The magazine of the ACCU

www.accu.org

{cvu}

Volume 20 Issue 5 November 2008 £3

Features

{cvu}

Volume 20 Issue 5
November 2008
ISSN 1354-3164
www.accu.org

Editor
Tim Penhey
cvu@accu.org

Guest Editor
Gail Ollis
accu@ollis.org.uk

Contributors
Nick Efford, Robert Finking, Stuart Golodetz, Pete Goodliffe, Paul Grenyer, Thomas Guest, Tony Jenkins, Steve Love, Anna-Jayne Metcalfe, Roger Orr, Peter Pilgrim, Matthew Wilson

ACCU Chair
Jez Higgins
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Tell me about it

I have a confession to make. I rarely read an issue of CVu from cover to cover – this issue, of course, being one of the exceptions! Not that there's anything wrong with the articles I skip over; far from it. They're accomplished and detailed pieces that really help with practical tasks, offering the kind of knowledge sharing that helps to make ACCU the supportive community that it is. So why do I sometimes skip them? Precisely because they contain detailed instructions; if an article explains how to do something I'm not likely to find myself doing, I'm not the audience for it. But that doesn't stop me from being curious. I'd still like to know the 'what' and 'why' of what others are doing, even if I'm never going to need the 'how' myself.

So when I got the chance to guest edit an issue of CVu and pick a theme for it, I thought it would be nice to pay particular attention to those questions of 'what?' and 'why?' and perhaps also get a glimpse into some areas which many of us don't encounter in our daily routine. I've asked writers to take a step back and look at their knowledge with an outsider's eye – no easy task, but they have risen to the challenge marvellously to address my rather philosophical questions.

We have articles addressing both the technical and human sides of code analysis; a chance to peek into the worlds of teaching and research; tasters of Gant and Groovy; a considered look at choosing the right C/C++ feature for the job; and a bird's eye view of virtualization that might just change your mind if you didn't think this was a technique for you. I hope you will find some interesting articles that would normally fall outside your radar, and I suspect you'll also find something new in the 'bigger picture' perspective on topics you're already familiar with.

I'm handing over to the next guest editor, Faye Williams, for the January issue. I'd like to wish her luck, and also thank Tim Penhey on your behalf for all his hard work as CVu editor over the last couple of years.

Gail

GAIL OLLIS
GUEST 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

FEATURES

SUBMISSION DATES

C Vu 21.1: 1st December 2008
C Vu 21.2: 1st February 2009

Overload 89: 1st January 2009
Overload 90: 1st March 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

Taming the Lint Monster (Part 1)

Anna-Jayne Metcalfe gives a personal perspective of the PC-Lint code analysis tool.

An all too common story

It's such a common story. Partway through a project, the company starts to become anxious about the number of defects that are being identified in the product, and how long they are taking to fix. Even worse, *customers are beginning to notice.*

Something must be done. Additional resources are thrown at the problem, but somehow it doesn't ever seem to be enough. The codebase is large, complex and hard to understand and maintain. It is – for all intents and purposes – a 'Big Ball of Mud' (not that anyone in the company would know such a term; after all – they are far too busy firefighting to read tech blogs and keep up to date with current trends in software development).

After several months of throwing additional firefighters at the problem, someone has the bright idea™ to find out just how much hidden nastiness is lurking in the code base waiting for the right moment to let loose its wrath on the unsuspecting team.

An appropriate tool is identified and procured, and then the real fun starts – actually *using* it.

Inevitably, it never quite turns out the way the team (or their managers) expect. Not only does it turn out to be an absolute nightmare to configure and use (after all you never appreciate how much work compiler project files can save you from until you have to maintain something comparable yourself), but when the team do finally get it working to their satisfaction the results it produces are so volumous that nobody quite knows what to do with them. Worse, they contain some **really bad news™**.

As all too often happens, dealing with the issues the tool raises is deemed to be a) too expensive, b) too risky and c) not as much fun as writing new copy-paste code (though nobody is ever quite honest enough to admit to the latter).

The team conveniently forget about the whole experience and go back to compiling at warning level 3 as they always have done. The installation disk for the offending tool is quietly hidden away in a desk drawer and forgotten...and of course, the Big Ball of Mud grows ever bigger until the inevitable 'let's just re-write it in language X' event a year or two later (Figure 1). With an eye on what language 'X' would look like on everyone's CV, of course...

But it sure did seem like a good idea at the time.

Figure 1

Meeting PC-Lint in a dark alley

My personal experience of PC-Lint started around 1996. At the time, I was leading a small team (actually, there were three of us...) developing virtual instrument software for a Windows NT 3.51 based automatic test system to support a military aircraft. My team was just one of several on the project, and although we had a good team we were badly under-resourced and constantly on edge – partly because of (seemingly politically motivated) interference by our prime contractor.

It was challenging, but highly stressful work (our Software Manager leaving the project due to stress was testament enough to that). Nevertheless, we were making good progress – our code was working and we were back on schedule (or at least we were after re-estimating everything from the ground up with three times the original estimate...). On the surface, the worst was behind us, and our system was actually starting to look pretty good.

It was at that point that someone had the bright idea of applying code analysis tools to the codebase to see if any potential nasties were lurking within. A demonstration by McCabe was arranged, and although it looked very impressive – especially on the (as it seemed at the time) huge monitor they brought (ours were tiny 14" things then) – but the price was well beyond our budget – especially at such a late stage of the project.

Enter Gimpel PC-Lint [1], a C/C++ static code analysis tool published by Gimpel Software (then at version 6, if I remember rightly). This was a tool I was not aware of at the time but which my co-team leader recommended.

When it arrived we had of course to learn how to use it. At the time we were using a mixture of Visual C++ 1.52 and 4.2, so the 'integration' consisted of adding a custom tool to run an analysis on a single file and display the results in the output window. Rather amazingly, that is still the way it is usually done today.

Needless to say the results seemed cryptic and verbose, so we did what most teams in this situation do – turn off most of the PC-Lint messages, leaving only those which we thought might indicate a serious problem. That is an entirely reasonable approach, but it does of course carry the risk that in doing so you may inadvertently mask something very significant.

The next thing that happened was that our code review policy was amended to include a requirement that each lint issue remaining in the codebase be justified by the developer responsible for it. Suddenly code reviews became more of a challenge, so in that respect the process worked well. We did, however only use a very small subset of PC-Lint's capability – and the warning policy remained (to my mind, looking back) far too lax for long term use.

ANNA-JAYNE METCALFE

Anna hasn't always written software for a living, but saw the light and defected to software development after several years writing 'Rusty Washer Reports' in the defence sector.
She has a taste for Belgian beer (hint, hint!) and may be contacted at anna@riverblade.co.uk.

