nervousness and a sure fail. Clearly enjoying himself, Mr Brown allows the silence to stretch on for a few more moments before clearing his throat noisily.

'You see before you', Mr Brown clearly has a taste for histrionics, 'a cylinder of metal, made from an alloy of...' he drones on for several minutes about the physical and chemical properties of the metal. He stops suddenly , wincing with an expression of pain – obviously Mr Grey has kicked him under the table. 'But I digress', he continues, somewhat superfluously. 'What I want is for you to explain how you would machine this into a piston for a 1600cc family saloon. You have access to all the modern metal working machines – lathes, millers, shapers, annealing facilities, etc. However, none of them are numerically controlled, so you will have to do it manually.' As an afterthought he adds, 'We call this the assembler test...'

Well, there you have it, as promised, but in the form of a little parable. I'm sure I don't really need to belabour the point.

That's all I'm doing on interviews for the time being, but in the event of there being vociferous demands from the readership of this magnificent magazine (actually a vague indication of interest from a single reader will suffice, I'm not proud), then I will return to producing a regular column. ■

ALAN LENTON

Alan is a programmer, a sociologist, a games designer, a wargamer, writer of a weekly tech news and analysis column, and an ocassional writer of short stories (see http://www.ibgames.com/alan/crystalfalls/index.html if you like horror). None of these skills seem to be appreciated by putative employers...

Improve Code by Removing It

Pete Goodliffe makes code better by making it smaller.

Less is more. It's quite a trite little maxim, but sometimes it really is true.

One of the most exciting improvements I've made to a codebase over the last few weeks is to *remove* vast chunks of it. Let me tell you, it's a good feeling.

Code indulgence

We were working as a small agile software development team. We'd been constructing a large software project following the hallowed Extreme Programming tenets, including YAGNI. That is, *You Aren't Gonna Need It* – don't write code that you don't need. Even if it is going to be needed in future versions, wait for that future version to add it. Don't do it now if you don't need it yet.

dead pieces of code will still spring up naturally during your software development

It sounds like eminently sensible advice. And we'd all bought in to it.

But human nature being what it is, we inevitably fell short in a few places. At one point, I observed that the product was taking too long to execute certain tasks – simple tasks that should have been near instantaneous. This was because they had been over-implemented; festooned with extra bells and whistles that were not required, but at the time had seemed like a good idea.

So I simplified the code, improved the product performance, and reduced the level of global code entropy simply by removing the offending features from the codebase. Helpfully, my unit tests told me that I hadn't broken anything else during the operation.

A simple and thoroughly satisfying experience.

So why did the unnecessary code end up there in the first place? Why did one programmer feel the need to write extra code, and how did it get past review or the pairing process? Almost certainly the programmers' indulging their own personal vices. Something like:

- It was a fun bit of extra stuff, and the programmer wanted to write it. *(Hint: Write code because it adds value, not because it amuses you.)*
- Someone thought that it might be needed in the future, so felt it was best to code it now. *(Hint: That isn't YAGNI. If you don't need it right now, don't write it right now.)*
- It didn't appear to be that big an 'extra', so it was easier to implement it rather than go back to the customer to see whether it was really required. *(Hint: It always takes longer to write and to maintain extra code. And the customer is actually quite approachable. A small extra bit of code snowballs over time to a large piece of work that needs maintenance.)*
- The programmer invented extra requirements that were not documented in the story that justified the extra feature. The requirement was actually bogus. *(Hint: Programmers do not set system requirements; the customer does.)*

Now, we had a well-understood lean development process, very good developers, and procedural checks in place to avoid this kind of thing. And unnecessary extra code still snuck in. That's quite a surprise, isn't it?

It's not bad, it's inevitable

Even if you can avoid adding unnecessary new features, dead pieces of code will still spring up naturally during your software development. Don't be embarrassed about it! They come from a number of unavoidable accidental sources, including:

- Features are removed from an application's user interface, but the back-end support code is left in. It's never called again. Instant code necrosis. Often it's not removed 'because we might need it in the future, and it's not going to hurt anyone left there...'
- Data types or classes that are no longer being used are rarely removed. It's not easy to tell that you're removing the last reference to a type when working in a separate part of the project.
- Legacy product features are never removed. The users no longer want them and will never use them again, but it's just not the done thing to reduce an application's functionality! We incur perpetual product testing overhead for features that will never be used again.
- The maintenance of code over its lifetime causes sections of a function to not be executable. Loops may never iterate because code added above them negates an invariant, or conditional code blocks are never entered. The older a codebase gets, the more of this we see.
- Wizard-generated UI code provides hooks that are frequently never used. If a developer accidentally double-clicks on a control, the wizard adds backend code, but the programmer never goes anywhere near the implementation. It's more work to remove these kinds of auto-generated code blocks than to simply ignore them and pretend that they don't exist.
- Many function return values are never used. We all know that it's morally reprehensible to ignore a function's error code, and we would never do that, would we? But many functions are written to do something and return a result that someone might find useful. Or might not. It's not an error code, just a small factoid. Why go through extra effort to calculate the return value, and write tests for it, if no one ever uses it?
- Much 'debug' code is necrotic. A lot of support code is not needed once the initial implementation has been completed. It's not unusual to see reams of inactive diagnostic printouts and invariant checks, testing hook points and the like, that will never be used again but clutter up the code and make maintenance harder.

Programmers do not set system requirements; the customer does

So what?

Does this really matter? Surely we should just accept that dead code is inevitable, and not worry about it too much if the project still works. What's the cost of unnecessary code?

- It is a simple fact of life that dead code, like any other code, requires requires maintenance over time, and any unit tests also need to be updated accordingly. It costs time and money.

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

Misconceptions about TDD

Matthew Jones dispels his concerns.

This article started out as a question posed on accu-general [1] about TDD and the intrusion of testability into class design. As I had hoped, the question generated a number of interesting replies which were well worth writing up.

It is a bit of a personal account, but we're not all experts in TDD so I hope that in summarising the discussion, anyone in a similar situation will learn something. It is now obvious that I was suffering from a number of misconceptions which were swiftly corrected by the responses from the list. It is likely that these misconceptions are common.

It is now obvious that I was suffering from a number of misconceptions

The background to the problem is that I am trying to encourage my software team to fully adopt TDD. In the process of writing some guidelines to help them, I was trying to get to grips with TDD, design for testability, and test coverage. We had just held a meeting to discuss this, and I had explained how I thought that TDD would naturally lead to 100% code coverage because no code is written unless a test calls it into existence. We incorrectly inferred that, taken to its logical conclusion, TDD meant all methods had to be public to be tested. This was obviously wrong, so I sought a higher authority.

Here is my original question:

> I'm trying to sort out design for test, and the scope of TDD, with my SWteam:
>
> - TDD implies every method of every class is tested.
> - Without allowing the test harness to intrude (e.g. friend classes etc)
>
> this means all methods must be public so they can be called by the test harness.
>
> - There are certainly cases where (e.g.) an important algorithm is encapsulated (i.e. private), but really ought to be tested.
> - If important methods remain private, you can only test them through other methods, which may prevent passing a full range of parameters etc.
>
> i.e. more of a black box test.
>
> This seems to go against encapsulation, and almost makes protected and private redudandant.
>
> In a recent discussion, I suggested that methods that would be private, but are public for testability, should be grouped under a suitable comment.
>
> It was suggested that we have a `"test_public:"` macro which is public for tests and private for 'deliverable' code, to enforce the approach and stop people taking short cuts and using should-be-private methods. These shortcuts might not happen now, but years down the line when the original team is no longer maintaining the code. It seems reasonable but goes against my intuition.

There are a number of points, so I shall summarise the responses accordingly.

TDD implies every method of every class is tested

This is the faulty premise which led me into drawing the other conclusions. By treating this almost as dogma we led ourselves into a number of cul-de-sacs.

MATTHEW JONES

Matthew started programming with BBC Basic, and then learned C on a summer job between school and VI form. He has been programming professionally for over 15 years, having moved on to C++, and usually works on large embedded systems. He can be contacted at m@badcrumble.net

Improve Code by Removing It (continued)

- Extra code also makes it harder to learn the project, requires extra understanding and navigating.
- Classes with one million methods that may, or may not, be used are impenetrable and only encourage sloppy use rather than careful programming.

Dead code won't kill you, but it will make your life harder than it needs to be

Dead code won't kill you, but it will make your life harder than it needs to be.

How can you find dead code? The best approach is to pay attention whilst working in the codebase. Be responsible for your actions, and ensure that you always clean up after your work. However, code reviews do help to highlight dead code. And if you're serious about rooting out unused code sections, there are some great code coverage tools that will show you exactly where the problems are.

There is no harm in removing dead code. It's not like you're throwing it away. Whenever you realise that you need an old feature again, it can easily be fetched from your source control system.

There is a counter argument to that simple (and true) view, though: How will a new recruit know that the removed code is available in source control if they don't know that it existed in the first place? What's going to stop them writing their own (buggy or incomplete) version instead? This is a valid concern. But similarly, what would stop them re-writing their own version if they simply didn't notice the code fragment was located elsewhere?

Conclusion

Dead code happens in even the best codebases. The larger the project, the more dead code you'll have. It's not a sign of failure. But not doing something about it when you find dead code is a sign of failure. When you discover code that is not being used, or find a code path that cannot be executed, remove that unnecessary code.

When writing a new piece of code, don't creep the specification. Don't add 'minor' features that you think are interesting, but no one has asked for. They'll be easy enough to add later, if they are required. Even if it seems like a good idea. Don't do it. ■

TDD implies that every line of code is tested, not that there is a test for every line of code. This is a very important distinction, and I failed to see it. By concentrating on one outcome of TDD, test coverage, I had forgotten about the bigger picture: that TDD shapes the overall design. High (but not necessarily full) test coverage is a beneficial side effect.

TDD causes the public interface of a class to be brought into existence, but says nothing about the implementation behind it. If the implementation makes use of private methods, the unit tests know nothing of this and therefore they should not try to tunnel through and access private methods directly. Private really is private.

TDD forces you to consider the design of a class as you write it. Because the unit tests are written before the 'rest of the world' exists (the world that the class will fit into later on), it also forces you to provide a framework for testing the class in isolation right now. Typically this will mean isolating the rest of the world behind interface classes.

All methods must be public so they can be unit tested

This is a gross intrusion of testing into the design. As I said in the original question it goes against encapsulation, not to mention intuition. This point is really the (incorrect) logical conclusion drawn from the other three assertions, but was second in the original list, so is second here. The discussion of the other points also address this one.

If we were to make all methods public for testability, then yes we would probably be able to achieve higher test coverage. However, we would have then reversed one of the fundamental principles of TDD: the development is now leading to tests. We have chosen to write a method that would have been private, then made it public, then written tests. Once the tests exist they justify the existence of the method to an outside observer. There is no way to spot that the method might, for example, be redundant. We also lose the knowledge that the method is an outcome of the design, not a requirement, and is therefore of secondary importance compared to the 'true' public interface.

Resorting to conditional compilation would be even uglier. It is widely considered to be *A Bad Thing* to test code that is different to what is delivered. In this case the only difference is in the access specifier of a number of methods, but once conditional compilation is accepted, it can be the start of more serious rot.

Important private methods really ought to be public so they are tested

A number of people pointed out that [2] describes a number of techniques that could be used to solve this problem. One is to make the method protected and virtual, then access it for testing through a derived class. The derived class only exists in the test context.

Another is to separate the code into a new class, with its own unit tests. An instance of this new class can then be passed to the original class and treated as a safe, tested, entity. If this relationship were to be reduced to an interface class we could move to using mock objects for test. If the design evolves, this separation would also allow use of the STRATEGY pattern [3], if the motivation arose (i.e. substituting different algorithms at run time).

In these cases TDD is doing its job more subtly: the testing considerations are driving us to alter the design by breaking up something that is irritating and 'doesn't fit' into something clearer and probably simpler. And it will almost certainly be easier to understand. The requirement for testability leads us to better design.

Testing private methods through the public interface leads to limited back box testing

It should be obvious by now that unit tests might not reach into some dark corners of private methods, and we shouldn't tie ourselves in knots trying to remedy this. If you can't test part of a private method, then it is not being exercised by the public interface, and its very existence is questionable. After all, if you remove it, no tests will fail, and therefore from the TDD point of view it doesn't need to exist in the first place.

100% code coverage is a separate goal from 100% TDD. It usually requires concerted effort to even get close to 100% code coverage. If we think we need it we should first ask ourselves whether it is really worth it. In most cases it is enough to 'just' have a well tested, well designed, and adaptable code base. And that is before even considering the exact meaning of 'code coverage', which is a whole subject unto itself, and outside the scope of this article.

Conclusions

I quickly realised that I had confused TDD, which leads to black box testing through the public interface, with high test coverage which usually requires a white box approach. I had concentrated too much on the 'T' and forgotten about the 'DD'.

The responses went beyond simply answering the original questions and provided good background to the whole subject of TDD. It is interesting that although at no point was 'legacy' mentioned, a number of responses recommended [2]. Another good recommendation was [4]. ■

References

[1] http://accu.org/index.php/mailinglists accu-general, Mon, 9 Mar 2009, thread entitled 'Design for TDD testability'.

[2] *Working Effectively with Legacy Code*, Michael Feathers.

[3] *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson and Vlissides

[4] *Test-Driven Development By Example*, Kent Beck.

Acknowledgements

Thank you to all the participants in the accu-general discussion, for both answering my question quickly and constructively, and providing the rich vein I mined for this article. In particular, I have used comments from Balog Pal, Bill Somerville, David Sykes, Jon Jagger, and Michael Feathers.

Getting REKURSIV

Seb Rose reminisces about object-oriented hardware.

James Clerk Maxwell was one of the 19th century's greatest physicists, developing a theory of electromagnetism that has been called the 'second great unification in physics'. If, at some time during his life, a stray time machine had deposited him in the nether regions of the eponymously named Edinburgh University building, he may well have died of starvation negotiating the maze of twisty little corridors (all alike) [9] in search of food or an exit. It didn't. He survived his time at Edinburgh and eventually died in Cambridge in 1879 of abdominal cancer – which only goes to show he should have remained in Edinburgh.

This tale is not about James Clerk Maxwell. It is not about physics. It is certainly not about the design of buildings, cancer or time travel. It may not even be about Edinburgh University. But it does start in the basement of the JCMB in the distant past of 1987 – when the internet was young, memory was scarce and foolish adventurers still dared to do bold (and foolish) things.

It was a sunny day and I was completing my long forgotten final year dissertation on a qwerty terminal in the basement of the JCMB. As a diversion I decided to check my e-mail, a novelty back then, in the slight hope that there would be something interesting in my inbox. There was. It was a brief message from a recent graduate, Ian Elsley, to everyone in the computer science department looking for someone interested in joining a team in Glasgow as a 'Firmware Engineer'. I replied, out of boredom or desperation, I can no longer remember which, asking for an explanation of what a firmware engineer was. The die was cast.

A couple of weeks later I was taken out for a drink by short, Glaswegian academic by the name of David Harland. He explained that my ignorance of all things firm was not important. Apparently, being a CS graduate from Edinburgh was sufficient for his purposes. He himself had been an astronomer and noticed, while working nights in the St Andrews observatory that the computer programmers were working somewhere a lot warmer. His powers of abstract thought meant that a move to CS or Maths was more or less inevitable, and CS had the edge because the area he became interested in didn't have a lot of literature at the time (and because he didn't understand mathematics). So he published prolifically and designed a language called 'Lingo' which is described as being 'based on the Smalltalk language with Algol-like extensions' [1].

Academics design programming languages. It's what they do. Normally it's a fairly safe exercise, because it never leaves the department and no one wastes any time or money on it. This time however, at least one butterfly must have flapped its wings. Linn, a manufacturer of high-end hi-fi had employed an itinerant Australian researcher by the name of John Wainwright to look into automation of their new warehouse at Eaglesham, near Glasgow. His beady eye fell upon David, who was a lecturer at Glasgow University, and he asked him to join the team. Characteristically, David refused. Equally characteristically, after sufficient badgering, he repented and joined the team as the resident boffin.

At this point I should explain that Linn was (and is) run and owned by Ivor Tiefenbrum, a colourful Glaswegian engineer, entrepreneur and sailor. His unswerving dedication to audiophilic quality has ensured Linn's continued survival, but his idiosyncratic approach to business does not always serve him so well. Linn had outgrown their somewhat dilapidated premises on Drakemire Drive in Castlemilk (one of the less salubrious suburbs of Glasgow) and he had built a modernistic edifice at the much more up market Eaglesham, south of the city. His dream was to have a fully automated warehouse, where computer controlled machines would glide silently to retrieve parts for the production line. A gleaming embodiment of all that Linn stood for.

Also at Linn was Bruno Beloff who had a summer job there while he completed his AI degree at Edinburgh. Robin Popplestone, his professor, had a robotics research contract with Linn (he was designing a six-axis robot wrist they thought they might need for making speakers) and took him along for a visit. Ironically, Ivor decided to dump Robin's project, but hired Bruno to building a controller for their record-cutting lathe. This turn around didn't help Bruno's academic career, but while he was in fourth year, Linn offered him a full-time job. 'I was too lazy to look elsewhere, so I took it' is how Bruno remembers it.