Listing 1

```
// Standard PC-Lint configuration options for Visual Studio 2005
// Generated by Visual Lint version 1.5.9.79 at Tuesday, April 08, 2008 16:40:13
au-sm123.lnt   // Effective C++ 3rd Edition policy
co-msc80.lnt   // Visual Studio 2005 compiler definitions
lib-mfc.lnt    // MFC library definitions
lib-stl.lnt    // Standard Template library definitions
lib-w32.lnt    // Win32 API definitions
lib-wnt.lnt    // Windows NT API definitions
lib-atl.lnt    // ATL definitions
riverblade.lnt // Other library tuning
options.lnt    // Warning policy
-si4 -sp4      // Integers and pointers are 4 bytes

// Include definitions for platform 'Win32'
-i"C:\Program Files\Microsoft SDKs\Windows\v6.0\Include"
-i"C:\Program Files\Microsoft SDKs\Windows\v6.0\Include\gl"
-i"C:\Program Files\Microsoft Visual Studio 8\VC\include"
-i"C:\Program Files\Microsoft Visual Studio 8\VC\atlmfc\include"
-i"C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\include"
```

Listing 2

```
-e27   // Avoid "Illegal character" errors on VS2005 .tlh and .tli files)
-e537  // (Warning -- Repeated include file)
-e655  // (Warning -- bit-wise operation uses (compatible) enum's)
-e730  // (Info -- Boolean argument to function)

-e783  // (Info -- Line does not end with new-line)
-e944  // (Note -- Left / Right argument for operator always evaluates to False)
-e1550 // (Warning -- exception thrown by function is not on throw-list of function)
-e1774 // (Info -- Could use dynamic_cast to downcast ptr to polymorphic type)
-e1904 // (Note -- Old-style C comment -- Effective C++ #4)
```

Listing 3

```
--- Module: CJFlatHeaderCtrl.cpp
}
CJFlatHeaderCtrl.cpp(160): error 1401:
       (Warning -- member 'CCJFlatHeaderCtrl::m_bSortAsc'
(line 146, file ..\Include\CJFlatHeaderCtrl.h) not initialized by constructor)

}
CJFlatHeaderCtrl.cpp(166): error 1740:
      (Info -- pointer member'CCJFlatHeaderCtrl::m_pParentWnd'
(line 150, file ..\Include\CJFlatHeaderCtrl.h)
      not directly freed or zero'ed by destructor

-- Effective C++ #6)

--- Global Wrap-up
error 900: (Note -- Successful completion, 2 messages produced)
```

Up close and personal with PC-Lint

PC-Lint and its Unix/Linux cousin Flexelint are among a number of other similar tools on the market (e.g. PreFAST, Klocwork Insight, Parasoft and QA C++), but most are part of a much larger integrated toolset with an enterprise price tag to match. The only open source contender I'm aware of (Splint [2]) is limited to C and its authors unfortunately seem to have no intention of adding C++ support (however, as the source code is freely available, I will happily leave that as an exercise for the reader...).

PC-Lint is a command line tool with a range of options rivalling those of a full featured C++ compiler. All of that configurability comes at a price of course – complexity and all it entails. It is far from easy to use, and the analysis results it produces can also be verbose and cryptic to say the least. Nevertheless, PC-Lint is *very* thorough, and more than capable of exposing potentially serious hidden flaws in your codebase.

At its absolute simplest, a PC-Lint command line to analyse a single file and output the results to the console looks something like this:

```
lint-nt std.lnt <filename>
```

where `lint-nt.exe` is the PC-Lint executable and `std.lnt` is a configuration file (Gimpel call them 'indirect files') describing the compiler and framework configuration (preprocessor symbols etc.), include paths and warning policy.

`std.lnt` usually consists of a set of references to other indirect files, together with a handful of options and a set of include folder specifications, (Listing 1).

Of particular note is `options.lnt`, which usually defines the warning policy, together with additional issue suppression options. Our own warning policy is actually pretty simple – it consists of the full set of Scott Meyers' recommendations (activated by the inclusion of `au-sm123.lnt` in `std.lnt`), with a handful of issues suppressed by **`-e`** directives (Listing 2).

Such an aggressive warning policy is however not well suited for use with a codebase which has not been analysed before. In such cases, replacing `au-sm123.lnt` with a less stringent alternative is often advisable until the major issues in the codebase have been dealt with.

If you use Microsoft Visual C++ you can download sample `std.lnt` and `options.lnt` files for all Visual Studio versions from Visual C++ 6.0 to Visual Studio 2008 and eMbedded Visual C++ 4.0 from the Riverblade website. [3]

When PC-Lint is run on a source file, the results are by default presented in textual form, as Listing 3shows.

Message formatting and presentation

The format of the analysis results can be configured using a suitable indirect file. With the appropriate configuration for a given development environment, most development environments can understand enough to provide 'double click to go to issue location' functionality for analysis results piped to its output window.

Gimpel provide a number of such environment options files in the PC-Lint installation (for example `env-vc8.lnt` for Visual Studio 2005), and others can be downloaded from the support page at http://www.gimpel.com/html/ptch80.htm (PC-Lint 8.00) or http://www.gimpel.com/html/ptch90.htm (PC-Lint 9.00).

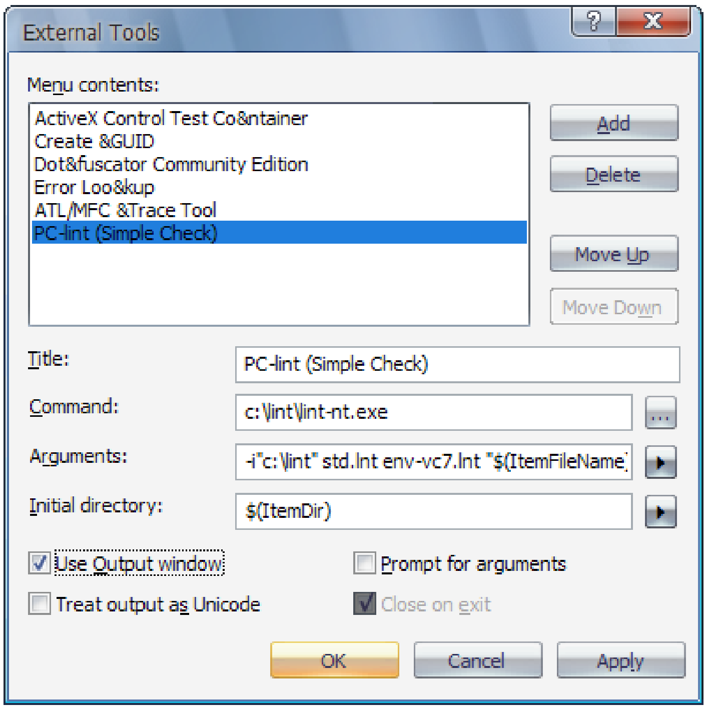
A special mention must be made of `env-xml.lnt`, which allows PC-Lint to easily generate XML (Listing 4).

If you intend to post-process the raw analysis results, the usefulness of this should be obvious!

Listing 4

```
<?xml version="1.0" ?>
<doc>
  <message>
    <file>fileb.cpp</file>
    <line>2</line> <type>Info</type>
    <code>753</code>
    <desc>local class 'X' (line 2, file fileb.cpp)
       not referenced</desc>
  </message>
  <message>
    <file>fileb.cpp</file>
    <line>4</line>
    <type>Info</type>
    <code>754</code>
    <desc>local structure member 'X::a' (line 4,
       file fileb.cpp) not referenced</desc>
  </message>
</doc>
```

Figure 2



Menu contents:
ActiveX Control Test Co&ntainer
Create &GUID
Dot&fuscator Community Edition
Error Loo&kup
ATL/MFC &Trace Tool
PC-lint (Simple Check)

Title: PC-lint (Simple Check)
Command: c:\lint\lint-nt.exe
Arguments: -i"c:\lint" std.lnt env-vc7.lnt "$(ItemFileName
Initial directory: $(ItemDir)

[✓] Use Output window
[ ] Prompt for arguments
[ ] Treat output as Unicode
[✓] Close on exit

(Very) basic IDE integration

Incidentally, some of the the `env-*.lnt` files also contain instructions on how to perform a basic integration of PC-Lint within the corresponding IDE. This usually takes the form of a custom tool, as you can see in this screenshot of an example for Visual Studio (Figure 2).