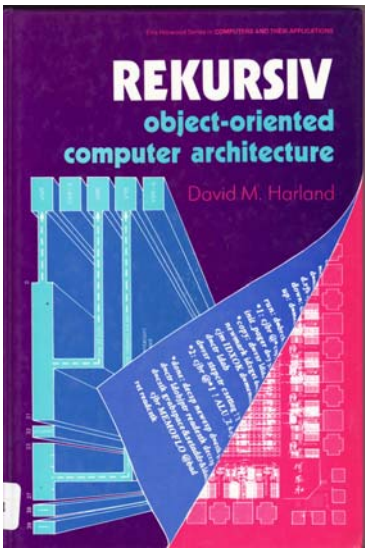


Figure 1

David and Bruno started building a prototype for an object-oriented machine out of standard components. 'One day we booked a flight to Southampton to give a seminar about it, but the plane journey ended in Birmingham (or thereabouts) and we were given train tickets to complete the trip. But the train drove into a yard and remained there for several hours. In whiling away the time, we started to sketch out how the logic could be re-partitioned into a set of custom chips – and on returning to Linn we talked Ivor into financing their design and production. So you see, when all is said and done, the Rekursiv is British Rail's fault!' is David's way of shifting the blame.

Ivor is not one to be daunted by impossible odds, unknowable costs or personal doubt. He set up a separate company, Linn Smart Computing, to develop the technology, installed his brother, Marcus, as the managing director and David as the Technical Director. He invested a large chunk of Linn's money in the venture and somewhere along the way shamed the Department of Trade and Industry into backing it with £10 million of the taxpayers' money [2]. Let the work commence!

The architecture was named REKURSIV. The chips were called:

- **OBJEKT** – for memory management
- **LOGIK** – the micro-sequencer
- **NUMERIK** – the data manipulator

These names are all trade marked by Linn, and some of them have subsequently resurfaced as Linn audio products.

There is a book (Figure 1) describing the architecture [3], which has been described by its author as 'best used as a door stop.' I think the best bit is the hypothetical discussion between a RISC proponent and an EISC (Extended Instruction Set Computer) proponent that includes exclamations such as 'That is pure conjecture!' and 'Rubbish!' Needless to say, the EISC man wins the day and the obvious conclusion is to build

SEB ROSE

Seb is a software contractor in some of the most hostile working environments, including banks, pension providers and manufacturers. He works with whatever technology is prescribed by the client – mainly C++, .NET and Java. Recently he realised a lifelong ambition and became a tractor owner.

an object oriented machine. The rest of the book describes the reasoning behind the project in detail and having perused it again recently I can say that it certainly was an interesting research project. David's view today is that 'trying to make it a commercial product was naive.' [4]

Figure 2

The thing about the **REKURSIV** was that users could program different instruction sets. James Lothian, who was worked at Edinburgh University microcoding the Prolog instruction set describes it as being 'a really interesting and unusual design: the main memory was in effect a persistent object store, with every object having its type, size and position in memory known in hardware, so that (for example) the hardware could prevent you from 'running off' the end of an array and corrupting surrounding memory. Paging of objects into and out of main memory was handled by the host machine (generally a Sun 3), and was completely transparent, even at the microcode level. This meant that you could write arbitrarily complex algorithms in microcode, even recursive ones, hence the machine's name. Every object had a unique identifier (a 40-bit number), and the MMU chip would translate that into the object's store address (if it was in main memory). Since only the MMU knew the object's address, an object could be moved around in memory without having to update references to it (since they were in terms of its object number); this made garbage-collection particularly straightforward.' [5]

Now let's meet the rest of the team:

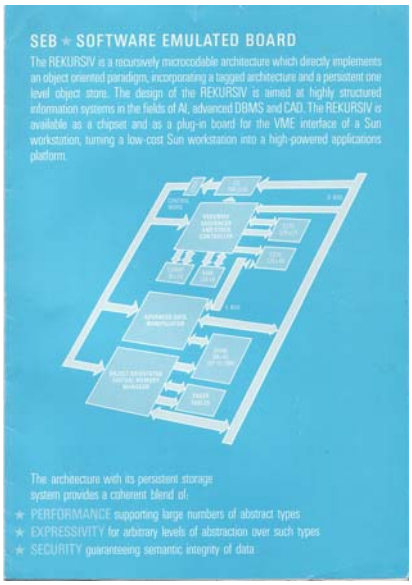
First in was Ian Elsley, the top graduate from Edinburgh University's class of '86. Linn needed a UNIX guy to run the development effort. Bruno knew Ian. Ian 'did' UNIX. Sweet [6]. To convince Ian to join, Bruno 'produced some crazy figures … that showed that in five years, the Rekursiv project would be bigger than DEC.' [10]

Next was Hugh Stabler, a friend of Ian's from (have you guessed yet?) Edinburgh. Hugh was an old school hacker, who had gone to University as a mature student. He could code the socks off anyone I've ever worked with and was a nice guy to boot. He was newly married with a young family, and pretty much just got on with the job (which in his case was knocking up a C compiler for the new chipset).

And then there was me. I soon found out that in this particular organisation a firmware engineer's job was to implement an assembler and emulator for the chipset. The chipset was being delivered on a board for a Sun Workstation, and the resulting emulator, hand crafted in X10, was duly released as SEB: 'Software Emulation of the Board'. (Figure 3) I should point out that apart from some BASIC programs I'd written for estate agents in Twickenham, I'd never written any commercial software. I'd never worked in UNIX. I'd never developed on a windowed platform. I was useless, but no one seemed to notice. And the software seemed to work [5]. After that I went on to micro-code the C instruction set, so I really did become a firmware engineer.

Meanwhile, Bruno was designing the chipset. He'd done a short course at LSI Logic (which lasted a whole day including a long lunch break), and was doing as well as could be expected, but needed some help. I called up an ex flatmate of mine from (can you tell where, yet?) Edinburgh, Duncan McIntyre, who had just graduated as an Electronic Engineer, and along he came. He took one look at Bruno's designs and immediately asked: 'where are the reset lines?' There weren't any. It was 'too late' to change the designs significantly, but they managed to fit a single reset line in to the program counter, enabling the chipset to be bootstrapped just in case we ever got that far.

Bruno and Duncan went on to share a flat together, and spent a fair amount of time at the LSI labs in Livingstone. Bruno remembers 'our extreme bad behaviour at LSI Logic in Livingston, where we poured breakfast cereal into each other's booths, smashed up keyboards, electrical fittings, and Duncan nearly broke the manager's jaw by slamming a door on him.' When David visited, he took great delight in 'locking up' the logic of the manager's mobile phone, which in those days was a thing the size of a brick, but the guy couldn't complain because Linn was paying his commission.



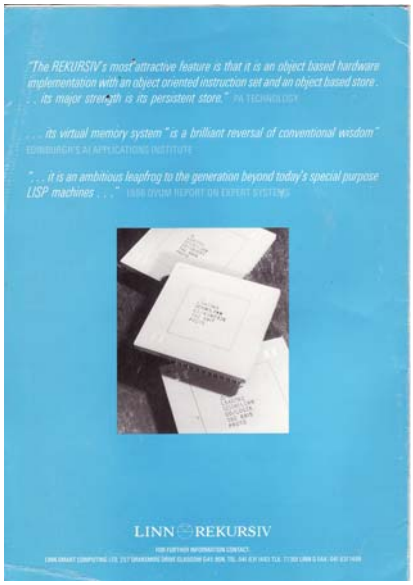


Figure 3

To complete the technical team, a 'real' hardware engineer was hired via an employment agency. Brian Drummond, who was still working at Linn the last time I checked, could have come straight out of Dilbert. He walked out of a meeting once and was later seen in his office pacing up and down, yelling, 'I must learn to communicate better!' out loud to himself. Another time, he got so frustrated he started banging his head on the wall so hard that he dented the plasterboard. Duncan found a large piece of polystyrene packing material, wrote 'BANG HEAD HERE' on it, and stuck it to the wall.

You might have got the impression that the REKURSIV team was geek meets nerd with a touch of dysfunction. You might be right, but like all successful (?) geek/nerd teams we had our mother. Kirstine Lambie was there before I arrived and stayed long after I left. She may still be in Glasgow, for

all I know, and I remember the firm common sense with which she organised the office.

In fact, Kirstine was the only sensible thing that I remember.

There were mad rumours of huge venture capital deals that never materialised (which finally came to an end when Black Monday put Britain's economy into freefall.)

Linn now had a product that they didn't understand and no one seemed to want

There were the mad cars. David had just learnt to drive and bought a Porsche as his first car. Bruno got himself a Lancia Delta Integrale, which he kept driving into the back of people at roundabouts.

And there was paranoia a-plenty. 'There was the time when Bruno's car was broken into and the chip plans stolen. Someone shot the window out with a low velocity gun; looking back it does seem very odd,' says Ian. 'Dave and I ended up meeting Bruno half way between Edinburgh and Glasgow to give him another car and the three of us talked in the back of a land rover with the engine running because someone had the idea that we couldn't be recorded that way.' [10]

A lot of time and money had been invested in the project, and the Linn warehouse still wasn't up and running. Black Monday was having a huge effect and Linn was running short of money, so David and Bruno were dispatched to various places to drum up custom. They weren't particularly successful, but seemed to have a good time. Bruno again 'Dave was the only person in Linn to hire a horse on a company credit card. Good on him. He took it for a weekend tour of Yosemite Park in California when he was supposed to be visiting computer companies in Silicon Valley. He came back, and said he could make it do anything it wanted.'

'Meanwhile I picked up a hitch-hiker in Santa Cruz and ended up, several days later, in a weird situation in the woods with her husband who, I eventually became convinced, was going to kill me. My ensuing departure eventually got me stopped by the California Highway Patrol. Somehow, I convinced them that I was not, in fact, extremely drunk. How, I still don't know.' [2]

'To be fair,' adds David, 'we did see IBM while we were over there – we passed it on the freeway at 55 miles per hour but did not feel the need to stop.'

'It wasn't IBM. It was Hitachi', says Bruno. 'David turned to me, grinned and said, "We saw Hitachi, but they didn't express an interest."'

I stayed at Linn for a whole 8 months, before heading into the wilds of northwest Scotland, and the project progressed happily without me. Around 20 boards were sold, but now no one seems to know where any of them are. Eric Smith thought that he'd found one in a surplus store in Colorado [7], but he later told me in a private e-mail that after further research he'd decided he hadn't.

Linn now had a product that they didn't understand and no one seemed to want. It had been built to compete with VAX's, but was seriously outgunned by Sun SPARC stations and even PCs. Linn had finally overreached its finances and the project was clearly at risk, but the final straw was a Linn delivery driver called Shug. He reversed into David's Porsche and Ivor unwisely decided that, since the incident had happened on private ground, Linn weren't responsible and wouldn't pay for the repairs. David quit and 'chucked all that I had in the way of bits and bobs of hardware into the Forth and Clyde Canal.' [8] I seem to remember hearing that all the backup media went the same way.

Ian still thinks we were onto something: 'The idea behind the Rekursiv was fabulous. Strongly typed hardware with a microcodable virtual machine that could be configured to pretty much any environment simulating any machine. That last part was why Hugh was brought in to show that it could even run a conventional high level language. When strongly typed object oriented languages were running in memory it was astonishingly fast for the time. But people were so locked into 'clock speeds' that they couldn't get their heads round a totally variable clock cycle, set by the microcode.' [9]

In the end, we didn't contribute to the sum of human knowledge in the way that James Clerk Maxwell did. Anything we achieved was, at best, compost in which other endeavours found sustenance. But the whole experience, at least from my point of view, pays tribute to the labyrinthine depths of the JCMB where, if you checked your e-mail at the right time, you might find an emerald the size of a Plover's egg [10].

Where are they now?

- David Harland lives in Glasgow and writes about the space program: http://homepage.ntlworld.com/dave.harland/My_Books/
- Bruno Beloff lives in Brighton and runs ClassCalendar.biz: www.classcalendar.biz
- Ian Elsley lives in Honduras and teaches diving
- Hugh Stabler was last seen in the Forest of Dean and latterly worked for Xerox. Present whereabouts unknown
- Duncan McIntyre lives in Twickenham and is CTO of reviewcentre.com
- Brian Drummond may well still live near Glasgow and work at Linn
- Kirstine Lambie lives in Bordeaux and teaches English
- I live near Edinburgh and work for IBM ■

References

[1] http://web.archive.org/web/20070607082132/http://www.erg.abdn.ac.uk/research/projects/lingo.html

[2] Bruno's version is 'We got into what was effectively the end of the Alvey Programme. I went with David to the office at Millbank Tower in London. The main man was there. He didn't understand anything about what we were trying to do. Then David punted the idea that the Japanese were trying to do something very similar, and from that moment the deal was done.' Personal e-mail 29 May 2009

[3] *REKURSIV – Object Oriented Computer Architecture* – Harland – 0-7458-0396-2

[4] David goes on to add: 'sometimes I feel that I should write a book which really explains the principle of abstraction as it should apply to the design of programming languages, but whenever this feeling arises I lay down until it goes away!' – Personal e-mail 31 March 2009

[5] Much later, after I'd left, they found that it 'wasn't well enough behaved to run with the X10/X11 protocol converter' – http://www.brouhaha.com/~eric/retrocomputing/rekursiv/rekursiv.txt

[6] Ian ended up sharing an office with David, which wasn't a good idea. When David disagreed with Ian he used to joke 'I think it's time for your annual decrement.'

[7] http://www.brouhaha.com/~eric/retrocomputing/rekursiv/

[8] Personal e-mail 31 March 2009

[9] Personal e-mail 31 May 2009

[10] http://www.ir.bbn.com/~bschwart/adventure.html

Out of Memory

Roger Orr has been tracking down some 'out of memory' problems.

I was inspired to write this article by some recent problems with programs failing because they ran out of memory. Understanding the problems proved a little tricky and the 'standard' tools used were slightly misleading.

The problem

Memory is one of the main resources needed by a computer program, but unfortunately it can be more difficult to deal with than some other resource types. There are three main reasons for this:

Firstly, there are typically lots of requests for memory. Modern programming styles allocate large numbers of dynamic objects from the heap and it is common to count the number of memory allocations in thousands, if not millions. Contrast this with file handles, for example, where a typical program might only open a handful of files. Techniques that involve manually tracking each individual resource may be viable for some types but are rarely effective for memory use.

you might think, 'out of memory' only has one cause – there's no memory left

Secondly, all memory in use looks much the same; if you examine program memory it is normally hard to identify what a particular memory location is being used for and where a particular byte was allocated. Indeed, just finding the boundaries between memory allocated in separate allocation requests is often a hard problem. Again, most resource types do not have this problem; the identifier and contents of the resource usually help with both getting the use of the resource and finding its original allocation, and the boundary of the resource (file handle, TCP/IP connection, etc) is clear.

Finally, behind the scenes, memory is complicated. We're used to dealing with this complication with other resource types and expect to deal with different failure modes. For example, failing to connect to a URL in Java will raise an IOException but the actual runtime type of the exception could be a `MalformedURLException`, a `ProtocolException`, a `SocketException`, an `UnknownHostException`, etc. This range of exception types helps to differentiate between the various underlying causes for the failure and hence resolve the cause of the error. Not so with memory (at least in the environments I know well) – memory allocation failures all look pretty much the same.

For example, the man page for malloc on many systems says: 'If there is an error, returns a `NULL` pointer and set `errno` to `ENOMEM`.' Some systems add `EAGAIN` as an additional value, but this focuses on whether it's worth retrying, not on why the error happened.

There are a number of tools available on most development systems that provide automated memory tracking. This typically adds some runtime overhead and/or uses additional memory when it is being used, but does provide a great deal of help with the first two points (tracking the large number of individual memory allocations). In my experience there's less help with understanding the complexity of memory allocation in cases where this is contributing to the memory issue.

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

What might out of memory mean?

But surely, you might think, 'out of memory' only has one cause – there's no memory left? Alas it isn't that simple, but to understand why not we have to expand a little on memory management.

Application

In most high level languages the application programmers allocate the bulk of their memory implicitly. For example when adding two strings together the runtime library takes care of allocating the memory for the new concatenated string; the runtime library is also responsible for deallocating the memory when the string is no longer required, perhaps using deterministic deallocation when the object goes out of scope or with the help of a garbage collector.

However, either explicitly or behind the scenes, the address of the allocated memory is needed. In most desktop architectures the address is a simple number; in some architectures a more complicated structure, perhaps involving a segment and an offset, may be used. Whatever the actual scheme though, these addresses have a finite range of values.

This provides the first, and most fundamental, limit to the amount of memory your process can allocate. The allocation may fail simply because there are no valid addresses left unused in the address space of the process.

So, for a typical 32bit process on Windows, valid addresses range from 0x00010000 to 0x7FEFFFF [1]. If a process is using that entire range then it won't be able to allocate any more memory and even purchasing another gigabyte of RAM won't help.

Additionally, for common operating systems (where code and data share the same address space) the range of available addresses is further restricted by any program code that is loaded into memory. Many environments also support code modules (DLLs or sharable libraries) which are often loaded at a variety of addresses in memory and further fragment the range of available addresses.

Other memory allocations may also come out of the same address range. For example, stack space for the main process and any additional threads, the address range for memory mapped files or shared memory regions, and in some operating systems areas of the address space are reserved for interprocess communication or access to hardware components.