Although such an integration is how it is often done, it can be less than ideal – the operation is modal (which you will *really* feel if the analysis takes more than a few seconds), and the analysis results arrive in a daunting blob of monospaced text. Every developer has their own strategies for dealing with those issues, of course – although I suspect that the industry standard is probably extended coffee breaks and lots of grepping….

To conclude part 1…

PC-Lint is a *very* capable tool, but one in which you really need to invest the time and effort to learn how to use it effectively if you want to reap the real benefits it can yield (if you are looking for a 'quick fix' for a dirty codebase, look away now…).

Although basic PC-Lint configurations are usually pretty straightforward, things can get rather complex very quickly. To close this first part of the article, take a look at Listing 5, an example command line generated by Visual Lint.

Even though this command line could be shortened by using relative paths it is still not really something you would want to have to type very often from memory, is it?

In part 2 ('Deconstructing the PC-Lint command line') we will look in detail at how this command line is formed, and some of the common PC-Lint options. We will also discuss how to configure PC-Lint for a particular project configuration, and consider some strategies for dealing with the analysis results it generates. ■

References

[1] http://www.gimpel.com
[2] http://www.splint.org
[3] http://www.riverblade.co.uk/products/visual_lint/downloads/PcLintConfigFiles.zip

Listing 5

```
"C:\Data\PC Lint\8.00\lint-nt.exe" -i"C:\Data\PC
Lint\8.00" -background -b --u
C:\Data\Code\Projects\Applications\SourceVersione
r\Development\SourceVersioner_vs71_Debug_Win32.ln
t -u "C:\Data\PC Lint\8.00\std_vs71.lnt" env-
vc7.lnt -t4 +ffb +linebuf -
iC:\Data\Code\Projects\Applications\SourceVersion
er\Development\Debug
c:\Data\Code\Projects\Applications\SourceVersione
r\Development\Shared\FileUtils.cpp
```

C++ Libraries and Tools
to Simplify Your Life

> **POCO C++ Libraries:** free open source libraries for internet-age cross-platform C++ applications

> **POCO Remoting:** the easiest way to distributed objects and SOAP/WSDL Web Services in C++

> **POCO Open Service Platform:** create high performance component-based, manageable and dynamically extensible applications in C++

> and much more: Fast Infoset, WebWidgets, ...

> available on many platforms – highly portable code

> scalable from embedded to enterprise applications

applied informatics
Free Download and Evaluation: **appinf.com/simplify**

Beyond Programming

Stuart Golodetz addresses the challenges of working in a research environment.

Software development and programming/coding are often seen as being essentially the same thing, and it is certainly true that dealing with code is a large part of what we do. As a result, it is easy to get into the mindset of thinking that an ability to write good code is all there is to being a good software developer. However, whilst it is certainly true that good software developers must be able to write good code, there is far more to the role than simply being able to hack a great program.

In this article, I want to explore some of the challenges you may face away from the code-face, based on my own experiences during my doctorate. My own work, as some Overload readers may have observed, is in medical imaging (or more specifically, medical image segmentation), a field notable for its reliance on collaborative, inter-disciplinary work, and large amounts of data from external sources. These dependencies make life a lot more complicated; the 'I am a rock, I am an island' dictum that might serve you (or, perhaps, your team as a whole) adequately in other settings is fundamentally unsuited to the challenges posed by an environment in which (busy) people in other organisations hold the knowledge and data you require before you can even start to think about writing any code.

My goal, then, is to highlight some of the problems raised by working in this way, and make suggestions as to how to mitigate/overcome them. Whilst a few of these problems are specific to the research domain, I suspect that many of them are equally applicable in the commercial settings with which a large number of you are probably more familiar. I would be curious to hear from people as to whether this suspicion is borne out in practice!

Non-technical problems are sometimes harder

In many ways, for technically-minded people like us, actual programming is the easier part of our job. Whether we misspent our formative years honing our ability to write the 'perfect' code comment, or learnt our trade from scratch on the job, coding is such a large part of what we do that we quickly get quite good at it: practice makes perfect.

Moreover, coding is by and large a deterministic activity: you write code, and the computer does what you told it to. You might have told it to do the wrong thing, but that's your own fault. What happens doesn't depend on the amount of pizza you ate beforehand, the phase of the moon, or the existence (or otherwise) of your pet hamster...unless it sadly happened to chew through the power cable, of course. I know that in some cases the *observable* behaviour can seem non-deterministic – think of undefined behaviour, multithreading, etc. – but your computer is a machine, and thus in principle predictable if you understand all the factors involved.

Finally, and somewhat related to the previous point, coding itself can be done largely independently of other people. This is not to say that it should be, and indeed interaction in the form of pair programming, code reviews, etc., makes good sense, besides making your job a great deal more interesting and fun, but you can (in principle) write code on your own, without having to interact with those around you.

None of this is to suggest that programming is an easy activity: far from it. My point is merely that the other aspects of software development I want to discuss are comparatively hard, especially for those with a primarily technical mindset.

What's your problem?

Many of the trickier problems you come across in software development are really process issues, less to do with the actual code and more to do with your interactions with key people like your users. These are the sort of problems that can cause your project to fail if not resolved. If you can't implement feature X or Y due to a coding issue, that may or may not be a deal-breaker for your project. If you're building a system your users don't want or, worse, you don't know what it is that they want, then your project is certainly doomed.

Let's look, then, at a few of the potential problems you might face:

The unknown system

You can't build a system if you don't know what it should do, and as such you must make it an absolute priority to pin this down as soon as possible. This seems like it's so obvious that it's barely worth saying, but it can be easier said than done in some cases. For instance, your users may not be familiar with what is possible on a computer: they may not know that a particular feature that could make their lives easier is implementable. Conversely, they might want you to automate the solution to a problem which can't even be solved by hand yet (even in principle). As software developers, we need to develop our communication skills so that we can clearly explain our own side of the story.

The other side of the coin is that your users generally understand their own problem domain far better than you could ever hope to; the problem here is in how to ask them the right questions to obtain the information you need to do your job. Useful tips include talking generally to users about what they do: this may help you ascertain areas where you might be able to help, and you can then ask more targeted questions about those areas. Good observational skills are also key here; simply by observing your users doing their job, you may notice things which would seem merely routine to them. Active listening is important: don't rely on them *guessing* what you need to know, *tell* them clearly and then take careful note of the answers you get.

Generally speaking, this whole process is one of *requirements analysis*, something that might seem to have little to do with the actual development of the software. Indeed, in large companies, you might well have a business/requirements analyst doing this sort of work, leaving the developers to get on with the code. Having a specialist facilitator present will certainly help the process along, but in my opinion it's still helpful for developers to interact directly with users if possible. If nothing else, it helps to build trust with your users, who will have more confidence in you if they can interact directly with the team doing the work and see for themselves that they're competent.

Data shortage

Some systems, for example those that work with medical images, require a substantial amount of data on which to work. You can't build such a system without it, but acquiring it can easily become a bottleneck, not least because you need to have a very clear idea of what data you need in the first place, something which generally requires a reasonable understanding of the system you're trying to build (see 'The unknown system', above).

STUART GOLODETZ

Stuart has been programming for 13 years and is studying for a computing doctorate at Oxford University. His current work is on the automatic segmentation of abdominal CT scans. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

Acquiring data is a process that can take a substantial amount of time, and you need to be as prepared as you can be for this in advance. For a start, data is rarely doled out haphazardly to anyone who asks for it (at least, it shouldn't be): generally you'll need at the very least to be able to present a good case for why you need the data to the people who currently hold it (your data holders, as I'll refer to them). You should definitely make the benefits of your system to them (not to you, or to random third parties) clear up-front if you want their help (this is a widely applicable principle in life in general, actually).