Runtime library

Within the runtime library the memory allocator will request memory from the operating system to fulfil the allocation requests made to it. Since operating system requests are usually quite expensive the runtime is often implemented by allocating big segments from the operating system and sub-allocating this memory to satisfy application requests.

There can be three main problems with such allocators.

Firstly, the allocator itself usually returns memory aligned to a suitable address for all operations – perhaps aligned to an 8 or 16 byte boundary – and also needs memory internally to manage the allocated memory. This can produce a large total overhead if your application makes many small requests. For example, consider the simple C++ program in Listing 1.

When I ran this program on my desktop machine it returned the value 89,134,808. But I've got much more memory than that on my machine!

Listing 1

```
#include <new>
int main() {
  int count = 0;
  while ( new(std::nothrow) char )
     ++count;
  return count;
}
```

The problem is with the overhead of the memory allocator and my example program is a 'worst case' where the overhead dwarfs the size of each allocation. Application programmers who allocate many small objects typically find alternative strategies to reduce this overhead; these may include using a pooled allocator or dealing with arrays of objects rather than individual ones.

In the second case memory can get fragmented – where the allocation and deallocation pattern of the application leads to small gaps between the memory cells remaining in use. Again, depending on memory usage, these small gaps can add up to a sizable amount of wasted memory. This problem is fairly well explored and most memory allocation implementations adopt a variety of schemes to try and reduce the amount of fragmentation. Languages that use garbage collection have an additional trick in that the collector may choose to reduce fragmentation by compacting memory that remains allocated during a garbage collection.

Linux . . . the operating system that likes to say 'Yes'

Finally you may be using multiple memory allocators (for example, a mixed-language program or one using modules each with their own allocator). Each allocator is holding some memory internally that is not in use but has not been returned to the operating system and a request for memory using one allocator may fail because the other allocators have all the available memory in use. This is a difficult problem to solve and causes some headaches for browser writers (among others) trying to support Java virtual machines, Flash plugins, scripting engines and 'normal' HTML rendering all running inside the same process.

Operating system

Looking at the requests made to the operating system these may fail for multiple reasons. Firstly there may be limits set for each process, including how much memory allocation it is allowed, and so the request may be denied when such a limit is reached. Then the operating system's own memory manager may have restrictions on the addresses returned, possibly based on the natural page size of the hardware architecture.

On 32-bit windows, for example, the `VirtualAlloc` call returns memory addresses aligned to a 64K boundary. This provides yet another restriction on the number of OS allocations that can be successful.

Consider the simple C program for Windows in Listing 2.

When I execute this on my desktop machine I get this result:

Listing 2

```
#include <windows.h>
#include <stdio.h>

int main()
{
  int count = 0;
  while ( VirtualAlloc( 0, 1, MEM_COMMIT,
     PAGE_READWRITE ) )
  {
    ++count;
  }
  printf(
    "Allocated %i bytes.\nLast error code %li\n",
    count, GetLastError() );
}
```

```
C:>VirtualAllocateOneByte
Allocated 32664 bytes.
Last error code 8
```

Error code 8 is `ERROR_NOT_ENOUGH_MEMORY` – so I've run out of memory after allocating less than 32Kb!! What I've actually run out of is addresses that are on a 64K boundary.

Finally, we may actually be out of physical memory. Every byte of physical memory might be in use by some process, device drive or the OS itself. However most operating systems support 'memory overcommit' where more memory can be allocated than the machine physically supports. The 'extra' memory is provided by swapping memory out to disk and transparently re-mapping the physical memory addresses freed up to other addresses in the same, or another, application.

So even if we are out of physical memory we might still get something back from an allocation request to the operating system. The OS may swap some existing data out to disk and give us the memory so freed up. Or the OS might reserve some space in the swap file and return us an address range that will only be swapped in when it is accessed. Different operating systems make different decisions on how liberal they are at this point. Each operating system seems to have its own preferred approach and many provide configurable choices!

Windows takes a conservative approach and only returns memory if it either really has the memory or it has successfully reserved space in the swap file; when you hit the swap space limits the allocation fails. This is a conservative policy and means the swap file expands to be bigger than may be strictly necessary but on the positive side you can't get a process failure when you later try to write to the allocated memory. Solaris also uses a conservative ('eager') approach by default but this can be configured (see the `vm_swap_eager` attribute). Linux takes a more liberal approach (The operating system that likes to say 'Yes') and by default returns allocated memory with only a quick check for 'obvious' overcommitment. This has the advantage that the swap space usage is less but the disadvantage that a subsequent attempt to actually use the memory may fail because there is no room to expand the swap space. Later versions of Linux provide some kernel configuration settings (see `vm.overcommit_memory` and `vm.overcommit_ratio`) that allow you to configure the behaviour. You may (or may not) find this extract from the relevant man pages reassuring: 'Depending on the percentage you use, in most situations this means a process will not be killed while accessing pages but will receive errors on memory allocation as appropriate.'

Lastly, many operating systems allow the user to reserve spaces in the address range and commit and free individual pages in the range as required. Many operating systems also use this technique to manage the stack – the stack is created in an address range of the maximum stack size but marked as 'reserved' and actual pages of memory are only committed to the stack as it grows. This 'lazy allocation' avoids up-front use of megabytes of memory for simple programs that have a shallow stack.

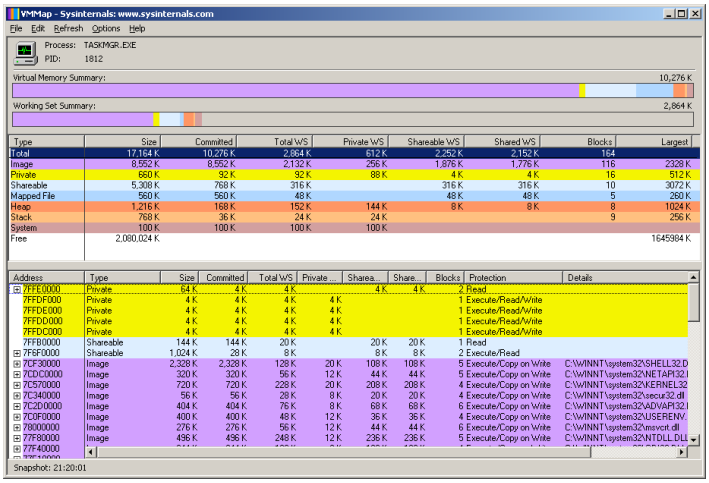
Putting it all together

For a memory allocation to fail any one of a number things may fail. First off we may not have an available address range in our address space for the allocated memory – both size and alignment may be important here. We may have inadequate resources for the runtime allocator and any housekeeping it needs. We may have exceeded some allocation quota, or lack a suitable target for an allocation from the operating system or finally we may have run out of free physical memory or disk space.

In the problem cases that inspired the article the limiting factor was the address space, but for slightly different reasons in each case.

In one case the application used a number of vectors internally and, after some analysis, we discovered that the out of memory problem occurred when a large vector had an additional item added. This extra item hit the current capacity of the vector and so required a re-allocation of the underlying memory buffer. The buffer was 100 Mb in size and the vector

Figure 1



class used had a simple doubling algorithm when resizing which meant a request was made for 200 Mb of contiguous memory for the new buffer. This request failed because there was no single free section in the address range that was this large. The application failed with an out of memory error although it had only allocated 800 Mb or so of memory and was running on a machine with 4 Gb of physical memory.

This was on Windows and I failed (then) to find a standard tool that helped analyse this problem. The debugger **`Windbg`** from Microsoft Debugging Tools for Windows came close with its **`!vadump`** command but this simply prints the details of each memory region allocation by the operating system to the process and I wanted to find the largest contiguous free region. Eventually I wrote a quick tool that scanned the address range and reported the size of the largest free memory region; implementation of a derivative of this tool is described below.

One solution to this problem is to replace the vector (that requires a single memory range) with a container that allocates in chunks, such as the deque. Another solution is to preset the capacity of the vector to the final allocation size, but this is hard to do in general! In our case we changed the resize algorithm for large vectors to avoid the doubling and hence reduce the need for such large allocations.

In a different case we had a message switching application written in Java running on a 32bit JVM on Solaris. The application would generate out of memory errors even though tools such as 'top' showed our application was using less than 2 Gb of memory (out of a possible limit over well over 3Gb) and we had many gigabytes of free physical memory on the machine.

The fundamental problem here also turned out to be running out of address space – our application was creating a pair of threads for each connection it was managing and each thread was reserving a stack of 1 MB. The threads all had a shallow call depth, so in fact only the first 8 Kb of each stack was actually being committed, the rest of each stack was just left as reserved address space. By the time you have a thousand threads in the program that means nearly 1 Gb of address space reserved and hence not available for other allocations.

There were three possible solutions we explored here. The first solution was to reduce the number of threads, but this proved hard in our case because the threads were being created by a third party library that we were not able to change. We explored reducing the size of the thread stack, but failed to find a mechanism that allowed us to do so (in some cases operating system APIs provide the feature, but higher level abstractions may not expose it to application code).

The eventual solution we went for in this case was to run in a 64 bit virtual machine. This slightly increased the actual amount of allocated memory used but enormously increased the available address range. The 1 Gb of reserved space drops from one quarter of the whole range in a 32bit program to a tiny fraction of the available 64 bit address range.

However, we found in this case too a lack of standard tools that provided diagnostics for the 'out of addressability' problem. In this case we eventually looked at the data from the /proc pseudo-filesystem and used some scripting to identify the reserved memory ranges.

A useful (Windows) tool

Earlier this year SysInternals [2] released **`vmmap`**, which is a Windows tool that provides a nice graphical view of the address space for a process and also provides the summary (totals and largest address ranges in various categories). I suspect this tool was written because running out of addressability is becoming more of an issue for other programmers too. (Figure 1)

Under the covers

I thought it would be instructive to provide source code (Listing 3) for a program which prints the address ranges for a Windows process in a similar fashion to **`vmmap`**.

The program expects a list of process IDs on the command line and simply prints the virtual address map, and a summary line, for each one.

The algorithm obtains the user-mode address range from **`GetSystemInfo`** and iterates through the range using **`VirtualQueryEx()`** to get the memory information for each address block. Once the complete range has been read the details for each block and the summary is produced (Note that the **`MEMORY_BASIC_INFORMATION`** structure holds additional information that this simple program does not display).

Here is some sample output from this program in action:

```
C:>virtmap 1812
Processing: 1812
       8K at 00010000 committed
      56K at 00012000 free
       4K at 00020000 committed
      60K at 00021000 free
     240K at 00030000 reserved
...
       4K at 7FFDF000 committed
       4K at 7FFE0000 committed
      60K at 7FFE1000 reserved
Committed: 10Mb, reserved: 6Mb, free: 2031Mb
(biggest: 1607Mb)
```

Listing 3

```
#include <windows.h>
#include <iostream>
#include <iomanip>
#include <iterator>
#include <string>
#include <vector>
// This class holds the virtual memory map for a
// process
class VirtMap
{
public:
  VirtMap();
  int VirtMap::process( HANDLE hProcess );
  void printOn( std::ostream& os ) const;
private:
  void add(
     MEMORY_BASIC_INFORMATION const & mb );
  std::vector<MEMORY_BASIC_INFORMATION> m_info;
  ULONG m_commit;
  ULONG m_reserve;
  ULONG m_free;
  ULONG m_biggest;
};
// Stream a memory map
std::ostream & operator<<( std::ostream & os,
   VirtMap const & virtMap )
```

Listing 3 (cont'd)

```
{
  virtMap.printOn( os );
  return os;
}
// Stream a memory block
std::ostream & operator<<( std::ostream &os,
   MEMORY_BASIC_INFORMATION const & mb)
{
  os << std::setw(8) << mb.RegionSize / 1024 <<
     "K at " << mb.BaseAddress;
  switch( mb.State )
  {
    case MEM_COMMIT: os << " committed"; break;
    case MEM_FREE: os << " free"; break;
    case MEM_RESERVE: os << " reserved"; break;
  }
  return os;
}
// Class VirtMap implementation
VirtMap::VirtMap()
: m_commit( 0 )
, m_reserve( 0 )
, m_free( 0 )
, m_biggest( 0 )
{
}
// Handle a single process
int VirtMap::process( HANDLE hProcess )
{
  int ret(0);
  SYSTEM_INFO SystemInfo;
  GetSystemInfo( &SystemInfo );
  PVOID baseAddress(
     SystemInfo.lpMinimumApplicationAddress );
  while ( baseAddress <
     SystemInfo.lpMaximumApplicationAddress )
  {
    MEMORY_BASIC_INFORMATION mb = { 0 };
    if ( ! VirtualQueryEx( hProcess, baseAddress,
       &mb, sizeof( mb ) ) )
    {
      int ret = ::GetLastError();
      std::cerr << "Cannot query memory at: "
         << baseAddress << ": " << ret
         << std::endl;
      break;
    }
    add( mb );
    baseAddress = (PCHAR)mb.BaseAddress
       + mb.RegionSize;
  }
  return ret;
}
// Add an item to the memory map
void VirtMap::add(
   MEMORY_BASIC_INFORMATION const & mb )
{
  m_info.push_back( mb );
  if ( mb.State & MEM_FREE )
  {
    m_free += mb.RegionSize;
    if ( m_biggest < mb.RegionSize )
       m_biggest = mb.RegionSize;
  }
  else if ( mb.State & MEM_RESERVE )
  {
    m_reserve += mb.RegionSize;
  }
  else if ( mb.State & MEM_COMMIT )
  {
    m_commit += mb.RegionSize;
  }
}
// Print the memory map
void VirtMap::printOn( std::ostream& os ) const
{
static const ULONG megabyte = 1024L * 1024L;

  std::copy( m_info.begin(), m_info.end(),
     std::ostream_iterator<
     MEMORY_BASIC_INFORMATION>(os, "\n" ) );
  os <<
    "Committed: " << m_commit / megabyte << "Mb"
    ", reserved: " << m_reserve / megabyte
    << "Mb", free: " << m_free / megabyte
    << "Mb" " (biggest: " << m_biggest / megabyte
    << "Mb)";
}
int main( int argc, char **argv )
{
  int ret = 0;
  for ( int idx = 1; idx != argc; ++idx )
  {
    int const pid = atoi( argv[idx] );
    HANDLE const hProcess = OpenProcess(
       PROCESS_QUERY_INFORMATION, FALSE, pid );
    if ( hProcess == 0 )
    {
      ret = ::GetLastError();
      std::cerr << "Unable to open process "
         << pid << ", error: " << ret
         << std::endl;
      break;
    }
    else
    {
      std::cout << "Processing: " << pid
         << std::endl;
      VirtMap virtMap;
      ret = virtMap.process( hProcess );
      CloseHandle( hProcess );
      if ( ret != 0 )
        break;
      std::cout << virtMap << std::endl;
    }
  }
  return ret;
}
```

Listing 3 (cont'd)

Conclusion

As we reach the end of the road for 32 bit operating systems memory once again has become a scarce resource – but this time round it is usually not physical memory that we're short of but managing to get enough address space in our processes to access the memory we need.

I hope this overview of some of the complexity of memory management will give people some help with identifying the cause of 'out of memory' problems. ■

Notes and resources

[1] Newer Windows versions support the /3GB switch which potentially adds an extra gigabyte of addressability to a process. See http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.mspx for more details.

[2] See http://technet.microsoft.com/en-us/sysinternals/

Safe and Efficient Error Information

Matthew Wilson investigates.

An STLSoft [1] user approached me recently, enquiring about the `stlsoft::error_desc` component (see sidebar) and Microsoft's so-called 'safe string' library [2]. The component – it's a class, but usually used as a function – takes an error code, and elicits the equivalent string form (via `strerror()`), for insertion into an error report statement, as in:

```
int code = ENOENT;

std::cerr << "error: "
   << stlsoft::error_desc(code) << std::endl;

ff::fmtln(std::cerr, "error: {0}",
   stlsoft::error_desc(code));
```

Why this is advantageous over invoking `strerror()` directly will be discussed later in the article.

The user's enquiry was whether the component should be implemented in terms of `strerror_s()` in contexts where the safe string library is available and active. I actually thought I'd already done this work, but finding I was wrong set about doing so; it's now available from 1.9.84 onwards. In making the changes I had cause to wonder about the safety of `strerror()`. This article looks at reasons why it might not be safe, and examines different ways of providing string error information. It then describes a simple technique for doing so, which, while neither rocket-science nor panacea, is safe, efficient, and guaranteed to be fail-free (because it's done at compile-time). The technique is equally applicable to C and C++ libraries, but I will present it as if the library is written in C, and consumed by C and C++.

Safety

The `strerror()` function is declared as follows:

```
char* strerror(int errnum);
```

The C standard (7.21.6.2) describes it as follows. (The numbers are mine.)