You need to be prepared for the fact that the data holders may not have the exact type of data that you need, or at least may not have easy access to it. If this is the case, you may have to redesign your system (including its feature list) to make use of the data available, a process that can take time, or alternatively try and find the desired data elsewhere, if it exists at all. Bear in mind that it can be easier and more productive in such a situation to spend the time and effort to change the design of your system than to try and establish good working relationships with other data holders elsewhere from scratch.

Even assuming that you've reached a stage where both you and your data holders can see some value in the system that you're proposing, and they've agreed to give you the data, there may be further obstacles to overcome. One minor issue is that it may take some time for them to actually acquire the data for you. There's nothing you can do about this besides keeping up-to-date with them to know how things are going, so you just have to find something else to do while you're waiting. The bigger problem at this stage, especially when dealing with things like medical images, is that there may be legal hurdles to your getting the data: in particular, you may need to apply for things like ethics approval. The best advice in this situation is to talk to the data holders about the way the process works: they will often have been through it before, and can give you useful advice. If possible, ask them to work with you when applying, as letters of support from them can help the process work more smoothly. With luck, the hard work you and they have put in will eventually pay off.

Time is money

It's a fact of life that the most skilled and useful people are often the busiest. Whilst there's thus some truth in the old adage that 'if you want something done, you should ask a busy person', you need to at least be aware that your users/data holders have many other calls on their time (many of which will, in all honesty, be more important than meeting you) and that you need to value the time they do grant you highly. It goes without saying that these people are vitally important to the whole project: you literally can't build your system without their help! The goal, then, is to take up as little of their time as possible, but to use the time you do spend with them productively.

There are two main issues here. Firstly, you need to do your planning well in advance. This goes both for arranging meetings (which incidentally should always be done by phone, to show that you are committed enough to call them/their secretaries, and to avoid any possible confusion) and for deciding what to say in meetings themselves. It may be necessary to meet people individually, since trying to convene meetings of large groups of people is much harder and can push the meeting back for months: not what anyone involved wants.

Once you actually meet with them, you should make it clear that you value their time and intend to waste as little of it as you can, and then get to the point quite quickly. Be as warm and friendly as you are (I hope!) normally, but don't go overboard and waste their time with platitudes: show them that their time is important to you rather than just telling them. They'll appreciate the consideration you show them more than anything you might say. Remember: like everyone else, they want to work with competent equals and feel like they're getting something out of the process too, not to feel like they're doing you a favour.

Avoiding Kermitdom

Continuing on from the last point, you may initially need to work hard to persuade people that you're competent and trustworthy (alternatively, 'not a muppet'). This is especially the case when establishing a completely new working relationship with people in another organisation. References from other people they already know and trust can certainly help break the ice here, particularly if they've worked productively with them in the past, but ultimately you will still have to prove yourself to them directly (after all, your referees may have inadvertently missed your budding ineptitude, and your new colleagues will probably want to make sure about you).

One good thing you can do here is to build them a prototype. This doesn't have to be a prototype of the system itself (especially if you don't yet know what that is), but it's well worth building *something*, preferably of relevance to the problem domain. Feel free to use old work for this as well, but make sure it's relevant.

Money takes time

In a research environment, hiring new people can take much longer than it would in a commercial setting. Whilst the process of interviewing and deciding to hire someone new takes the same amount of time in both cases, funding issues are different. Researchers need to be aware in advance of university funding procedures and, since funding applications have a long turn-around, to plan ahead.Getting funding without a track record in a particular domain is especially hard (and rightly so in many ways), but if your group is branching out into new areas then it may once again be a case of having to prove yourself, this time to your potential funders (e.g. a government research council).

Applying for funding can certainly be a difficult process, but there are at least a few things you can do to make it easier. First of all, get the timing right. If you submit your application too early, you probably won't have been able to define your project in detail, and it will get rejected. This is a waste of everybody's time: don't be in unnecessary haste. The flip side is that once you know you have your project reasonably well pinned down, you should really knuckle down and get the application in as soon as possible. Applying for funding may be a somewhat onerous process, but you have to do it sooner or later, and the long turn-around means that it really should be sooner in this case.

The second piece of advice is to gain some experience in your chosen field before applying if you don't have any in advance. For example, nobody wants to fund the physicist applying to study medieval history because he thought 'knights sounded interesting'. Do your homework. Yes, it can be hard to do your initial research without proper funding, but that's the way it is unfortunately.

Finally, bear in mind that if you can get any sort of grant to do some research (however small an amount), you can use it as a basis for future applications. Feasibility studies are a good way to get some initial funding, since people are always happier to shell out a small amount of money for a risky idea than a large amount. You can always apCply for further funding later on.

Conclusions

As software developers, it is important for us to understand the whole development process, and where programming fits into it. Programming is obviously a key skill for a developer, but the challenge is far greater than just one of writing code. The vast majority of the non-technical skills you need are largely untaught, so you have to pick them up on the job. The key things to learn are communication skills, planning and patience. You need to be able to communicate clearly with users, data holders and others, and understand their needs if you are to have a productive working relationship with them.

Training can help with this, but in truth you learn most from being put in situations where the sort of issues we've discussed arise. I would suggest that the lesson for employers, then, is to enhance their developers' skill-sets by putting them in situations where they have to exercise new skills. Obviously developers are still primarily there to write code, and shouldn't be forced to undertake roles which don't suit them, but at the very least, showing them the whole picture enhances their effectiveness at what they do best. True code gurus will generally welcome the opportunity to understand their work in context in any case. ■

This 'Software' Stuff

Pete Goodliffe takes a peek into a sweet can of software development. And it's not fizzy.

It's a sad fact of life that I won't be able to rely on my sharply honed intellect forever. At some unspecified time in the future my wits will fade, and I won't be the sharp technical genius I am now. So I need a pension plan, a way to make my millions so that I can live in luxury in my old age. Or considering the current credit crunch, perhaps I should be aiming to make *billions*, just to be safe?

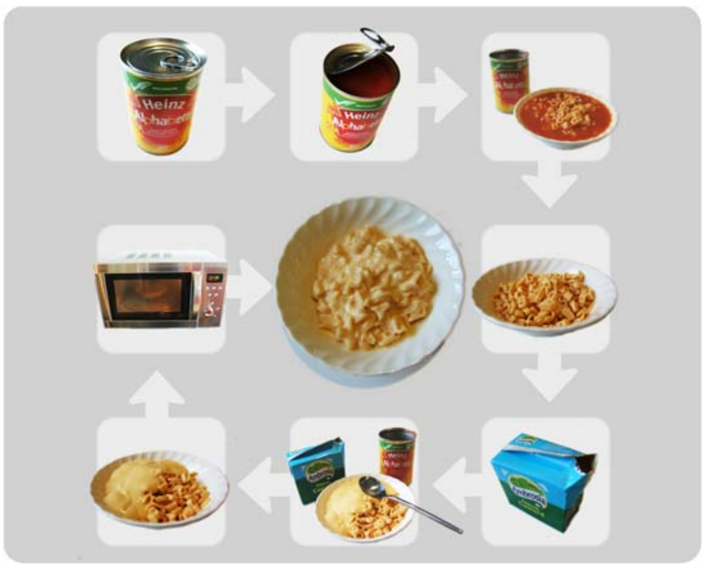
Figure 1 shows my original plan for world domination. It was so simple it couldn't fail. Milk + fizz = *fizzy milk*! Original experiments with a Soda Stream proved somewhat messy. And it turns out that curdled milk doesn't taste very nice. But that was a mere implementation issue.