1 The `strerror` function maps the number in `errnum` to a message string.

2 Typically, the values for `errnum` come from `errno`, but `strerror` shall map any value of type `int` to a message.

3 The implementation shall behave as if no library function calls the `strerror` function.

4 The `strerror` function returns a pointer to the `string`, the contents of which are locale-specific.

Listing 1

```
char* strerror(int errnum)
{
  switch(errnum)
  {
    case ENOMEM:
      return "out of memory";
    case EBADF:
      return "bad file descriptor";
    . . . /* all the other standard codes */
    default:
      return "unknown error";
  }
}
```

The stlsoft::error_desc interface

This class has a very simple interface, reflecting its simple, immutable nature as the string form of an error:

```
// in namespace stlsoft
template <typename C>
class basic_error_desc
{
public:
  explicit basic_error_desc(int err);
  ~basic_error_desc() throw();
  C const* c_str() const throw();
  size_t length() const throw(); // also size()
};

typedef basic_error_desc<char>    error_desc_a;
typedef basic_error_desc<wchar_t> error_desc_w;
typedef basic_error_desc<char>    error_desc;
```

The interface also includes definitions of the string access shims [5, 3] to enable it to be inserted into FastFormat [6, 7] and Pantheios [8] statements, as shown in the examples.

```
char const* c_str_data_a(error_desc_a const& ed);
size_t c_str_len_a(error_desc_a const& ed);
char const* c_str_ptr_a(error_desc_a const& ed);
wchar_t const* c_str_data_w(
   error_desc_w const& ed);
template <typename C>
C const* c_str_data(
   basic_error_desc<C> const& ed);
. . . // etc.
```

5 The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

If we were to set aside condition 4, then it would be extremely easy to satisfy all the others, along the lines of Listing 1 (a locale-ignorant implementation of `strerror()`).

As you can see, there are no runtime state changes at all. Every string already exists, placed in a constant segment in the program by the compiler, and loaded into memory as part of static initialisation [3], so there's no chance of any allocation failure. In supporting condition 4, however, the program must be able to change the associations between error codes and the strings returned. It is possible to envisage a static implementation whereby a switch-based lookup-function such as that shown in Listing 1 would be defined for each locale, as in Listing 2 (pseudo-code for a locale-aware `strerror()`).

Obviously such a scheme would depend on all possible locales being known at the time of creation of the given standard library implementation's writing. I don't know whether this is even possible, but

MATTHEW WILSON

Matthew is a software development consultant for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at stlsoft@gmail.com.

Listing 2

```
char* _strerror_English(int errnum);
char* _strerror_French(int errnum);

char* strerror(int errnum)
{
  locale_id_t lid = getlocale();
  switch(lid)
  {
    case "French":
      return _strerror_French(errnum);
    . . . /* and all other cases */
    default:
      /* deliberate fall through */
    case "English":
      return _strerror_English(errnum);
  }
```

it's probably not desirable as the resulting program size would be impacted. Whatever the ins and outs, it's easy to foresee that more dynamic abilities may be required. Hence condition 4.

Given the need to load locale-specific error strings dynamically, the standard allows for pretty much any implementation, from one as fixed as that shown above to one that has a single buffer and loads the required string from some external resource upon each call. Hence condition 5.

The impact of this is two-fold: thread-safety and multiple calls per statement. Obviously, if a given implementation of **strerror()** does have a single result buffer then if two threads are calling it at the same time there is a likelihood of erroneous results being obtained. Ideally, an implementation will implement **strerror()** to be thread-safe, using thread-specific storage [4, 3].

Even in the case where each thread is provided with its own storage, it is possible for **strerror()** to produce erroneous results. Consider the following seemingly correct code:

```
printf("ENOMEM=%s, EBADF=%s", strerror(ENOMEM),
   strerror(EBADF));
```

Whether this produces the expected result or gives two apparently identical error strings for the different errors is entirely implementation-dependent.

The original purpose of **stlsoft::error_desc** was to obviate this second problem (with the assumption that the first was not an issue). Thus, the following code has no undefined behaviour:

```
ff::fmtln(std::cerr, "ENOMEM={0}, EBADF={1}"
  , stlsoft::error_desc(ENOMEM)
  , stlsoft::error_desc(EBADF));
```

Listing 3

```
// (a) via strerror()
ff::fmtln(std::cerr,
   "could not open file {0}: {1}", file,
   strerror(errno));

// (b) via strerror_s()
char buff[1001];
if(0 != strerror_s(buff,
   STLSOFT_NUM_ELEMENTS(buff), errno))
{
  const char backup[] = "unknown error";
  strncpy_s(buff, STLSOFT_NUM_ELEMENTS(buff),
     backup, STLSOFT_NUM_ELEMENTS(backup) -1);
}
ff::fmtln(std::cerr,
   "could not open file {0}: {1}", file, buff);
```

strerror_r() and strerror_s()

Some platforms provide the non-standard **strerror_r()** function as safe alternative, with the following signature:

```
int strerror_r(int errnum, char *buff,
   size_t cchBuff);
```

Microsoft's new so-called 'safe string' library provides the roughly equivalent **strerror_s()**, which has the slightly different form:

```
int strerror_s(char *buff, size_t cchBuff,
   int errnum);
```

Each takes an error code, a pointer to a buffer to receive the error string, and a buffer size (in characters). Use of either function obviates both problems with **strerror()**. That's good news.

However, there are several downsides:

1. Neither are standard (nor particularly ubiquitous)
2. They're far less convenient to use, leading to significant reductions in transparency
3. If you don't pass in a sufficiently large buffer, you're not informed how large it should have been. In fact, with **strerror_s()**, you don't even get told that you've provided too-little space! In either case you're forced to code up a guess loop

If you don't pass in a sufficiently large buffer, you're not informed how large it should have been

We can illustrate the second downside. Compare the example in Listing 3, showing the different amount of code that must be written to report an error.

That's one line of code, and one statement, for **strerror()**; seven lines, two variables and three statements for **strerror_s()**. Given that code reliability has a significantly relation to expressiveness, something's smelling a bit squiffy here.

Furthermore, the **strerror_s()** was not really correct. Oh, sure, we're very unlikely to find an error code string that has more than 1000 characters. But it's not impossible. So a full and correct implementation would actually look more like Listing 4.

Who wants to see such muck in their application code? If a façade such as **stlsoft::error_desc** (see sidebar) was not already available, having to use either of these 'safer' monstrosities would be a substantial motivation for its creation. The new version of **stlsoft::error_desc** encapsulates such an auto_buffer-based guess loop, where it cannot intrude on the transparency of client code.

Library errors

Let's now turn our attention away from **strerror()** and its troubled kin, and look at a simple technique that I use in several standard libraries for eliciting error strings in a safe and efficient way. First, let's look at a use case that illustrates why we might be concerned with the performance of error→string conversions, as well as correctness.

The Pantheios logging API library uses string access shims [5, 3] to elicit the string forms of arguments. Wherever possible it elicits the length of the argument's string form directly, e.g. it invokes **std::string::size()**. When dealing with **strerror()**'s result – and that obtained from **strerror_r()**/**strerror_s()**, for that matter – **strlen()** must be invoked. In the case where the strings are being obtained from literals, known at compile-time, this is a pointless waste of cycles.

The technique I use in several of my libraries avoids this wasted effort, by defining strings as literals, and keeping a record of the string pointer and

Listing 4

```
// (b) via strerror_s(), correctly
int errnum = errno; // Remember here, in case
                    // overwritten
stlsoft::auto_buffer<char, 128> buff(128);
for(;; buff.resize(buff.size() * 2))
{
  if(0 != strerror_s(buff, buff.size(), errno))
  {
    ff::fmtln(
       std::cerr, "could not open file {0}: {1}",
       file, "unknown error");
    break;
  }
  else
  {
    if(::strlen(buff.data()) == buff.size() - 1)
    {
      // Might not be all of string, so loop
    }
    else
    {
      ff::fmtln(
         std::cerr,
         "could not open file {0}: {1}",
         file, buff.data());
      break;
    }
  }
}
```

the length in a statically-defined data structure. To see how this is done, let's postulate a fictional library **AcmeLib**, whose API looks like the Listing 5.

Listing 5

```
/* AcmeLib.h */
enum ACMELIB_RC
{
  ACMELIB_RC_SUCCESS = 0,
  ACMELIB_RC_ERROR,
  . . .
  /* insert new values *before* this */
  ACMELIB_RC_max_value
};

int AcmeLib_doStuff(char const* whatStuff);

char const* AcmeLib_getRcStringPtr(
   ACMELIB_RC rc);
size_t AcmeLib_getRcStringLen(ACMELIB_RC rc);

#ifdef __cplusplus
namespace stlsoft
{
  /* Shims allow insertion of ACMELIB_RC codes
   * into FastFormat, Pantheios, etc.
   */
  inline char const* c_str_data_a(ACMELIB_RC rc)
  {
    return AcmeLib_getRcStringPtr(rc);
  }
  inline size_t c_str_len_a(ACMELIB_RC rc)
  {
    return AcmeLib_getRcStringLen(rc);
  }
}
#endif /* __cplusplus */
```

Listing 6

```
char const* AcmeLib_getRcStringPtr(ACMELIB_RC rc)
{
  switch(rc)
  {
    case ACMELIB_RC_SUCCESS:
      return "operation completed successfully";
    case ACMELIB_RC_ERROR:
      return "operation failed";
    default:
      return "unknown error";
  }
}
size_t AcmeLib_getRcStringLen(ACMELIB_RC rc)
{
  return strlen(AcmeLib_getRcStringPtr(rc));
}
```

The simple but slow way to implement this would be as shown in Listing 6:

But we don't have to waste the fact that the compiler knows all about the literals. We can capture this information at compile-time, rather than calculating it at runtime, every time. First we need a data structure. Assuming that **AcmeLib** deals only in multibyte-character types, we can define **AcmeLibRcString_t_** (Listing 7), and a function to perform lookups on arrays of it. (FYI, the use of trailing underscore is my personal, standards-adhering, affectation to denote that a particular symbol is component-internal, rather than part of an API. Feel free to copy/ignore as you choose.)

This function attempts to find the entry matching the code, and returns its pointer (return value) and the length (out parameter). If no matching entry is found, it uses the default string. Note that two searches are performed. The first will work if the range of codes is contiguous and starts at 0 (and the entry array is ordered in the same way); since most error code enumerations (or integer constants, otherwise) do this, it's a worthwhile optimisation. If that fails, then a linear search is carried out.

the painful differences in behaviour with relative inclusion paths between different compilers

With this worker function, only a couple more things are required. First, we need to define the array of structures, which depends on the definition of a couple of macros (Listing 8).

Now you should be able to see how the technique is able to preserve the knowledge about the code string lengths. Each use of the **SEVERITY_STR_DECL()** macro defines the code string and defines a static instance of **AcmeLibRcString_t_** which associates the code, the string and the string length. This is all done at compile time, so there're no runtime issues, and everything is already available when it is called.

The one criticism I make every time I use this technique is that there's a violation of DRY SPOT [9,3] in the fact that the **SEVERITY_STR_DECL()** and **SEVERITY_STR_ENTRY()** must correspond one to one. I always think of doing a bit of extra pre-processor smarts and defining them as two-parameter macro invocations in a separate file, and then including that file twice within **AcmeLib_LookupErrorStringA_()**, surrounded by appropriate pre-processor modifications to ensure that the end result is the same. Something along the lines of Listing 9.

That I haven't is down to the painful differences in behaviour with relative inclusion paths between different compilers rather than just pure sloppiness. (Well that's my story and I'm sticking to it!) And thorough unit-testing helps prevent mistakes. But you should consider doing it 100% DRY.

Listing 7

```
struct AcmeLibRcString_t_
{
  int         code;
  char const* str;
  size_t      len;
};

static char const* AcmeLib_LookupCodeA_(
  int               code
, AcmeLibRcString_t_ const** mappings
, size_t            numMappings
, size_t*           len
)
{
  static const defaultString = "unknown error";
  /* Use Null Object (Variable) pattern here for
   * len, so do not need to check elsewhere. */
  size_t  len_;
  if(NULL == len)
  {
    len = &len_;
  }
  /* Checked, indexed search. */
  if( code >= 0 &&
      code < ACMELIB_RC_max_value)
  {
    if(code == mappings[code]->code)
    {
      return (*len = mappings[code]->len,
         mappings[code]->str);
    }
  }
  /* Linear search. Should only be needed if
   * order in AcmeLib_LookupErrorStringA_()
   * messed up. */
  { size_t i; for(i = 0; i != numMappings; ++i)
  {
    if(code == mappings[i]->code)
    {
      return (*len = mappings[i]->len,
         mappings[i]->str);
    }
  }}
  return (*len = NUM_ELEMENTS(defaultString) - 1,
     defaultString);
}
```

Anyway, we're not actually there yet. The final action is to implement the AcmeLib API functions in terms of these facilities (see Listing 10).

Whether we are asking for string or string length, we get a straight lookup into a static table (or a linear search, at worst, if you've chosen to mal-order your enumerators). No other functions are involved, including **`strlen()`**, so this has the added advantage of total modularity to go with the absence of wasted cycles.

It may appear at first blush that this is just an exercise in premature optimisation. Well, it certainly has an aspect of optimisation. But it's also a matter of moving processing back a stage in the cycle, namely into compilation rather than runtime. This is almost always a good thing. In this case, because the strings are effectively hard-wired into the program, 'created' (or, more properly, loaded) during static initialisation [3], they cannot fail to be present when needed by any executing code, even code that's executing (in the dynamic initialisation phase [3]) before **`main()`**. So, for libraries such as FastFormat [6, 7] and Pantheios [8] that ensure availability to all (C++) compilation units by coupling reference-counted APIs [3] with Schwarz counters [10, 3] – a technique that probably warrants another article in itself – having the error codes available via this technique is essential.

Listing 8

```
#define NUM_ELEMENTS(x) (sizeof(x) /
sizeof(0[x]))

#define SEVERITY_STR_DECL(rc, desc) \
  static const char s_str##rc[] = desc; \
  static const AcmeLibRcString_t_  s_rct##rc = \
   { rc, s_str##rc, NUM_ELEMENTS(s_str##rc) - 1 }

#define SEVERITY_STR_ENTRY(rc)  \
  &s_rct##rc

static char const* AcmeLib_LookupErrorStringA_(
   int error, size_t* len)
{
  SEVERITY_STR_DECL(ACMELIB_RC_SUCCESS,
     "operation completed successfully"  );
  SEVERITY_STR_DECL(ACMELIB_RC_ERROR  ,
     "operation failed"                  );

  static const AcmeLibRcString_t_* s_strings[] =
  {
    SEVERITY_STR_ENTRY(ACMELIB_RC_SUCCESS),
    SEVERITY_STR_ENTRY(ACMELIB_RC_ERROR),
  };

  return AcmeLib_LookupCodeA_(error, s_strings,
     NUM_ELEMENTS(s_strings), len);
}
```

Listing 9

```
/* in rcdefs.h: */
ACMELIB_DEFINE_RC(ACMELIB_RC_SUCCESS,
   "operation completed successfully")
ACMELIB_DEFINE_RC(ACMELIB_RC_ERROR,
   "operation failed")

#undef ACMELIB_DEFINE_RC

/* in original file: */
static char const* AcmeLib_LookupErrorStringA_(
   int error, size_t* len)
{
#define ACMELIB_DEFINE_RC(
   c, s) SEVERITY_STR_DECL(c, s);
#include "rcdefs.h"

  static const AcmeLibRcString_t_* s_strings[] =
  {
#define ACMELIB_DEFINE_RC(
   c, s) SEVERITY_STR_ENTRY(c),
#include "rcdefs.h"
  };
  . . .
```

Listing 10

```
char const* AcmeLib_getErrorString(ACMELIB_RC
code)
{
  return AcmeLib_LookupErrorStringA_((int)code,
     NULL);
}

size_t AcmeLib_getErrorStringLength(
   ACMELIB_RC code)
{
  size_t  len;
  AcmeLib_LookupErrorStringA_((int)code, &len);
  return len;
}
```

Inspirational (P)articles

Frances Buontempo introduces Paul Grenyer's inspiration.

In CVu May 2009, I shared the inspiration created by reading an interview with Donald Knuth. I want this to become a regular feature because I passionately believe sharing positive experiences causes ripples, as though the inspiration spark permeates through the ether and hits surrounding people. Terry Pratchett has written about such inspiration particles in various books, hence the title of this series.

Today, Paul Grenyer would like to share the pleasure and delight of writing software using unfamiliar libraries just working straight away. If you have a story to share of a recent uplifting or encouraging moment, please send it to frances.buontempo@gmail.com.

Tomcat Servlet with Spring Timer

I recently had a requirement to write a service, in Java, that monitors a directory and when new files with the correctly formatted name appear, send them to another system. All fairly simple stuff. There are many different ways of writing Java services, but we use Tomcat quite heavily, so rather than investigate another way, I decided to write a Tomcat servlet to act as the service.

I started off by extending **`GenericServlet`** and overriding the **`init`** and **`destroy`** methods to write log messages to standard out. Then I wrote the appropriate `web.xml` to tell Tomcat about the servlet and wrapped it all up in a **`war`** file (basically a zip file with a Tomcat specific directory layout) and deployed it to my local Tomcat installation. I finally checked the logs and found the log messages I'd put in the code. Not bad going for twenty minutes work and my first Tomcat servlet written from scratch.

We've been gradually learning about Spring recently and I remembered reading that Spring had timers that would be perfect for polling the directory for files. So I integrated Spring into my servlet, repackaged and redeployed it and then checked the logs to make sure the Spring application context had fired up correctly. It had.