Figure 1

However, before I got a chance to work out the finer details of the recipe, I received some devastating news... fizzy milk has already been done. Some evil American company was already selling it. Gutted, and with the patent rights slipping through my fingers, I went back to the drawing board to come up with my new pension plan. And this time it's a good one.

This sure-fire money spinner goes back to the classic foods of my youth: custard and Alphabetti Spagghetti. I'll share the secret with you, because you look trustworthy. I'm sure you can see where I'm going: Alphabetti Spaghetti + custard = *Alphabetti Custard*! My initial experiments have proved promising. Figure 2 shows the latest recipe.

Figure 2



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@cthree.org

The real trick here is to wash the spaghetti first, otherwise you end up with a really quite bizarre concoction. Admittedly, even when you do make it right you end up with a relatively bizarre concoction; it's kinda like rice pudding, but wheatier. It's an acquired taste, but I think it could catch on. I'll be rich. And famous. Probably.

Just don't steal my recipe.

So what?

I can imagine that by now, you're wondering why on earth I'm telling you all this? Is there some twisted moral to this sordid story? Yes, there is. And it's not tenuous at all.

I'm telling you this because of a simple observation that I've made about the software industry in the years I've been incarcerated within it. Too much software is like my Alphabetti Custard: it's *the wrong thing*, written *the wrong way*.

It's obvious how to make Alphabetti custard the 'right way' – you'd make the pasta first by hand and avoid all that tedious washing, wouldn't you? (Just nod and say "yes" at this point). Anyone with industry experience will know what I'm talking about (in the software sense, at least).

As conscientious software developers we should all aspire to the opposite: to write *the right thing* in the *right way*. One of the keys characteristics of truly excellent programmers is actually caring about the software that we write, and how we write it. That's what I'd like to investigate in this mini-series of articles. My aim (grand as it is) is to work out how to become better programmers And here's how we're going to do it.

I will pose a series of personal questions as we go along. So how much you get out of these articles depends on how much you are willing to put in. It's you own fault it they don't work! When you see each question, consider if it applies to you. Remember the ones that are most pertinent to you . And then think about what you can do about them?

So that's the game plan, and here's your first question...

Do I...want to improve as a programmer? Do I actually want to write the right thing in the right way?

If your answer is "no" then give up and stop reading now.

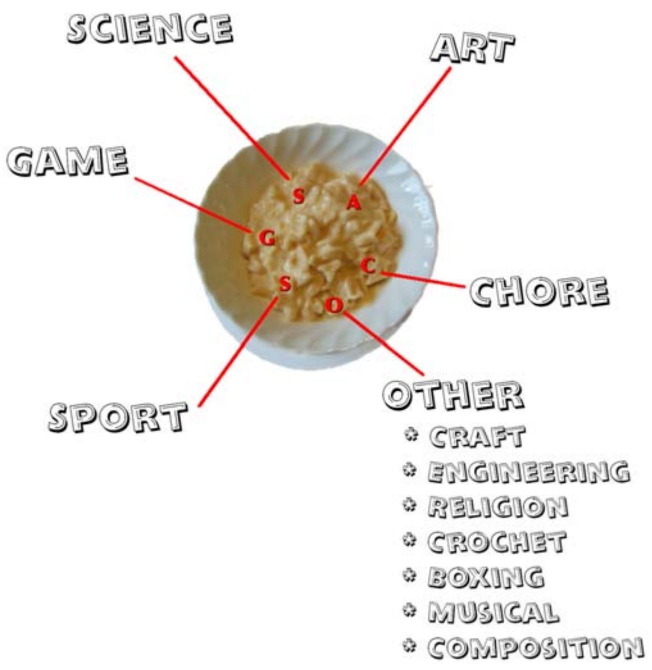
This software stuff

So, back to this 'software' stuff. If we want to get better at writing it, then the first question has to be: *what is it?* Let's peer into the bowl and find out. To be sure, software is complicated stuff. It has many aspects. Now, this isn't going to be a great intellectual treaty on software development. But as the quick breakdown in Figure 3 shows, it is part science, art, game, sport, chore, and more. (That's a real bowl of Alphabetti Custard, by the way.)

Based on that, how can we find keys to help us deliver the right stuff in the right way? I have this strange fascination with what an 'ideal' programmer would look like. I'd like to know what to aspire to. And so based on the unscientific breakdown above, and after some genetic research, I have discovered what the ideal programmer looks like.

Feast your eyes on Figure 4: *the ideal programmer*. Now that's something to aspire to. He looks like the kind of person who'd eat Alphabetti Custard!

Figure 3



We'll look at each of these aspects of software development in turn and see what we can learn from them.

Software is... an art

So first up, a great programmer needs to be a great artist. But is programming *really* an art? That's a debate that has long been held in software development circles. I'm sure that someone somewhere is still whittering about it. Some people think that programming is an engineering discipline, some an artform, some sit the in between, considering it a craft (and I did call my book *Code Craft*, after all).

Knuth is probably the most famous proponent of software as art, naming his series of books *The Art of Computer Programming*. He said this: Some programs are elegant, some are exquisite, some are sparkling. My claim is that is it possible to write grand programs, noble programs, truly magnificent ones! Stirring stuff. There's more to code than bits and bytes. There's more than brackets and braces. There's structure and elegance. There's poise and balance.

Whether or not we can generalise about the whole programming pursuit, a good programmer certainly needs a sense of aesthetics to write exceptionally good code. And there are many parts of the development process akin to the creation of a work of art. The process is:

- **Creative** It requires individual, personal use of imagination. The software must be skilfully constructed and precisely designed. Programmers must have a vision for the code they are about to create, and a plan of how they will make it. Sometimes that involves a great deal of ingenuity.
- **Aesthetic** Good code is hallmarked by elegance, by beauty and balance. It stands within certain cultural idioms. We consider the code's form and its function.
- **Mechanical** As any artist, we work in our particular medium with our particular tools, processes and techniques. We work under commission for generous benefactors.
- **Team-based** Many forms of art are not single-person endeavours. Not every art form sees an artist setting alone in their studio slaving day and night until their masterpiece is complete. Consider the orchestra; each member held together by the conductor. Consider a musical composer, writing a piece which will then be interpreted by the performer(s). Or the architect designing a building that will be erected by a team of builders.

We haven't the time or space to go into any more depth in this article, but it's clear that in many respects, the skill set of an artist is similar to that of a programmer. Michelangelo was the archetypal renaissance man: a painter, sculptor, architect, poet, engineer. Perhaps he'd've made an incredible programmer! When asked about how he created one of his most famous works, the statue of David, he said: I looked into the stone and saw him [David] there, and just chipped away everything else.

Is that what you do? Do you reduce and remove the complexities of the problem space, chipping them all away until you reach the beautiful code you were aiming for?

So here are a few questions to ask yourself...

Figure 4

Do I... consider the creative aspects of software development, or treat it as a mechanistic logical activity?

Should I... develop a keener sense of elegance and aesthetics in code? Should I look beyond what's functional and solves the immediate problem?

Do I... think that my idea of 'beautiful' code is the One True Opinion? Should I consider artistry as a team pursuit?

Until next time

That's quite a lot to be thinking about until next time. In the next article we'll look at programming as a science, and programming as a sport.

Until then, don't eat too much Alphabetti Custard.... ■

Cititec

www.cititec.com

Greenfield Software Development Careers in Financial Markets

Cititec are specialist and boutique global suppliers to a large number of Investment banks as well as many smaller financial institutions.

Stefano Cicu specialises in placing C++, C# and Java developers across a range of business sectors and he is always interested in speaking to fellow ACCU members and introducing them to compelling opportunitites.

We have a variety of vacancies ranging from Tactical / RAD Development through to huge global strategic projects using emerging technologies.

Stefano can be contacted to discuss opportunities or your recruitment needs if you are hiring at **Stefano.cicu@cititec.com** or you can call him directly on **0207 608 5879**

atsco INVESTOR IN PEOPLE Recruitment & Employment Confederation

Tell me about... Virtualization

Thomas Guest explains the virtues of virtual machines.

Virtualization solves a computing problem by adding a layer of indirection. The problem being: how to run multiple operating systems on a single computer; and the indirection: to slip a software shim between a guest operating system and the hosting platform, which continues to run its native operating system.

Figure 1

An example makes this clear. I work on an Apple computer which runs OS X, a flavour of BSD Unix. To develop portable code which will build and run on Linux and Windows as well as OS X, I use virtualization software. Using this software enables my Apple computer to run (for example) Windows XP and Linux Fedora Core 7 as guest operating systems alongside its native OS X. Effectively, I have three computers running on the same physical machine with no need for extra power supplies, keyboards, mice, monitors and so on. (Figure 1.)

The Parallels Desktop [1] virtualization software I use doesn't come free but it's cheaper and more convenient than buying more hardware. VMware [2], perhaps the single biggest name in virtualization, does offer free entry-level products for doing a similar job on Windows and Linux platforms.

This article will not provide details on installing and configuring virtualization software, troubleshooting problems, and so on: the products have matured to the extent that these details are hardly needed, and any problems are quickly answered by online forums. We won't attempt to explain how exactly virtualization works. Instead, we'll talk more about what virtualization is, how we can use it, and why we should be interested it.

Virtualization in general

Before we go further I should explain this article uses the term 'virtualization' in the specific way described in the introduction. More generally, in computing, virtualization refers to the abstraction in software of the platform on which a program runs. The Java Virtual Machine is a well known example, allowing software developers to build a single executable which should run on any machine. The JVM also isolates the application from the rest of the machine. These advantages, of portability and isolation, also apply to the full virtualization we'll discuss in this article.

Creating a virtual machine

Setting up a virtual machine feels just like setting up a normal machine, except you don't need new hardware. All you need is your host machine, the guest operating system media, and a suitable licence to use it. With your host up and running, mount the install media, start up the virtualization software, click 'Create new machine', and follow the prompts. You'll have to specify what resources to grant the machine (disk space, RAM, etc.) but very quickly you'll be following the standard install procedure for your guest OS, selecting languages, packages and so on.

You don't need actual physical media: you can create your virtual machine by booting it up from a DVD image on disk or over a network in much the same way.

Here's one we made earlier

Actually, you may not require **any** install media. Your virtualization software is capable of booting up a pre-built virtual machine. VMware terms such machines virtual appliances [3]. Running such an appliance is as easy as downloading it (which, at around 300Mb or less, requires far less bandwidth than a typical install image) and clicking on the downloaded file.

What you'll typically be getting is a stripped-down Unix server, pre-built for a specific purpose, with stable, tested, compatible versions of whatever packages it requires for that purpose, and capable of operating within, say, 256Mb of RAM and as much hard disk as you're prepared to allow it. You can run this Unix server on a Windows machine. You can reconfigure it. You can even transfer it to a different machine.

As an example, suppose you want evaluate Trac [4], an integrated version control and project management application. Trac may be open-source, popular and free, but I can personally vouch that on a Unix system it takes some setting up, and I can't imagine getting it to work natively on a Windows server. Using virtualization, you simply download a virtual machine which has been loaded with the latest stable release of Trac. Boot up this machine using your host virtualisation software and run it on any supported operating system – Windows included. Do the same with Redmine [5], another web-based project management application, and you can compare it with Trac. Once you've completed your evaluation, delete the one you don't like and keep going with the other. As a virtual machine, it's easy to move it to a new host, if desired.

VMware provide instructions for creating appliances and host a library of such appliances [3] on their website. Organisations like JumpBox [6] make a business out of providing virtual machines which run on a number of different virtual platforms.

Running a virtual machine

Parallels Desktop creates shortcuts which I click on to power up the virtual machines. VMware on Windows does something similar. My perception is that these virtual machines boot as quickly as their physical counterparts would, but it could simply be that I'm using the host system to do something else while they start up in the background.

Exactly how the guest operating system integrates with the host varies. Some systems/configurations give you a window within a window; the guest user interface is displayed as a whole within a single window on the host, and you switch focus to this window to use it. More sophisticated systems integrate seamlessly, so you can tab between host and guest applications as if they were all running natively, and guest and host file browsers see both machines' file systems transparently. The end-user experience is of the host and guest operating in parallel, as a single computer which can run software native to both systems.

In my experience, the first mode can be awkward to use. I much prefer the second: any friction context-switching between machines, and you find yourself preferring separate machines and a KVM, or using an X server to display X windows presented by a remote machine.

THOMAS GUEST

Thomas is an enthusiastic and experienced programmer who has worked on everything from embedded systems to clustered servers. His website is http://wordaligned.org and he can be contacted at thomas.guest@gmail.com

Peripheral access

Which peripherals can the guest operating system access? Certainly, the guest wouldn't be much use if it couldn't make use of monitor, keyboard and mouse – although you may suffer translation and configuration wrinkles due to the different keyboard layout and mouse button conventions used by different operating systems.

Access to other peripherals and interfaces will depend on the virtualization software: check the product information. My guest Windows XP can use its host's network interfaces, USB ports, DVD drive, speakers and built-in camera. Actually, I didn't realise it could use the camera, having never had cause to use the camera from within Windows, but a quick check shows it can. Access to networked printers also just works.

Opportunities

I've already mentioned some obvious uses for virtualization, and I'll add some more which I've found useful in the past:

- you can develop for multiple platforms using a single machine
- you can download pre-built machines designed to run particular applications, saving you from package management headaches
- if you use a laptop, virtualization allows you to carry many machines around with you: a sales person could demonstrate Unix-based software on a Windows machine, for example
- you can script the creation of machines, and test e.g. clustered server configurations, without needing a rack filled with hardware
- you can test on multiple platforms.

Even if your application itself is server-based and only runs on a single platform, virtualization allows you to test its web interface on multiple browsers. And even if your development is tied to a single operating system, virtualization allows you to keep old versions of that operating system alive on new hardware, and indeed to constrain the resources available to these old versions.

Hosting companies often use virtualization to create an indirection between user accounts and the hosting hardware farm. Users have root access to their own virtual machine yet are isolated from other root users on the same hardware (for example, rebooting a virtual machine doesn't affect other machines on the same host); and their virtual machine can be transferred between physical hosts without them even realising.

It would even be possible to distribute software as a virtual appliance. Rather than requiring your users to install version X of Python, version Y of SQLite, version Z of the database bindings and so on, you might consider distributing an entire system which runs as a virtual machine.

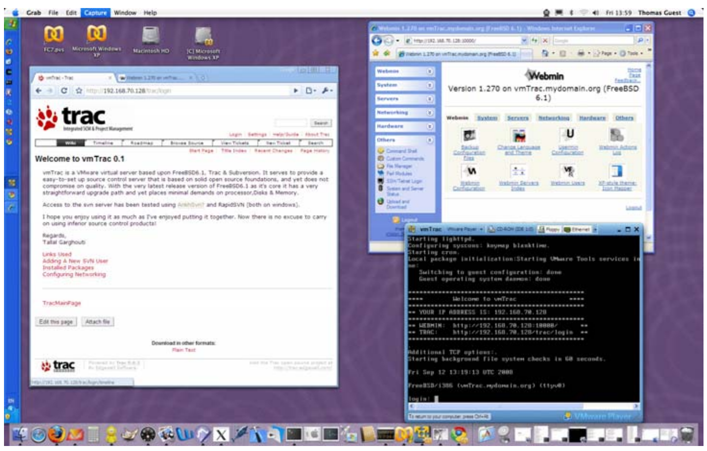
Considerations

Running a virtual machine requires real resources. I deliberately chose Windows XP over Vista for this reason; XP has the smaller footprint and it's all I personally need for developing software which ports to Windows.