Next I created a Ticker class by implementing the Java TimerTask interface and implementing the run method to write **`"Tick"`** to standard out. I then registered the class as a Spring bean and created a Spring **`ScheduledTimerTask`**, set the tick interval to one second and created an anonymous **`TimerFactoryBean`**. Making the **`TimerFactoryBean`** anonymous causes it to be instantiated when the Spring context is started, rather than waiting for an explicit instantiation from code somewhere. So, what should happen is that the ticker starts as soon as the application starts. Sure enough as soon as I repackaged and deployed, **`"Tick"`** was written to standard out every second.

It occurred to me that the class extending **`GenericServlet`** was redundant. So, not expecting it to work, I removed the class from the servlet and `web.xml` entirely and repackaged and redeployed. That's when I had my real 'Whoah! That's really neat!' moment. To my amazement and joy the ticker started again and kept ticking every second. I already knew Spring and Tomcat worked well together, but having Tomcat start the Spring context without needing a servlet class is pure genius.

It may seem like such a small and simple thing, but creating my first Java service and Spring timer and having them work together in a very simple way was a real revelation for me.

Safe and Efficient Error Information (continued)

Finally, though much less generally important, such ultimate modularity – not even depending on the C standard library – is not usually significant, but with very small self-contained C libraries it can be. And it certainly does no harm to your portability.

Localisation?

The elephant in this particular living room is, of course, that the technique suggests that there is only one human-language string representation of the error codes. That may be the case where you're developing for an in-house product, or one aimed at only one nationality.

It may also be the case if you don't plan on propagating such low-level errors to any users, only storing them in logs. (Although it smacks of Anglophone arrogance, of course, it's still likely that many/most of the programmers of your product will be able to read the English error messages.)

If neither of these assumptions hold true, then you need to consider internationalising your error code strings. The problem here is that as soon as you allow things to be dynamic, you get into the possibility of failures to load/allocate, and the difficulty of knowing the code string size a priori (and therefore efficiently). In this case, prefer to code up dynamic error code to string translation functionality, but have it fallback to statically defined (using the above technique) English strings. This enables you to fulfil the requirements of, say, your application logging whether or not the locale-specific strings are present and can be loaded.

Implementing such a fail-soft mechanism is beyond the scope imposed by the remaining space of this article. Interested readers may be stimulated to write further on the matter … ■

References

[1] The STLSoft libraries; http://stlsoft.org/

[2] http://msdn.microsoft.com/en-us/library/8ef0s5kh(VS.80).aspx

[3] *Imperfect C++: Practical Solutions for Real Life Programming*, Matthew Wilson, Addison-Wesley, 2004

[4] *Programming with POSIX Threads*, David R. Butenhof, Addison-Wesley, 1997

[5] *Extended STL, volume 1: Collections and Iterators*, Matthew Wilson, Addison-Wesley, 2007

[6] 'An Introduction to FastFormat, part 1: The State of the Art', Matthew Wilson, *Overload* 89, February 2009

[7] 'An Introduction to FastFormat, part 2: Custom Argument and Sink Types', Matthew Wilson, *Overload* 91, April 2009

[8] The Pantheios logging API library; http://pantheios.org/

[9] *The Pragmatic Programmer*, Dave Thomas and Andy Hunt, Addison-Wesley, 2000

[10] *C++ Gems*, Stanley Lippman (ed.), Cambridge University Press, 1998

Code Critique Competition 57

Set and collated by Roger Orr.

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

Can someone please help me to understand why the following trivial C program crashes?

Note: You may answer the question in C, or convert the program to C++ (with justification).

Last issue's code is shown in Listing 1.

Listing 1

```
#include <stdio.h>
#include <string.h>

typedef enum {
  Hearts,
  Diamonds,
  Clubs,
  Spades
} Suit;

typedef struct Card
{
    Suit suit;
    int value;
} Card;

typedef Card Deck[52];

void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 51; i++)
  {
    myDeck[i]->suit = i % 4;
    myDeck[i]->value = i % 13;
  }
}

void PrintDeck(Deck * myDeck)
{
  int i = 0;
  for(;i < 52; i++)
  {
     printf("Card %d %d\n", myDeck[i]->suit,
       myDeck[i]->value);
  }
}

int main()
{
  Deck myDeck;
  memset(&myDeck,0,sizeof(Deck));
  LoadDeck(&myDeck);
  PrintDeck(&myDeck);
  return 0;
}
```

Critiques

Bingfeng Zhao <Bingfeng.Zhao@ca.com>

First, we must clear what the type of `Deck` is. The definition of it is:

```
typedef Card Deck[52];
```

So `Deck` is an array of 52 Cards. Then, why does the following code crash?

```
void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 51; i++)
  {
    myDeck[i]->suit = i % 4;
    myDeck[i]->value = i % 13;
  }
}
```

Here, the type of `myDeck` is a pointer to an array of 52 Cards. Then, what does `myDeck[i]` mean?

Here, we take `myDeck` as a pointer to an array, it's valid in C if the pointer is really the address the first element of an array. Unfortunately, here it's not! `myDeck` is a pointer to an array of 52 Cards, so `myDeck[i]` means the `i`th element of an array of array of 52 cards. This is definitely an error.

The fix is simple, instead of

```
myDeck[i]->suit = i % 4;
myDeck[i]->value = i % 13;
```

use

```
(*myDeck)[i].suit = (Suit)(i % 4);
(*myDeck)[i].value = i % 13;
```

Of course, we need to also update `PrintDeck()`.

Ian Bruntlett <ianbruntlett@hotmail.com>

[Ed: Ian's entry annotated the code directly.]

Change `LoadDeck` argument from `Deck *` to `Card *`, fix missing last element and index by card:

```
void LoadDeck(Card * pFirstCard)
{
  int i = 0;
  for(; i < 52; i++)
  {
    pFirstCard[i].suit = i % 4;
    pFirstCard[i].value = i % 13;
  }
}
```

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

Change `PrintDeck` argument from `Deck *` to `Card *`:

```
void PrintDeck(Card * pFirstCard)
{
  int i = 0;
  for(;i < 52; i++)
  {
     printf("Card %d %d\n", pFirstCard[i].suit,
        pFirstCard[i].value);
  }
}
```

Then change the calls to `LoadDeck` and `PrintDeck` to take argument of type `"(Card *) &myDeck"`.

Carl Bateman <CarlBateman@hotmail.com>

'Can someone please help me to understand why the following trivial C program crashes?'

Almost certainly... while we're waiting, here's what I found after playing around with the code in Visual Studio.

In the function `void LoadDeck(Deck * myDeck)`, `myDeck` is a pointer to a `Deck`, `Deck` is `typedef`'ed as `Card[52]`. `myDeck[i]` accesses the `i`th element from the base address of `myDeck`... So, the problem is that, in this case, an element is a `Deck`, which is 52 times greater that a `Card`. So after the first loop iteration the code is trying to access beyond the end of the array.

I found several solutions for this:

1. Rather than indexing `myDeck`, index what it points to:

   ```
   (*myDeck)[i].suit = i % 4;
   ```

   which works because the pointee of `*myDeck` is a `Card`, so the `i`th element is of the right size.

2. The previous method does away with the `->` operator, if you have a particular attachment to this syntax

   ```
   ((*myDeck) + i)->suit = i % 4;
   ```

   works... because we're using pointer arithmetic rather than array indexing and so are accessing a pointer to a `Card`.

3. Rewrite the function to take a pointer to `Card`

   ```
   void LoadDeck(Card * myDeck)
   {
     int i = 0;
     for(; i < 51; i++)
     {
       myDeck[i].suit = i % 4;
   ```

   This cuts down on the brackets.

The same applies to `PrintDeck`, which also incorrectly iterates beyond the end of the array [Ed: it is the other way round – `LoadDeck` is wrong].

It should be

```
for(;i < 51; i++)
```

rather than

```
for(;i < 52; i++)
```

To avoid this sort of inconsistency, consider using macros or consts instead of magic numbers.

eg

```
#define ARRAY_SIZE 52
#define LAST_ARRAY_ELEMENT 51
```

or

```
const int array_size = 52;
const int last_array_element = array_size - 1;
```

Since, the OP asked for help with a C program, I'd no more re-write it in C++ than I would in Pascal... ;)

Nevin ":-l" Liber <nevin@eviloverlord.com>

There are three bugs in the code:

1. Both `LoadDeck()` and `PrintDeck()` are passed pointers to a `Deck`. However, the expression `myDeck[i]` treats them as if they were a `Deck` array, which is wrong (a `Deck` is a `Card` array, which might be why the student may have gotten confused). `(*myDeck)[i]` will access an individual `Card` within the `Deck`.
2. `LoadDeck()` only initialized the first 51 elements of the `Deck`.
3. In `LoadDeck()`, one cannot initialize an element of type `Suit` with an `int` (I can't remember if C enforces this, but C++ certainly does) [Ed: assigning an `int` is legal in C but not C++].

Incorporating those fixes:

```
void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 52; i++)
  {
    (*myDeck)[i].suit = (Suit)(i % 4);
    (*myDeck)[i].value = i % 13;
  }
}

void PrintDeck(Deck * myDeck)
{
  int i = 0;
  for(;i < 52; i++)
  {
    printf("Card %d %d\n", (*myDeck)[i].suit,
      (*myDeck)[i].value);
  }
}
```

Note: while `LoadDeck()` now initializes `myDeck` correctly, it does it in a funky way that I believe was unintentional. It works because 4 (the number of suits) and 13 (the number of ranks) are relatively prime (have no common factors besides 1), so the combination of `i%4` and `i%13` will never both have the same result in any contiguous set of `4*13==52` iterations.

Now, I would rewrite this in C++, for the following reasons:

1. Strong typing. A `Deck` should really *contain* a `Card` array, not *be* one. Also, once it is a type, it can enforce the invariant that the cards are initialized to a standard deck of playing cards.
2. Better output and debugging. Once we have strong typing, we can write stream inserters for all of the value types (`Suit`, `Rank`, `Card`, `Deck`), which helps with debugging.

First, start by expanding on `Suit`:

```
inline std::ostream& operator<<(
  std::ostream& ostream, Suit const& suit)
{
  switch (suit)
  {
    case Hearts:   return ostream << "Hearts";
    case Diamonds: return ostream << "Diamonds";
    case Clubs:    return ostream << "Clubs";
    case Spades:   return ostream << "Spades";
  }

  return ostream << "Suit("
    << static_cast<int>(suit) << ')';
}
```

If the suit is valid, we get the correct textual representation. If it is invalid, we still get to see the number that is stored.

We now make a type for **Rank** and do a similar thing:

```
enum Rank
{
  Ace,
  Two,
  // ...
  Queen,
  King
};

inline std::ostream& operator<<(
   std::ostream& ostream, Rank const& rank)
{
  switch (rank)
  {
    case Ace:   return ostream << "Ace";
    case Two:   return ostream << "Two";
    // ...
    case Queen: return ostream << "Queen";
    case King:  return ostream << "King";
  }

  return ostream << "Rank("
     << static_cast<int>(rank) << ')';
}
```

Note: It is kind of funky for **Two** have a value of 1, **Three** have a value of 2, etc. For an application involving a deck of cards, this is not unreasonable, although we would normally put the **enum** inside a **namespace**, **struct**/**class**, or change the label names. For simplicity, just leave this as is.

Card is a pretty simple **struct**:

```
struct Card
{
  Card() {}
  Card(Rank const& rank, Suit const& suit)
  : suit(suit), rank(rank) {}
  friend std::ostream& operator<<(
     std::ostream& ostream, Card const& card)
  { return ostream << card.rank
     << " of " << card.suit; }
  Suit    suit;
  Rank    rank;
};
```

The **Card::Card()** default constructor is necessary for the next class, **Deck**, because **Deck** first default initializes the **Card** array, then overwrites each card with the correct value.

```
struct Deck
{
  Deck()
  {
    for (size_t c = 0; 52 != c; ++c)
      cards[c] = Card(
        static_cast<Rank>(c % 13),
        static_cast<Suit>(c % 4));
  }

  friend std::ostream& operator<<
    (std::ostream& ostream, Deck const& deck)
  {
    std::copy(deck.cards, deck.cards + 52,
      std::ostream_iterator<Card>
      (ostream, "\n"));
    return ostream;
  }
  Card    cards[52];
};
```

LoadDeck() is pretty simple (and is only necessary if someone wants to reset the order of cards in the **Deck**):

```
inline void LoadDeck(Deck& myDeck)
   { myDeck = Deck(); }
Similarly, PrintDeck() is pretty trivial, as the
   stream inserter does all the work:
inline void PrintDeck(Deck const& myDeck)
   { std::cout << myDeck; }
```

Finally, fix up **main()**. We now pass **myDeck** by reference, not by pointer. Also, we need to remove the call to **memset()**; while not strictly necessary even in the original program, it would do the wrong thing in this C++ version (under most implementations by overwriting the initialized cards with the Ace of Hearts).

```
int main()
{
  Deck    myDeck;
  LoadDeck(myDeck);
  PrintDeck(myDeck);
  return 0;
}
```

Balog Pal <pasa@lib.hu>

This code is a good example of how 'error-correcting eye' works. :) At first read it looked okay except for having 51 in iteration limit, so I wonder if the attached text was 'it looks like it's working'; it could even pass a review on those grounds.

The fortunate crash asks for a closer look, and investigation, until you say: 'Wait, you have **Deck ***, not **Card ***, and apply an index to that!' immediately followed with 'but then how does this thing compile?' Only then realizing, that **->** is used instead of **'.'** that masks the bug. **myDeck[i]->suit** should really be **(*myDeck)[i].suit** to follow the original intent. And the original form thinks we passed an array of 52 decks to operate on. And the **->** operator applied to type **Card[52]** unfortunately works, as we have no real arrays in C just some magic syntax, that works well in most cases, and misfires like this in others. The compiler simply replaces the array with a pointer to its first element, and **->** operates fine on that. **LoadDeck** would set the first card of 52 passed decks instead of setting 52 cards of a single deck. Good reason for crash.

I believe the main question is about this riddle, not about writing a really solid card-game, so let's keep the correction to C and as close to the original as possible.

The trivial approach is certainly to change **Deck *** to **Card *** in function signatures, and pass **myDeck** without **&**. Though fixing the immediate problem I'd classify that as a step backwards. And exploiting the dangerous magic we just recently got burnt with. Having an abstraction for 'deck' and treating it like an object as we see in **main** is a good thing, and makes the code way more literary [so literary it even looks correct as mentioned first ;-]

So we keep **Deck** and its usage, and get rid of the danger source: using the C-array in that raw form. In a C++ solution it would be replaced entirely, but C is C, so we live with the available tools. Instead of making **deck** just **typedef** of the array, let's make it a genuine object with that content:

```
typedef struct Deck
{
  Card cards[52];
} Deck;
```

This shall remove most of the problems related to **deck**, **&deck**, ***deck**, **deck[i]**, **deck.**, or **deck->**, a typo here will be caught by the compiler correctly. It also serves as a better abstraction: if some more information emerges, we just add more fields to the structure without a need to change any code. It also makes it possible to pass **Deck** by value if we like to – and doing so will work as the code is read. No surprises that the black magic really passed a pointer anyway.

Then to make the program work with the change, we replace `myDeck[i]->` instances (I counted 4) with `myDeck->cards[i]`. And after also correcting 51 to 52, the program is functional.

Certainly this 52 stands out as a magic number, have 3 copies that should be changed in sync, and already caused a bug – it works for now but leaves a trap. One solution is to use `sizeof()` in the iterations, leaving just a single instance. Too bad it would be a bug, it only works as long as `Card` is 1 byte... What we really meant is the count of array. It is possible to obtain, but not in a nice way. But even then, we have a half-baked solution, as changing the value in one place. So instead we introduce constants for those elements in the first place, and use those everywhere:

```
#include <stdio.h>
#include <string.h>

typedef enum {
  Hearts,
  Diamonds,
  Clubs,
  Spades
  /* , maxSuit */
} Suit;

typedef struct Card
{
  Suit suit;
  int value;
} Card;

enum
{
  maxSuit = 4,
  maxValue = 13,
  maxCard = maxSuit * maxValue
};

typedef struct Deck
{
  Card cards[maxCard];
} Deck;

void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < maxCard; ++i)
  {
    myDeck->cards[i].suit = i % maxSuit;
    myDeck->cards[i].value = i % maxValue;
  }
}

void PrintDeck(const Deck * myDeck)
{
  int i = 0;
  for(;i < maxCard; ++i)
  {
    printf("Card %d %d\n",
      myDeck->cards[i].suit,
      myDeck->cards[i].value);
  }
}

int main()
{
  Deck myDeck;
  memset(&myDeck,0,sizeof(Deck));
  LoadDeck(&myDeck);
  PrintDeck(&myDeck);
  return 0;
}
```

Defined that way you can simply reconfigure for a 32-card deck, changing `maxValue` to 8. With `Suit` we still have slight redundancy if the bound is separate from its defining `enum` – but the `max` can be put there. In my code I generally have a `max_` in the enumerated lists that is known to be a special value, just outside the expected range – but without knowing that it could be confused as a real item.

Constants are made into `enum` because C has no good genuine support. Originally `#define` was used for the purpose but that should be avoided. `enum` creates good replacement for integer literals, with quite the same attributes. It has no name itself, flagging its only purpose.