As already mentioned, your guest operating system needs licensing. You need to pay to use Windows even if you're running it on an Apple computer and have already paid for OS X. You'll also need to go through the usual activation procedure.

You'll need to tend to your virtual machines like any others on your local network. They need naming and backing up. User accounts must be created. Depending on what presence they have on your network, you may want to configure DHCP, or take anti-virus measures. You also need to consider upgrading them.

I have run into wrinkles and irritations with the hardware abstraction side of virtualization. For example, the Apple keyboard I use doesn't map exactly to what I'd want when using Windows. It's occasionally taken me some fiddling with X configuration files to get a Linux graphical interface displaying properly. Generally, though, someone else will have found and fixed the problem before you, and searching online forums turns up an answer.


Grab File Edit Capture Window Help
FC7.pvs
Microsoft Windows XP
Macintosh HD
[C] Microsoft Windows XP
trac
Integrated SCM & Project Management
Login | Settings | Help/Guide | About Trac
Wiki | Timeline | Roadmap | Browse Source | View Tickets | New Ticket | Search
Edit this page
Attach file
Webmin
System
Servers
Networking
Hardware
Others
VMware Player

Welcome to vmTrac 0.1

vmTrac is a VMware virtual server based upon FreeBSD6.1, Trac & Subversion. It serves to provide a easy-to-set up source control server that is based on solid open source foundations, and yet does not compromise on quality. With the very latest release version of FreeBSD6.1 as it's core it has a very straightforward upgrade path and yet places minimal demands on processor,Disks & Memory.

Access to the svn server has been tested using AnkhSvn and RapidSVN (both on windows).

I hope you enjoy using it as much as I've enjoyed putting it together. Now there is no excuse to carry on using inferior source control products!

Regards,
Tallal Garghouti

Links Used
Adding A New SVN User
Installed Packages
Configuring Networking

TracMainPage

Download in other formats:
Plain Text

Version 1.270 on vmTrac.mydomain.org (FreeBSD 6.1)

```
Starting lighttpd.
Configuring syscons: keymap blanktime.
Starting cron.
Local package initialization:Starting VMware Tools services in
ne:
   Switching to guest configuration: done
   Guest operating system daemon: done

************************************************
****         Welcome to vmTrac              ****
************************************************
** YOUR IP ADDRESS IS: 192.168.70.128
************************************************
** WEBMIN:  http://192.168.70.128:10000/       **
** TRAC:    http://192.168.70.128/trac/login   **
************************************************

Additional TCP options:.
Starting background file system checks in 60 seconds.

Fri Sep 12 13:19:13 UTC 2008

FreeBSD/i386 (vmTrac.mydomain.org) (ttyv0)

login:
```

Figure 2

In this article's introduction we classed virtualization as yet another computing problem solved by indirection. Indirection has a price. What about the accumulated expense of everything passing through the software shim which abstracts the platform? Surely a guest operating system can't be as fast as a native one on equivalent hardware? I have no hard figures to present here but personal experience suggests no perceptible difference: the only thing I have noticed is that my guest Windows XP seems to use only one of its host's two CPUs.

Just for fun

We've seen it's possible and sensible for a platform to host a guest operating system within its native operating system. How about trying something silly? Could our guest operating system itself use virtualization to become a host for a guest of its own?

I gave it a go. Using Parallels Desktop on my OS X host, running Windows XP as a guest, I installed (the free) VMware Player virtualization software for Windows. So far, so good. Next I downloaded vmTrac, a 113Mb VMware appliance which packages Trac, Subversion, WebMin and Lighttpd on a FreeBSD core. I extracted the archive and opened the appliance using VMware Player.

Figure 2 shows my desktop. I've used Google Chrome and Internet Explorer to access Trac and WebMin, which are running as web applications on FreeBSD, itself running as a Windows XP guest, and Windows XP is a guest on OS X. ■

Further reading

http://www.justsoftwaresolutions.co.uk/testing/testing-on-multiple-platforms-with-vmware.html

Although virtualization products are mature and modern hardware is ready to accommodate them, the options and possibilities still take some explaining. You'll find plenty of good material on the VMware [2] and Parallels [1] websites. It's also worth searching for other virtualization platforms. This is a growing market, there are lots of competitors and good deals to be had.

Testing on Multiple Platforms with VMware by Anthony Williams provides a clear overview of the subject addressed in this article.

References

[1] http://www.parallels.com/
[2] http://www.vmware.com/
[3] http://www.vmware.com/appliances/
[4] http://trac.edgewall.org/
[5] http://www.redmine.org
[6] http://www.jumpbox.com

Thanks

I would like to thank everyone at CVu for their help with this article.

XML is not the build system you're looking for

Paul Grenyer provides an introduction to Gant.

On Thursday 21st August I attended a Skills Matter [1] 'In the Brain' session on Gant [2] given by Russel Winder at their offices in London:

> I will be doing an 'In the Brain' session on Gant (The Groovy way of scripting Ant tasks) on Thursday 2008-08-21 18:30. This will happen at Skills Matter, 1 Seckford Street, London EC1R 0BE, UK.
>
> As part of this session I am going to undertake 'The Gant Challenge'. The idea is for people to bring small examples of Ant (or other) builds that really irritate them so we can create the Gant version live and show that Gant can do the business where Ant often cannot.
>
> If you are in the area then, feel free to drop by – though you need to register beforehand so some forethought is needed. This is planned as a 90min session after which things move to a local hostelry.

Skills Matter is an organisation which supports the Agile and Open Source developer community, by organising free events, training courses, conferences and by publishing thousands of podcasts on ideas and technologies that drive innovation.

Russel's 90 minutes soon became 120, but it was very informative and very interesting. The basic gist was that, although Ant [3] is a powerful build system, XML is not the best way of specifying the steps in a build process. Chiefly because it is verbose, not human readable as a programming language can be and it's difficult to do common programming tasks such as **`if`**, **`for`**, **`foreach`**, **`switch`**, etc.

Gant is written in Groovy [4] and sits on top of Ant giving an extra layer of indirection to existing Ant tasks. I had only heard of Groovy before and never used it, so Russel spent a little time showing me, and the others, simple Groovy programs and how to run them. I was also introduced to a new concept known as closures.

Wikipedia described closures [5] as follows:

> In computer science, a closure is a function that is evaluated in an environment containing one or more bound variables. When called, the function can access these variables. The explicit use of closures is associated with functional programming and with languages such as ML and Lisp.

Gant build scripts are also Groovy programs and bring with them all the power of the Groovy programming language, so it is possible to do just about anything and have a more readable, less verbose method of describing build steps.

Once I got home and found a spare hour I gave Gant a go. I'm using Windows. So I downloaded the Windows installer for Groovy [6]. It was easy to install. I just used all of the defaults, except that I specified that environment variables should be system rather than user and I didn't install Gant as it was only version 1.2.0 and a later version was available from the website. I had to fiddle with the environment variables by hand to get them right, but eventually I got:

```
groovy -v
Groovy Version: 1.5.6 JVM: 10.0-b22
```

It should be noted that the uninstall does not remove the environment variables so **`GROOVY_HOME`** and the system path entry to the `bin` folder must be removed by hand if uninstalling.