A thing that may stand out is using `memset` on the object. In C++ that is certainly a no-no, and we would use constructors. In C it is common to `memset` structure content or allocate them with `calloc`. It is no problem as long as all the members have a fitting binary representation, or are initialized before use. Here, with the fixed bound we overwrite all the items anyway, so `memset` is redundant, but it could serve well in the future.

One more important change I did was adding `const` in `PrintDeck`. Any function that has no business to change the state shall receive only `const` objects. It helps much in reading and understanding the code to make it correct.

Vince Milner <vinnymilner@yahoo.co.uk>

The heart of the problem with this program is confusion around the dereferencing of the `myDeck` parameter in the `LoadDeck` and the `PrintDeck` functions.

The `myDeck` parameter is of type: `"pointer to Deck"` i.e. `"pointer to array of 52 Cards"`. To obtain the suit of a particular `Card`, the program uses:

```
myDeck[i]->suit
```

Although it is intended that '`i`' index the `Card`s contained in the array pointed to by `myDeck`, what actually happens is that `myDeck[i]`, being equivalent to `*(myDeck + i)`, actually steps `i` times the size of `Deck` (i.e. `i` times the size of an array of 52 Cards).

Since `myDeck[i]` is of type `"array of 52 Cards"`, it is converted (by the 'usual unary conversions') to type `"pointer to Card"` pointing to the first `Card` in the array. Hence it is syntactically valid to write

```
myDeck[i]->suit
```

(although for `i > 0`, this will, of course, point to an illegal address).

To correct this, we first need to dereference `myDeck`, before indexing the `Deck` array and accessing suit:

```
(*myDeck)[i].suit
```

An alternative is to define `Deck` as a `struct`:

```
typedef struct {
  Card cards[52];
} Deck;
```

allowing us to do:

```
myDeck->cards[i].suit
```

which is clearer (if slightly longer) and avoids the previous confusion.

Other issues with the program are that:

- the `for` loop in `LoadDeck` ends one `Card` short of the end of the `Deck`,
- the call to `memset` is superfluous since `LoadDeck` immediately resets all the values
- the '`Card`' in '`struct Card`' is redundant in a `typedef` of this form
- `int`s are assigned to type `Suit`, better to use an array to map them

The final corrected program is as follows:

```
#include <stdio.h>

typedef enum {
  Hearts=0,
  Diamonds=1,
  Clubs=2,
  Spades=3
} Suit;

const Suit suits [] =
{ Hearts, Diamonds, Clubs, Spades };

typedef struct {
  Suit suit;
  int value;
} Card;

typedef struct
{
  Card cards[52];
} Deck;

void LoadDeck(Deck * myDeck) {
  int i = 0;
  for(; i < 52; i++) {
    myDeck->cards[i].suit = suits[i % 4];
    myDeck->cards[i].value = i % 13;
  }
}

void PrintDeck(Deck * myDeck) {
  int i = 0;
  for(;i < 52; i++) {
    printf("Card %d %d\n",
      myDeck->cards[i].suit,
      myDeck->cards[i].value);
  }
}

int main() {
  Deck myDeck;
  LoadDeck(&myDeck);
  PrintDeck(&myDeck);
  return 0;
}
```

Joe Wood <joew@aleph.org.uk>

For such a short program it manages to raise many issues. Some of which are really context dependent, and we cannot know the best approach. It seems unlikely that anyone would just want to load a deck of cards and display the resultant deck.

Some basic questions arise:

- Is there only one deck of cards (a singleton)? Are multiple uses going to be close together. i.e. would a simple array of decks suffice?
- Is `Deck` going to be extended (inherited from) to provide different types of hands, for example a single deck or a bridge hand?
- Should `Deck` be able to distinguish between the face of a card and its value, for example an ace in pontoon?
- Should we just display the integer values of the enums or should they be converted to more user friendly strings, for example '1' or 'ace'?

Mending the supplied code

The main problem with the code as presented is when the pointer is dereferenced. As it stands the expression `myDeck[i]->suit` is accessing the `i`th `myDeck` array rather than the `i`th element of the `myDeck` array. Pictorially this is much clearer.

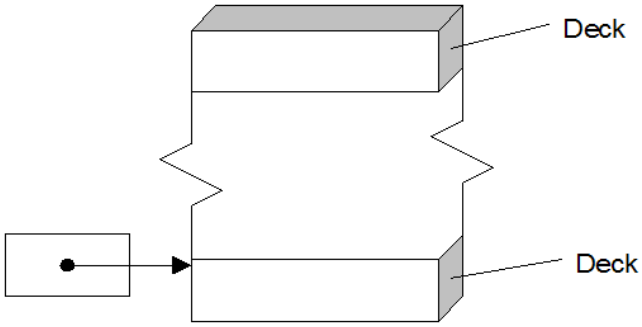


Figure 1

In Figure 1 we access the top few bytes [1] of the ith deck in an array of decks, not the intended outcome. In Figure 2 we access only a single deck in the ith position, the intended outcome.

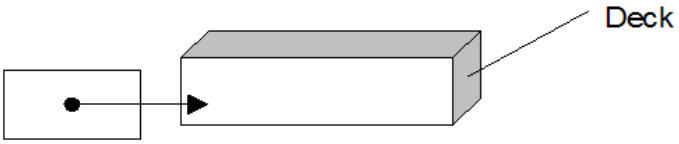


Figure 2

So the desired code should be `(*myDeck)[i].suit` or probably better `myDeck[i].suit` provided we change the parameter [2] of `LoadDeck` and `PrintDeck` to '`Deck myDeck`' since `Deck` is a simple array and passed by address anyway.

This change needs to be propagated around the code in the obvious manner.

There are a few other minor problems:

There is some confusion on the number of cards in the deck, better to introduce a macro, `CARDS_IN_PACK`, and define `deck` using `typedef card deck[CARDS_IN_PACK];`, and remove all references to 51/52.

- As written, `LoadDeck` and `PrintDeck` are only used locally and they should therefore be declared static. This clearly indicates that they are not used outside this file and may allow better code optimisation.
- Why are we calling `memset`, just to immediately overwrite the result in the next line? It is good to initialise variables, but this seems a little odd. A call to a C++ style constructor would be much better, but not possible in C.

Now we must address enums, they may appear to offer type safety, but this is entirely bogus, even in C++ [Ed: not entirely]. We can still write `suit=200` or `suit=-1` or even worse `suit=suit+Spades`. Ada would chuck the three assignments out, and with good cause.

In summary we have the following 2 declarations,

```
#define CARDS_IN_PACK   (52)
typedef Card Deck[CARDS_IN_PACK];
// Deck is an array of Cards
```

and the three routines become:

```
static void LoadDeck ( Deck mydeck )
{
  int i = 0;
  for(; i < CARDS_IN_PACK; i++)
  {
    mydeck[i].suit  = i %  4;
    mydeck[i].value = i % 13;
  }
}

static void PrintDeck ( const Deck mydeck )
{
  int i = 0;
  for(; i < CARDS_IN_PACK ; i++)
```

```
  {
    printf("Card %d %d\n",
      mydeck[i].suit, mydeck[i].value);
  }
}

int main ()
{
  Deck myDeck;
  LoadDeck(myDeck);
  PrintDeck(myDeck);
  return 0;
}
```

A C++ solution???

The start of a possible C++ solution is presented below.

```
class Card
{
public:
  enum Suit {
    Hearts,
    Diamonds,
    Clubs,
    Spades
};

friend std::ostream & operator<<(std::ostream
   &, const Card &);
Card (const Suit & s, const int & v) {
  _suit  = s;
  _value = v;
}

Card() {
  _suit = Hearts;
  _value = 0;
}

private:
  Suit _suit;
  int  _value;
};

std::ostream & operator<<
  (std::ostream & os, const Card & c){
  os << "Card " << c._suit << " " << c._value;
  return os;
}
```

Suit is unsafe as we have discussed before, and **value** is worse. There is no way to represent ace or king for example. What would a value of -1 indicate, or a value of 200? We assume that value is supposed to run from 1 .. 10 plus jack, queen and king but there is no way to ensure this.

Card should almost certainly be equipped with a **copy** constructor and an assignment operator because in any real card game there would be a need to move cards around in a deck.

```
class Deck
{
public:
  Deck()
  : _deck()
  {}

  void LoadDeck () {
    _deck.resize(52);
    for (size_t i=0; i != _deck.size(); ++i) {
      _deck[i] = Card(Card::Suit(i%4), i%13);
    }
  }
  void PrintDeck () const {
    for (size_t i=0; i != _deck.size(); ++i) {
      std::cout << _deck[i] << std::endl;
    }
  }

private:
  typedef std::vector<Card> deck;
  deck _deck;
};
```

and finally the **main** function is simply

```
int main()
{
  Deck myDeck;
  myDeck.LoadDeck();
  myDeck.PrintDeck();
  return 0;
}
```

C++ version can be improved in a number of ways. Proper assignment operators for **Card** and **Deck**. Improve **Deck** to allow different number of cards, and cards to have different values. Probably, make **Deck** a suitable base class for inheritance. But that is for a future exercise.

Commentary

I picked this code because the **typedef** hid the fault. Use of **typedef** is often recommended to make code clearer and easier to change. This fragment illustrates a nasty interaction with raw arrays and a case where the usual advice backfired!

Although the C++ code for this has some advantages, I am not quite persuaded that these are enough to change the code from C. A lot would depend on the context in which the programmer was working.

The Winner of CC 57

I liked Joe's pictures; I thought they were a good way of explaining what was the problem with the code. Understanding this particular problem is tricky, especially for relatively new C programmers.

The entrants picked various ways to resolve the primary issue; to my mind any solution leaving the **typedef** unchanged was likely to cause future problems. The problems caused by decaying arrays to pointers are best avoided completely rather than just papered over! I think the most elegant simple solution was to create a **struct** containing the array of **Card**s as both Pal and Vince did.

I also liked the use of the **suits** array by Vince to avoid implicitly converting an integer into a **Suit** so on balance I have decided to award him this issue's prize.

Code Critique 58

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

Can someone please help me to understand why the program below (Listing 2) crashes? I tried to fix it by using **strncat** but it still doesn't work.

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.

Jeff Sutherland: Agile Software Development in the Enterprise

Chris Oldwood grapples with Jeff Sutherland and Scrum.

The May 2009 talk at the ACCU London branch was given by Jeff Sutherland, the co-creator of Scrum. This was a hugely popular talk hosted at the JP Morgan premises allowing a greater number of attendees, but even so a waiting list was needed. I was one of the 70 or so people lucky enough to get a place.

As the title suggests the talk was aimed squarely at those people who work in larger organisations or distributed teams, perhaps where they are a little more sceptical about the scalability of agile development practices. The blurb on the ACCU web site also describes 'extraordinary financial returns' in case anyone needed a further carrot to listen to one of the founders of the Agile Alliance.

Jeff started with a brief introduction to what Agile Development is, along with a look at the core values from the Agile Manifesto and some of the other Agile Methods such as eXtreme Programming. However, rather than just listing XP as another methodology, he actively promoted the use of the XP technical disciplines such as TDD and Pair Programming. This was highlighted further in a later slide that discussed a more symbiotic relationship between Scrum and XP.

He swiftly moved on to the meat of Scrum and spent some time discussing Velocity and how it is achieved. The key factors appeared to be having a skilled Product Owner that drives the 'Ready' queue to ensure the developers are sufficiently productive, and the features being 'Done' at a level accepted by the business. The first point rose the question of what exactly is meant by 'Ready' and how you find someone who is skilled enough in the business to prepare items for developer consumption – which in the Investment Banking world probably means a trader. However, they don't come cheap, but Jeff's argument is that they would be worth their weight in gold as they are the lynchpin. This somewhat surprising answer became perfect fodder for the pub discussions later. The second point about items only being considered 'Done' when they pass the business acceptance tests, was driven home by a study that showed that 1 hour of immediate testing, when postponed, would grow to become 23 hours of testing later.

After another brief history lesson on how Scrum was invented, Jeff started to go through some case studies to show where they had achieved Hyper-productivity and in the case of one client where that had translated into a significant growth in revenue – 8 times. Along the way we learnt that 65% of requirements change and amusingly that 63% of features are unused, not that any correlation was suggested mind you. The subject then turned to how Scrum scales to large teams with thousands of developers and how even an un-coached team can increase its velocity by 35%, but a coached team can achieve an increase of 200 to 400%. Another favoured sound bite was 'Never outsource to waterfall teams'.

The final two case studies were about failed projects and were used to show how Distributed Scrum not only scales, but was also able to turn the projects around. Although the choice of 'Lines of Code' was not greeted as the best metric for rating productivity (which Jeff agreed with), it did show where the project fortunes changed. Both studies were also analysed using 'Function Points per Developer/Month' to compare the real projects with the expected velocity according to a previous smaller study which compared Scrum to Waterfall (17.8 vs. 2.0) – they both came in a shade over 15.

Not unsurprisingly there were a barrage of questions from the attendees and not nearly enough time to answer them all. The only recourse was to seek out a watering hole and discuss the somewhat compelling statistical information further over a few beers. Somehow we lost focus and before long a quest was on to find out if Brian Blessed had ever been in Star Trek...

Code Critique Competition (continued)

Listing 2

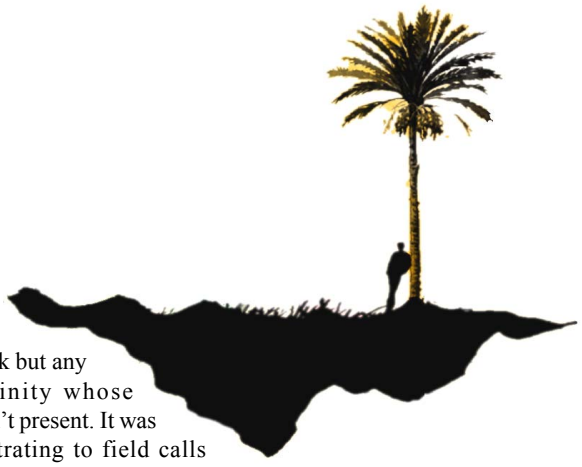
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  char * home_path = NULL;
  char * fullpath = NULL;
  char * fullfile_directory = NULL;
  char buffer[225];
  FILE *f;
  FILE *readfile;

  home_path = getenv("HOME");
  fullfile_directory = strncat(home_path,
    "/snapshot.html", 255-1);
  printf("Searching: %s\n", getenv("HOME"),
    fullfile_directory, 255-1);
  readfile = fopen(fullfile_directory, "r");
  while (!feof(readfile))
  {
```

Listing 2 (cont'd)

```
    if (readfile != NULL)
    {
      f = fopen("/tmp/output.log", "a+");
      fgets(buffer, 256, readfile);
      if (strstr(buffer, "<title>"))
      {
        printf("Extracting title\n");
        fprintf(f, "title: %s", buffer);
      }
      else if(strstr(buffer, "<h1>"))
      {
        printf("Extracting heading\n");
        fprintf(f, "Heading: %s", buffer);
      }
    }
  }
  fclose(f);
  fclose(readfile);
  return 0;
}
```

Desert Island Books

Paul Grenyer introduces Gail Ollis.



This is the first time I've struggled to find a story to write about the member featured in a Desert Island books edition. I've chatted with – and even met – Gail on a few occasions, for some reason my mind was completely blank (sorry Gail!). So I asked her if there was someone who could help me out and Gail responded with:

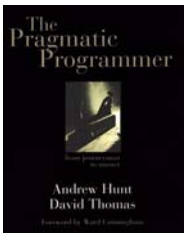
> Hmm, I'd say that you emailing me with a sexist joke on the basis that my ACCU-General postings seemed to imply a sense of humour gives us a bit of history! And I seem to recall that you were wearing the 'drunken, womanizing menace' T-shirt on the first occasion we met in real life. I think you were also one of the two people (Allan Kelly being the other) responsible for me presenting at conference for the first time – it hadn't crossed my mind until you both asked if I was thinking of doing it. I've no idea if it was prompted by something specific I'd said on the list or just the general air of brilliance in my postings that accompanied that of having a sense of humour!!

None of this of course sounds like me, so it couldn't possibly be true....oh alright then! It's all true.

Gail Ollis

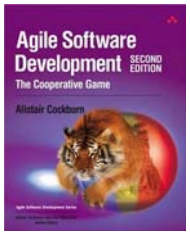
These days I study psychology instead of writing code – a consequence of having asked myself 'WHY did they do that?!' one time too many when faced with the decisions of my software colleagues. A desert island doesn't offer much opportunity for studying human behaviour (beyond finding out what happens to a garrulous, sociable personality suddenly subjected to complete isolation!) so it's really not a good time for me to become a castaway. On the other hand, perhaps it's a really good chance to review the literature that has been influential in my belief that there really is a better way.

Programmers' bibles

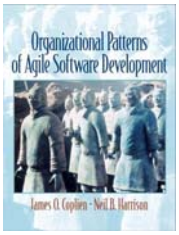


That puts *The Pragmatic Programmer* [1] at the very top of my list. Apart from anything else, it's an old friend which always seems to know what I think but expresses it much better. Contemplating his desert island, Allan Kelly mentioned his slow progress through *Gödel, Escher, Bach* (a book that's still on my 'to read' pile). On my desert island list, *The Pragmatic Programmer* is the book that took some time to read. This seems a little bizarre, since it's a wonderfully approachable book, but it took me two attempts to finish it. When I first tried, I was in the throes of a death march project that was getting things wrong on just about every principle described in the book. The material was all too easy to understand; every wise word was illustrated by a painful counter-example in my own working life. Perhaps its emotional impact is a useful tool for auditing – the number of times you put your head into your despairing hands is a pretty good guide to how far your organisation has to go! I fared much better on the second attempt and finished it feeling that every programmer should read it. There's plenty of quite specific good advice in it, but the philosophy behind all of it is epitomised by the book's first two tips: 'Care About Your Craft' and 'Think! About Your Work'.

My second choice is *Peopleware* [2]. Where *Pragmatic Programmer* explains principles that guide the everyday decisions programmers need to make, *Peopleware* discusses the environment in which they can do that to the best of their abilities. By way of example, here's a quick rant. I once worked for a company which insisted that external calls were answered within three rings (for all the talk of 'internal customers', apparently these didn't warrant the same treatment!) This mandate extended not just to the phone on your desk but any in the vicinity whose owner wasn't present. It was pretty frustrating to field calls without the information to do so usefully, but it was also frustrating for the poor callers; they'd have had a much better service from a prompt and well-informed response to a voicemail message. And instead of endlessly breaking precious flow, the company would have had a more productive workforce. As DeMarco and Lister put it, 'people who are charged with getting work done must have some peace and quiet to do it in'. Rant over; *Peopleware*, meanwhile, manages to put across this and many other insightful messages without ranting at all!



Because it's another rich source of sound advice, my next choice is *Agile Software Development* [3]. This book, like others with 'agile' in their title, is sometimes perceived as being applicable only to projects that are 'doing agile'. Having worked in an environment that ran for garlic and holy water whenever anyone mentioned the a-word, I can't comment on how helpful it might be for agile-specific issues. Nonetheless it contains plenty of ideas that apply more generally, inside agile projects or outside them. Take information radiators, which 'display information in a place where passersby can see it'. Or how about the idea that 'automated regression tests improve both the system design quality and the programmers' quality of life'? Sadly there are many workplaces where this is not perceived as a statement of the obvious. Like so much of this book, it's widely applicable and much-needed advice. If my desert island resources extend to pen & paper maybe I can devote some of my copious spare time to editing this book down to common principles so that we can persuade the most agile-sceptic audience to read and benefit from Cockburn's *Software Development (the expurgated version)*.

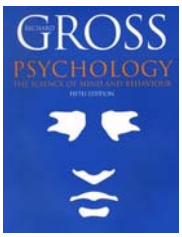


My final software book is another 'agile' one whose title risks alienating the audience that arguably needs it most. *Organizational Patterns of Agile Software Development* [4], like Cockburn's book, does indeed include some patterns that clearly fit in the agile mould, but also many more that could usefully be employed more widely. In fact this book is particularly good (and somewhat unusual) in helping readers identify measures to benefit them in their own, unique circumstances; every pattern begins with a statement of the context in which it applies. The book also counsels 'apply one pattern at a time, and if it doesn't feel right, back out'. It's therefore bound to bring me happy memories of my 'Santa Claus and Other Methodologies' presentation at ACCU 2008 (summarised in CVu 20.6); the contextual approach of the patterns fits right in with my crucial question – 'What problem are you trying to solve?' (which still holds the record for the biggest text I have ever used on a slide) – and the advice for applying them is far from a 'one size fits all' prescription.

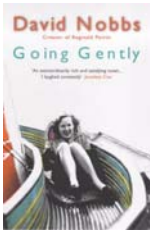
Other books

My own desire to solve problems has taken me from software development to studying psychology and its application to the issues of software

development. Being on the island is a good opportunity to continue my studies without distraction, and although it won't offer much scope for research ('Tell me, parrots, why did you use magic numbers in this squawking function?') there's plenty of time to think about how psychological principles relate to the problems described in my chosen books and formulate research proposals for when I am, I hope, eventually rescued. To do this I'll need psychology literature as well as the software development books so I'm going to have to cheat on the book allowance, but I'll settle for just one extra: *Psychology: The Science of Mind and Behaviour* [5]. It's a wonderful reference book, so comprehensive that I shall have to be very careful, in the absence of health care, not to drop it on my foot, and so readable that I won't regret the choice.



A psychological angle was, of course, present in my 2007 conference talk about what to expect when you try to introduce new ideas. I related the styles of reaction that you may encounter to characters from the Winnie-the-Pooh stories (most surprising finding: exuberant Tiggers can actually hamper your efforts), so perhaps my 'novel' should be one of those. They're rather thin books, though, and since I'm only allowed five I want to get my money's worth with weightier tomes. Being on the desert island will be something of an emotional rollercoaster, so it's fitting that my fiction book should be the only one that has made me cry AND laugh out loud. *Going Gently* [6] is the life story of Kate, who is 'enriched by the product of three fine sons, only one of whom was a murderer'. To say that Kate is an old woman on her death bed risks making the story sound maudlin or plain depressing, but it is neither. Its spirit is much more that of the aphorism one of my friends uses in signing off his email: 'Life should not be a journey to the grave with the intention of arriving safely in an attractive and well preserved body, but rather to skid in sideways, chocolate in one hand, body thoroughly used up, totally worn out and screaming 'WOO HOO what a ride!'' (source unknown). It's rare to find funny and poignant together, rarer still for it to succeed in both; this book achieves it memorably.



Music

What about music? There has to be music. I suppose the isolation may give me the perfect opportunity to practice *Bridge Over Troubled Water* and see if I can learn to sing it without squeaking on the top note, but I'd go mad if my own efforts were all the music I had.

Choosing just a couple of CDs seems more of an ordeal than being stranded on the desert island. If I had a collection with a dominant theme maybe I could pick some outstanding examples, but its one consistent feature is diversity, if that isn't a contradiction in terms. I was made acutely aware of this when an adult student in a basic IT class where I help out as a volunteer asked me what sort of music I like. 'Lots of different things', I said, and started listing a few that came to mind: 'Beatles, Eno, Gorillaz...' – his evident amusement (or was it bemusement?) at the last of these was quite entertaining, up to the moment he gave his response: 'it's unusual for someone your age to be interested in Gorillaz!' Wounding though the implication of being 'past it' might be, I was more struck by the word 'interested'. It seems to imply that I know something more about them, but in fact for most of the many artists in my collection I neither know nor care much about anything but the music. Among the rare exceptions are the Beatles (a teenage obsession – and since I am feeling a bit old after the Gorillaz incident I should add that the Beatles had long since disbanded when I 'discovered' them!) and Brian Eno (an interesting guy who could – and indeed did – hold his own on Question Time). These two are represented in the list of albums I return to again and again, which must be a good recommendation for them claiming their place in my tiny driftwood CD rack.

So *Sergeant Pepper's Lonely Hearts Club Band* has to go in. It's not necessarily because I think it's the Beatles' best – once I'd have given that title to *The Beatles*, commonly known as 'The White Album', without hesitation, and these days I wouldn't even consider 'what's their best album?' a meaningful question. *Sergeant Pepper* is included because it contains a lot of old friends I can sing along with. A few of the island's curious wildlife attending my lonely karaoke sessions on the beach are the only audience I'd ever countenance.

As for which Eno to take, I have a dilemma and may find that all my options get washed away by the waves before I have time to choose which to save. My first thought, before I even contemplated artists, was to wonder whether to choose something complex – plenty to keep it interesting over repeated listening – or something that blends in, as desirable and unremarked as the air we breathe and therefore not a choice I'd ever tire of. Brian Eno can provide either of these.

Music For Films has been part of my soundtrack since I first heard it at the age of eighteen. You've almost certainly heard some of it, whether or not you recognise the title; it's often used on TV. But just because it's normally in the background, don't make the mistake of thinking it's bland. It bears being in the foreground... but its hypnotic quality is such that listening to it attentively is much harder than slipping into reverie.

By complete contrast, add David Byrne to the picture and you get *My Life in the Bush of Ghosts*. This extraordinary synthesis of recorded speech and music almost defies explanation. Certainly I can't begin to do it justice, and my clumsy efforts risk making it sound unappealing. It is moving, sometimes disturbing music – and with so much 'going on' that it would help to keep my mind occupied. Lacking the musical vocabulary to describe it more fully gives me a good excuse simply to recommend that you hear it for yourself.

Perhaps I should start keeping them together in the same jewel case so that I can take both Eno albums, but if I get caught trying to cheat the system I will have to rescue *Music For Films*. There's enough in the books to occupy my mind, so I can afford to take the music that will soothe it. With no-one to talk to but the parrots, I think I'll need it.

What's it all about?

Desert Island Disks is one of BBC Radio 4's most popular and enduring programmes:

http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml

The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island.

I've been thinking for a while that it would be entertaining to get ACCU members to choose their Desert Island Books. The format will be slightly different from the Radio 4 show. Members will choose about 5 books, one of which must be a novel, and up to two albums. The programming books must have made a big impact on their programming life or be ones that they would take to a desert island. The inclusion of a novel and a couple of albums will also help us to learn a little more about the person. The ACCU has some amazing personalities and I'm sure we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

References

[1] Hunt, A. & Thomas, D. (1999) *The Pragmatic Programmer*, Addison-Wesley, ISBN 020161622X

[2] DeMarco, T. & Lister, T. (1999) *Peopleware: Productive Projects and Teams* (2nd ed.), Dorset House, ISBN 0-932633-43-9

[3] Cockburn, A. (2002) *Agile Software Development* (2nd ed.), Addison-Wesley, ISBN 0-321-48275-1

[4] Coplien, J. & Harrison, N. (2005) *Organizational Patterns of Agile Software Development*, Pearson Prentice Hall, ISBN 0-13-14670-9

[5] Gross, R. (2005) *Psychology: The Science of Mind and Behaviour* (5th ed.), Hodder Education, ISBN 978-0-340-90098-7

[6] Nobbs, D. (2001) *Going Gently*, Arrow Books, ISBN 978-0099414650.

Next issue: Anna-Jayne Metcalfe.

My 2009 ACCU Conference

Chris Oldwood shares his experiences.

This was my second ACCU conference – my first being 2008. Last year I was somewhat star struck as many of the authors of the books on my desk were there. Of course they were all very accessible, especially in the bar after hours. This year it didn't take nearly as long to slip into the routine of walking up to random people and chatting with them over 1 or 5 beers. I also vowed not to stay up until dawn every night and instead make every keynote – I almost succeeded. Another change for me was to focus less on the technology and more on the 'message', which as it turned out was pretty hard to miss in some cases...

I got into the swing of things pretty quickly by arriving on the Monday night and taking the pre-conference session on Tuesday by Alisdair Meredith about C++ 0X. I have been loosely following the proposals for the next C++ standard on various blogs, but it still felt good to have a leading figure talk you through the proposals, and how they interact, such as R-Value references and move semantics. As expected he covered the new Memory Model and threading features in reasonable depth, illustrating how the Double-Checked Lock Pattern would be implemented correctly. The middle part of the day covered some of the more settled features like `unique_ptr`, initialiser lists, variadic macros and templates and then he devoted a significant amount of time towards the end to Concepts – the other big new feature. I was doing well, but I'm afraid he lost me when it got to discussing Axioms, luckily there's still plenty of time to digest all this before it becomes a reality.

What were they fighting for? Tabs vs. Spaces? Vi vs. Emacs? Honour?

Wednesday was the start of the conference proper and the opening keynote was from Robert 'Uncle Bob' Martin about Software Craftsmanship. This was an interesting and insightful piece about how we have attempted to cast ourselves as scientists or engineers or architects and consequently have ended up drawing some pretty poor analogies that has helped neither us nor our customers. This also laid a nice backdrop for Nicolai's keynote which was due at the end the day.

F# is a new language joining the .Net family and one whose blog I have followed out of curiosity, so I chose Oliver Sturm's 'Functional Programming in F#' as my first session. After double-checking with him first about how 'heavy' it was going to be, I was pleased to see that he covered much of the language's background to set a context for its use. Using the interactive interpreter he walked us through some F# basics, but its foibles as a statically-typed .Net language looked apparent to those that were perhaps used to other more flexible FP languages, so he didn't get time to cover some of the more advanced features.

I followed lunch with Roger Orr talking about Refactoring in the Real World. The 'RW' suffix was important as this hugely over-subscribed session took on a workshop style as the attendees were all too keen to share their experiences. Roger used rock climbing safety, i.e. only moving one limb at a time as his analogy to drive home the importance of small steps and the use of version control. As someone just starting to do this formally it was useful to see how to avoid refactoring everything in sight when you come to an old codebase that has seen a lot of action.

CHRIS OLDWOOD

Chris started out as a bedroom coder in the 80s, writing assember on 8-bit micros. These days it's C++ on Windows in big plush corporate offices. He is also the commentator for the Godmanchester Gala Day Duck Race.

Michael Feathers and Steve Freeman filled my afternoon with a workshop about Programming Paradigms. They took a simple problem and got four teams of people to solve it using different programming paradigms – Procedural, OO, Functional and Rules Based. The teams would then rotate and implement a new feature using a different paradigm. The paradigms were compared and Steve showed various implementations using Java to illustrate how a multi-paradigm language could be used. This session was also hugely popular and even though I was only a bystander I still found it thought provoking.

This year saw the day finishing with a keynote as well, and it was Nicolai Josuttis who delivered a scathing comment on the lack of quality being applied in the industry due to the marketing drones calling the shots. The large dose of irony that accompanied his examples did not cloud the underlying message which is perhaps more apparent in large corporations than smaller software houses. It certainly generated a fair bit of discussion in the bar later.

Thursday's keynote was delivered by Linda Rising on the level of abstraction required to make patterns work successfully. She pointed out the dangers of being too specific to one problem domain – namely that other people will fail to see the similarities and discard them or create a new, but very similar pattern, when in reality they should have created a higher-order pattern that encompasses both. I found her account of the POLP conferences shepherding process and writers' workshop, and how that differs from the usual conference format, particularly interesting.

I decided to go with Phil Nash's introduction to Objective-C after the coffee break as I had seen snippets of example Objective-C code in magazines and thought the syntax was weird, but as Phil quickly showed (after a brief OO language history lesson), it isn't really that strange and I was quite taken with many aspects of it until he started to describe the reference counting. Unfortunately time got the better of him, but nonetheless he appeared to cover most of the key language features, and also its reliance on conventions, which appears to be a big aspect. It was a good introduction that has certainly grabbed my attention.

Lunch was again spent outside enjoying the sun. It almost seemed a shame to have to come back inside, but I did for Hubert Matthews talk on Modelling Archetypes. This was, much to my surprise, probably the most enjoyable talk I went to because Hubert managed to bring into a focus a number of questions I had floating around my head about lightweight modelling in general. He introduced a number of entity types to represent certain key concepts like transactions and roles and then proceeded to give them a colour. He showed some example models and went on to explain how the various entities naturally fit together, with the colours making it much easier to see the structure of the model, and more importantly, to see potential problems.

A short coffee break and it was back for more C++, with a session from Andrei Alexandrescu about replacing the Iterator concept with Ranges. His talks are always entertaining as he tries to get a little audience participation going and you know that he's going to hit you with something cool! He didn't disappoint as he showed that the iterator concept in C++ has limitations that make implementing certain algorithms or containers difficult or impossible. His solution was to replace the discrete pair of begin/end iterators with a Range concept, which not only encapsulated that behaviour but also vastly simplifies others such as reverse iterators. Once again Andrei makes everything look so simple and leaves you wondering why no one thought of it before.

Another new addition for ACCU 2009 was in store after the sessions – a series of Lightning Talks. These were short (5 minute) presentations from anyone about anything. A fair selection of topics were covered in the first 9 talks, such as: Seb Rose giving a quick description of the Rational Jazz Server, Didier Verna comparing the finer points of Lisp, Jazz and Aikido and Mark Bartosik discussing debugging Windows applications.

Thursday night's drinking had an extra twist this year as Steve Love and Richard Harris decided to do a spot of fencing in the hotel car park! They had proper kit and everything – not just a couple of broom handles and duffle coats. But, what were they fighting for? Tabs vs. Spaces? Vi vs. Emacs? Honour?

Due to an over-indulgence of Leffe I managed to miss Friday's keynote and instead started the day with a session on Portable Code from Steve Love. He took a much wider look at what it means to write 'portable code' by starting with the more obvious issues around OS's and tool-chains, but then introducing Testing as another 'platform' along with external services and even developers' differing abilities. Steve's message about how hard it is to write truly portable code was pretty clear, but by following industry best practices where possible, we can make our code malleable enough to easily cope with whatever 'platform' we need to write for.

Another pleasant lunch outside in the sun was followed by the second talk from Andrei Alexandrescu – this time about Cranking Policies Up. This was based on a Scott Meyers article, called 'Enforcing Code Feature Requirements in C++', that heavily used Template Meta-programming to attribute features to code, such as 'thread-safe' or 'portable' so that feature incompatible code could be flagged at compile time. Andrei's focus was on showing why the C++ language, as it stands today, does not make the technique scalable. It also gave him the chance to give the D language another plug, by pointing out where it was superior.

My last session for the day was from John Lakos about Designing Good Components, Interfaces and Contracts. In true Lakos fashion, he had some 470 slides (less than last year though!) to cover in just 90 minutes. He gave a whirlwind tour of Physical Design – as defined in his book – and then went on to discuss contracts, in particular how and when to define behaviour and when to leave behaviour undefined, perhaps using defensive programming to highlight when UB has been invoked. Unfortunately, given the ambitious size of the talk it overran and he had little time to spend on the final section about the different forms of inheritance.

I just managed to squeeze a quick pint in before the second set of Lightning Talks. Once again we had another 9 presentations, this time kicked off by 'Caring Will Only Cause You Pain' by Peter Hammond which beautifully continued the theme Nicolai addressed days ago in the evening keynote about the death of quality. A bit later Olve Maudal continued the onslaught by using the analogy that you wouldn't want to work in a messy kitchen, so why do it with your codebase? And later still Mark Dalgarno also tackled Nicolai's topic of cut-n-paste coding by discussing Code Clones. In between were other talks about Git, Scala, crime, character encodings and the Google web toolkit.

The Conference Dinner that evening was another good opportunity to continue stepping outside my 'comfort zone' and just talk to random people. The principle is that the speaker gets to stay put and the attendees switch tables after each course, and with Giovanni controlling proceedings, no one is going to argue when it comes time to switch tables. I managed to speak to a whole bunch of new people and talk to some speakers whose talks I missed because of the annoying physical limitation of only being in one place at a time.

Allan Kelly kicked off the final day with a keynote that looked into the management of software development. Using Fred Brooks' comment on the quality of people being more important, he introduced 9 principles (although it was 10 really as it was 0-based!) for better management. These principles tackled balancing workload, relationships, quality and risk which hopefully will lead to both happy customers and developers. Although aimed more towards managers, he did pronounce that 'Managers are not Aliens' and try to get developers to be more aware of their managers' needs. He also managed to finish with the most blatant book plug of the conference (I think it was called *Changing Software Development: Learning to become Agile*).

After copious amounts of coffee to aid recovery from the night before I decided to join Klaus Marquardt's workshop called 'Patterns for Versions, Releases and Compatibility'. Whilst giving his introduction he produced a large ball of wool which seemed a little odd, until we found out that we were going to discuss and list the various aspects of our current project's release cycle, attaching them to the wool to create a washing line, of sorts, that describes the process. We then got to go around the room and see how other people in other domains tackle releases and put love-hearts or lightning bolts on any practices we particularly liked or loathed. There was a good cross section of application domains to ensure this session had the diversity it needed.

My final session of the conference was spent with the bare-footed Pete Goodliffe discussing Legacy Code and how we can learn to live with it. This was a talk about what legacy code is, how do we go about understanding what it does and then how can we change it without breaking anything else and hopefully improving it at the same time. The majority of the talk was about the 'understanding' aspect as he initially covered the wider picture such as the build process, documentation, customers, testing etc to show that the context of the code is often bigger than just the raw lines of source code. Pete then focused on how to 'map' the software to get a feel for its structure, by looking at interfaces, dependencies, file layout etc. The final section on modifying the code concentrated on testing practices, but also touched on other improvements such as cleaning up old comments. He finished by walking through a code change with the aid of frogs! I found Pete's session nicely put some of the recent books I'd read (such as Michael Feathers) into a bigger context.

The sessions – the bit you pay for – are worth every penny in their own right, but they're not the only reason for coming to the ACCU conference. The other reason is networking, or more simply, meeting other likeminded individuals. Every lunch time and evening in the bar once again proved a goldmine as I met new people and got to talk to a few of the speakers in a more informal setting. The great thing is that you know everyone in that bar (well, those under 90 at any rate) is there for the same reason – drinking beer and talking about programming stuff and probably even the real world every now and then. I know it's only my second year, but it lost none of the magic I experienced last year.

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-glamourous 'not recommended' rating, you are entitled to another book completely free.

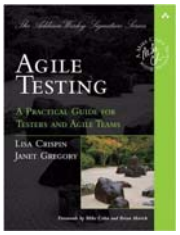
I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Agile Testing: A Practical Guide for Testers and Agile Teams

By Lisa Crispin and Janet Gregory, published by Addison Wesley, ISBN: 978-0321534460, 576 pages

Reviewed by Paul Grenyer



Recommended

This book is pretty much what it says on the tin and that's a good thing. Behind all the usual Shiny Happy People Having Fun stuff you usually get in books from the Agile community is some sound, well expressed advice. This isn't just a book for Agile testers. There's a lot of good practical information that all testers should learn. It's a difficult balance to achieve, but I think the use of the word Agile may put off a lot of people who should really be reading this book.

As well as the general practical testing advice the book also covers a lot of fundamental Agile stuff. It sets out some Agile testing principals and discusses the problems a lot of teams have when transitioning to Agile. It's all been written before, but never from a testers perspective, but to be honest it's not that different from the developer perspective.

The book is very hung up on the idea that developers in Agile teams, and indeed testers in or joining Agile teams have difficulty seeing how testers fit in as developers are doing unit testing and therefore the code is supposedly already tested. Personally I feel that anyone who doesn't understand that not everything can be unit tested and see that 'independent' testers are vital is probably in the wrong job. The book made me view our own tester, who does not come from a programming background, in a totally different way. Instead of seeing him as someone who just carries out the manual user interface tests, I now see him as an integrated member of the team who needs to take part when requirements are gathered and should also be writing integration and end-to-end tests as well as maintaining continuous integration. The necessary training has now commenced.

Most of the Agile discussion is at the beginning of the book. The practical stuff comes later and is quite detailed, including most of the sorts of testing, including automated GUI testing, that should be carried out.

I think this book will make most people think differently about testing in a good way. Recommended.

The CERT C Secure Coding Standard

By Robert C Seacord, published by Addison Wesley, ISBN: 978-0321563217, 720 pages

Reviewed by Balog Pál

Recommended, if ...

There is another review of this book, written by Derek M Jones, in the May 2009 CVu. If you consider to buy or read this book, please read that review first. Derek classified the book as 'Not recommended', with fair explanation. And the facts he list are true.

Yet, the interpretation of the facts can be different. The book is on a very serious topic – problems that cause software problems, especially 'security vulnerabilities', that manifest in the world in spreading malware, data loss, data theft, with the monetary impact measured in $ billions yearly. The rules and guidelines in the book, if put into practice could decrease these kind of defects much. While indeed many guidelines can be called 'platitudes', still they are not followed, and in 2009 the software is still full of those banal problems. Among them, those pesky format strings still create 'hack-me' engines, despite the many texts about them and specific warning built in the compiler...

This book is not very good for you, if:

- You are on starter level, or did not read some other morality guides and guideline books. You need a fair amount of background, or to follow all the listed references.
- You work on a system that is not (mostly) written in C99. If your system is C++, many of the listed problems apply, but the solutions are very different. Wait for the C++ version, that is under way; you can use the material on the web. If your system is C90, you'll need adjustments (or follow the suggestion to go C99).
- - You're short on budget. The material is on the web for free access, you can buy books not available.

This book is good for you if you have a C project with problems, and you're responsible for making it better. Especially if you have to create or manage the guidelines, the process, allocate resources. It can save you very much time, providing much preprocessed material. I very much liked the idea that every item has an ID, and the exceptions too. When put to work, it can streamline reviews and corrections, just using those. Also the calculating of 'priority' and 'level' that includes the estimated cost of cleanup.

For each item there is also a good list of references, covering both theory and the actual vulnerabilities encountered in the world. Derek noticed the severity and impact values are too subjective. As I see this is not subjective, but the source of the book – it was created mostly from 'security vulnerabilities' observed. That is certainly a subset of all possible problems. And each real system has its own environment.

If the guidelines are put to use, that should not be a problem, as the adopter can substitute the

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Holborn Books Ltd** (020 7831 0022) www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792) blackwells.extra@blackwell.co.uk

relevant values, along with selecting items from this book and elsewhere.

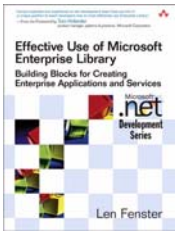
The book is also a good read for seniors who know most of the problems, but can find a couple they were no aware of yet. And possibly find some manifestations in the codebase. You can do it on the web too. That is well maintained, so if you spot a problem, just leave a comment, it may trigger update overnight.

So as summary this book is not perfect, is not a silver bullet, also will not replace knowledge and thinking of good engineers – but can be used to good effect if one wishes, and nets a fair value.

Effective Use of Microsoft Enterprise Library – Building Blocks for Creating Enterprise Applications and Services

By Len Fenster, published by Pearson Education, ISBN: 978-0321334213, 736 pages

Reviewed by Omar Bashir



Not recommended

This is one of the two books in the market explaining in detail the functionality provided by and the use of Microsoft Enterprise Library. Even though this book was published in 2006, it was already significantly outdated then as it only targets the Enterprise Library version for the .Net Framework 1.1 whereas the .Net Framework 2.0 and a corresponding version of the Enterprise Library had already been released. The latest version of the .Net Framework is 3.5 which is being widely used in production environments. The latest version of the Library (4.1) was released in October 2008 containing more application blocks than those described in this book. Although, the book still covers certain application blocks that provide considerable functionality in many enterprise applications and services, but as discussed later, there have been significant modifications in those application blocks since the release of the .Net Framework 2.0.

Application blocks are resuable, extensible and configurable source code components that have been developed and released by Microsoft's Patterns and Practices Team. These application blocks provide considerable functionality in certain common aspects of applications like configuration management, cache management, data access, exception handling, logging and security. These application blocks can be extended to alter or enhance the functionality they provide to fulfil specific requirements of various applications. Furthermore, new application blocks can be developed and included to provide functionality that the Enterprise Library does not provide by default.

The book has an elaborate preface but does not have an introduction chapter. This can be considered as a drawback as readers may generally omit or only casually read through the preface. Parts of the preface could have been used as contents of an introduction chapter which may have also explained the procedure to install the library as the library is not bundled with any of Microsoft's development tools. In addition, an introduction chapter may have introduced various tools that assist in developing software using this Library (e.g., the configuration management tool).

The book contains examples in both C# and VB.Net. The author starts demonstrating the use of the Library quite early in the first chapter. However, it was a bit annoying to try these examples directly from the book as they do not include the specification of the namespaces of various classes being used. References to be included in projects using various Enterprise Library application blocks were also not provided. Other information that can be very useful to developers but was not provided include the assemblies that contain the functionality of corresponding application blocks. All this had to be ascertained experimentally with some inconvenience.

The author seems keener to explain ways to extend the Enterprise Library than to use the existing functionality. Readers generally have to go a considerable distance within chapters before being able to write simple programs to exercise the existing functionality of these application blocks. As mentioned above, examples are snippets rather than complete programs requiring considerable effort on the part of novice developers to get them working.

The final chapter of the book attempts to explain development of new application blocks. Initial few sections of this chapter explain core block functions, pluggable providers and the use of factories in decoupling the two within an application block. The chapter demonstrates the development of an application block (Data Mapper) that is not provided with the Enterprise Library. However, the author seems to have rushed through the example resulting in the explanation of the implementation not being very clear.

The book has three appendices. Appendix A explains the use of the Data Mapper Application Block described in the final chapter in the manner similar to other application blocks explained earlier. Appendix B provides information on creating a .Net managed data provider to retrieve and store data in XML files. Appendix C explains changes between the version of Enterprise Library for .Net Framework 1.1 and that for .Net Framework 2.0. Configuration Application Block has been deprecated as similar features are provided in the System.Configuration namespace of the .Net Framework. Configuration helper classes have been added to facilitate working with the System.Configuration namespace. Considerable enhancements have been made to instrumentation. Logging Application Block has been simplified and takes advantage of the functionality provided by System.Diagnostics. Similarly the Data Access Application Block has also been simplified and is better integrated with ADO.Net. Finally, with the Security Application Block, authentication, role management and profile management have been deprecated as they are part of the .Net Framework.

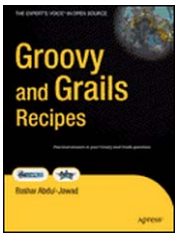
The author has a clear writing style and provides good explanation of various concepts without being too verbose. The book appears to have been targeted primarily at users with some experience of the previous version of the Library as, in addition to omitting necessary information mentioned above, it explains in detail the enhancements in application blocks from their previous versions. For graphical representation, the author primarily relies on class diagrams representing only the static structure of the software. There are very few graphical architectural descriptions and hardly any sequence or interaction diagrams specifying the dynamics of the software used in the examples. A few of the latter could have significantly enhanced the understanding of the software discussed by the author.

Because Microsoft has significantly enhanced the Enterprise Library since this book was published, the contents of the book are heavily outdated and, therefore, it is not recommended. Developers intending to use the Enterprise Library can get useful and up to date information with examples from the documentation that is installed with the installation of Library.

Groovy and Grails Recipies

By Bashar Abdul-Jawad, published by Apress, ISBN: 978-1-4302-1600-1, 297 pages

Reviewed by Alan Griffiths



Groovy is one of a number of 'dynamic languages' that have become popular recently. Groovy is reminiscent of Smalltalk in that the object model allows object types to be modified as the program runs and its use of closures. Its other distinguishing feature is a flexible mapping into the Java JVM that allows existing Java libraries to be accessed via several convenient syntaxes. Java code can also invoke Groovy allowing an application to use the more suitable language for various parts of a project.

Grails is, as the name suggests, a web framework inspired by Ruby-on-Rails. I've not used either of these for significant projects, but they both appear to address a lot of the tedium of managing user access and session control that I experienced using JSP nearly a decade ago (so some progress is being made in the industry).

This book doesn't take the usual didactic approach to teaching a programming language. It assumes a level of competence from the reader and comprises an enormous number of worked examples or 'recipes'. These focus on showing how accessing various technologies can be accessed easily from a Groovy program. The Grails framework is the technology treated in

View From The Chair

Jez Higgins
chair@accu.org

My thanks to those of you able to attend this year's AGM. On behalf of the officers and committee I would like to thank you for electing or reelecting us, and for the confidence you place in us. One significant change this year say Allan Kelly standing down as Publications Officer, with Roger Orr being elected to the post. In the past, Allan has said the role of Publications Officer involved long periods of nothing punctuated by the occasional crisis. He may have changed his opinion over the last year, as while searching for a new editor for CVu we also changed the publications schedule as well. Allan's put in a great deal of work for ACCU over the past several years and particularly in the last year, and I'd like to thank him that and for the help he's given me as Chair. Roger is, of course, a long standing member of ACCU, regular conference speaker, and the man behind compiling the Code Critique. I'm glad to have him on board.

The AGM takes place, of course, during the ACCU Conference. The year's conference was, again, a great success. At a time when commercial conferences seem to be scaling back or even closing up completely, the ACCU Conference continues to be strong and successful. Last autumn, the conference committee was buried in a virtual avalanche of submissions. Those submissions translated in a strong conference programme, that attracted a more-or-less sell-out crowd. The conference committee made some innovations in the programme, introducing the lightening talks for example, and, in Susan Greenfield, bringing in a speaker from well outside what might be considered our normal orbit. Risky perhaps, but what a reward. Giovanni Asproni, the conference chair, and his committee of Astrid Byro, Francis Glassborow, Alan Lenton, Ewan Milne, Roger Orr, Tim Penhey, and James Slaughter deserve hearty congratulations for a terrific job. Many thanks, chaps. I'd say 'Now get on with next year's', except that I know discussions are already under way, with additional plans being laid for a one day conference this autumn. Keep an eye on accu-general and the website for news on that. I'm looking forward to it already.

Membership

Mick Brooks
accumembership@accu.org

Since I prepared the AGM reports we've welcomed 25 new members to ACCU, while 20 have left, for an overall increase of 5 leaving a paying membership of 763. I suspect this churn is representative for this time of year, thanks to new memberships for late conference bookings and the expiry of memberships for those that joined for the conference last year but choose not to renew.

It came up in discussion at the AGM that, although we do have an ever increasing number of members from around the world, there's a large core based in and around a few cities: Oxford, Cambridge, London. I think this clustering tells, amongst other things, of how the ACCU membership has grown: through the personal contacts of existing members. Feedback suggests it's not necessarily the direct approach ('Why don't you join ACCU?') that works best, but simply using ACCU to help out where it's useful. If someone's looking for a book on a subject, point them at a review or two on the website. Keep an ear out for what your colleagues are working on – is there a journal article that could be relevant?

As ever, send any questions or suggestions about membership and renewals to me at accumembership@accu.org.

Publicity

David Carter-Hitchin
ads@accu.org

Membership numbers are dropping and we need to do something about it, and urgently. For this reason I have stepped up efforts to get leaflets out to all libraries in the country, wherever possible. These can be displayed on notice boards, be put into the societies and clubs listings and also placed at reception for people to take away. The time required to help with this project is very small: You go to your library and see out many leaflets are required for the whole county (most libraries offer a scheme to distribute the leaflets for you), tell me the number, receive leaflets, give leaflets to library. Easy. If you haven't already done so, please send me an e-mail at publicity@accu.org if you are able to help.

Bookcase (continued)

greatest depth – the Grails related recipes take up most of the second half of the book.

Taken in large doses the persistent claims that 'Groovy is great because you can do ***this*** like ***this***' can be a bit wearing, but that is less of an issue if the book is used as a reference.

There are a lot of tasks for which Groovy is a suitable language – allowing a lot of programmer productivity and this is clearly demonstrated by some comparisons with Java. The recipes in this book provide an easy source of examples for using a wide range of technologies. (This does require the reader to have the knowledge to decide what to use and how it fits into the application.)