PAUL GRENYER

An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

Listing 1

```
includeTargets << gant.targets.Clean
cleanPattern << [ '**/*~' ,  '**/*.bak' ]
cleanDirectory << 'build'

target ( stuff : 'A target to do some stuff.' ) {
  println ( 'Stuff' )
  depends ( clean )
  echo ( message : 'A default message from Ant.' )
  otherStuff ( )
}

target ( otherStuff : 'A target to do some other
   stuff' ) {
  println ( 'OtherStuff' )
  echo ( message : 'Another message from Ant.' )
  clean ( )
}

setDefaultTarget ( stuff )
```

I installed the latest version of Gant [7] from the website. Installation consisted of unzipping the zip file and adding the path to the Gant `bin` folder to the system path. Ant is also required and **`ANT_HOME`** must be set. Once they were in place I got:

```
gant -V
Gant version 1.4.0
```

All relatively straight forward so far. Now for a test. The Gant home page gives Listing 1 as an example.

I dropped it into a file called `build.gant`, typed **`gant`** at the command line and got:

```
Stuff
     [echo] A default message from Ant.
OtherStuff
     [echo] Another message from Ant.
```

Brilliant! Assuming Groovy and Ant are properly installed it would appear that Gant works straight out of the box. I'm a big fan of things that work straight out of the box and always strive to make my own projects work that way.

Now for the acid test, a straight comparison between a simple Ant build script and Gant one. Let's take a very simple Java program:

```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello, Gant World!");
  }
}
```

The usual Ant script to compile this would look something like Listing 2. It's quite verbose with a lot of angle brackets. Although XML is meant to be human readable, it's far from simple to read. The equivalent Gant script, which I worked out in about 5 minutes from the docs and Russel's session, is Listing 3.

As you can see the Gant script is clearer, simpler and easier to read. Like the Ant script, it has two targets with the run target dependent on the

Let the Machine Debug For You

Robert Finking takes stock of code analysis techniques.

Over the past few decades a number of semi-automated debugging technologies have emerged. Rather than doing all the debugging by hand, why not let the machine do some of it for you?

As with computer aided anything, there are strengths and weaknesses of the techniques available. Automated debugging tools are great at spotting some bugs and completely blind to others.

The two classes of tools which we're going to look at in this article are great at spotting run time bugs, but no good at finding functional defects. What sorts of things do we mean by run-time bugs?

- Null pointer dereferences
- Array/alloc bounds overruns (read or write)
- Uninitialised memory reads
- Double free
- Memory leaks

Generally speaking we're talking about tools for Java/C/C++ here. There are bits of tool support available for other languages but the bulk of what is available is for C derivative languages. There are two reasons for this. Firstly, it is a lot easier to reason about what is going on with a piece of source code if it is statically typed. With more dynamic languages such as Python, it is harder to produce this kind of tool. Secondly, languages derived from C tend to have a lot of potential runtime problems that other languages have defined away. This is less true of Java, but it has its own problems to worry about!

The two techniques we're going to explore in this article are static analysis and dynamic analysis. Dynamic analysis covers a wide range of techniques, but we're explicitly going to look at automatic memory usage checkers. Static analysis also covers a wide range of techniques, many of which are still being researched. We are going to focus on the kinds of tools currently available for use today.

Example malloc debuggers

- mpatrol
- NJAMD (Not Just Another Malloc Debugger)
- Electric Fence/DUMA [2]
- Dmalloc
- BoundsChecker.

Dynamic analysis

Dynamic analysis of code can tell you *what* problems exist and *how* they occur.

Malloc debuggers

Malloc debuggers are the most widely available form of free tool. Most of these are distributed in source form and as a result work on a wide variety of platforms. They operate by providing an alternate library for memory allocation calls. Some of these use non-standard function names and

ROBERT FINKING

Robert Finking is a professional software engineer who is always trying to improve and pass it on. He is currently engaged part time in a research programme attempting to break new ground in software tools technology. He likes Jesus and beer. He can be contacted at robertfinkng555@o2.co.uk (the mis-spelling of Finking is intentional).

XML is not the build system you're looking for (continued)

compile target. The compile target uses javac to build a Java class file and the run target executes the class file using java. Finally the default target is specified.

This is all very attractive to me, despite needing to add yet another tool to my build environment installation. Overall it was a very interesting, informative and enjoyable session. I always enjoy anything that Russel is involved in, he has a lot of charisma. It was certainly worth the five hour round trip from Norwich for the evening. ■

References

[1] http://skillsmatter.com/
[2] http://gant.codehaus.org/
[3] http://ant.apache.org/
[4] http://groovy.codehaus.org/
[5] http://en.wikipedia.org/wiki/Closure_(computer_science)
[6] http://dist.codehaus.org/groovy/distributions/installers/windows/nsis/groovy-1.5.6-installer.exe
[7] http://dist.codehaus.org/gant/distributions/gant-1.4.0_groovy-1.5.6.zip

Listing 2

```
<project name = "HelloWorld" default="run"
   basedir=".">
  <target name = "compile">
    <javac srcdir="src" />
  </target>
  <target name = "run" depends="compile">
    <java classname="HelloWorld">
      <classpath>
        <pathelement path="${basedir}"/>
      </classpath>
    </java>
  </target>
</project>
```

Listing 3

```
target ( compile : 'Compiles.' )
{
   javac ( srcdir : "src"  )
}

target ( run : 'runs.' )
{
   depends(compile)
   java( classname : 'HelloWorld',
         classpath : 'src' )
}

setDefaultTarget ( run )
```

Mudflap options

Some of the most useful Mudflap options settable in the MUDFLAP_OPTIONS environment variable

`-mode-nop`	mudflaps do nothing
`-mode-check`	mudflaps check for memory violations (active)
`-viol-gdb`	violations fork a gdb process
`-print-leaks`	print any memory leaks at program shutdown
`-check-initialization`	detect uninitialized object reads
`-abbreviate`	abbreviate repetitive listings (active)
`-wipe-stack`	wipe stack objects at unwind
`-wipe-heap`	wipe heap objects at free
`-heur-stack-bound`	enable a simple upper stack bound heuristic

therefore require you to write your code with them in mind from the outset.

Most however provide re-implementations of the standard **`malloc`** and **`free`** operations and so can be slotted in to an existing development. It's simply a case of including them in your list of link libraries and specifying that they be linked ahead of the standard library.

In fact the standard GNU C library now includes a simple **`malloc`** debug facility. To use this the application must be started in a special mode which is enabled by setting the **`MALLOC_TRACE`** environment variable to the name of the file to use for output, and by inserting calls to **`mtrace()`** in the source code (see [1]).

As the name suggests these tools are mainly the preserve of C, though some of the libraries cover **`new`** and **`delete`** too and so are effective for C++. Since Java has garbage collection built into the VM, no equivalent is available for it. Of course that's not to say you can't have memory leaks in Java – you can. However because it's far less of a problem, dynamic debugging tools are more or less restricted to memory usage profilers such as: http://www.ibm.com/developerworks/java/library/j-leaks/heapsummary.gif

Of course that's not to say you can't have memory leaks in Java – you can

For most of the malloc debuggers available, the overhead of tracking the memory usage slows down execution considerably. One of the most well known debuggers, Electric Fence (and its descendant DUMA [2]), gets around this problem by using hardware acceleration. Since the machine's MMU generates a hardware interrupt (page fault) if an attempt is made to access memory beyond the end of the page, Electric Fence allocates all memory so that it ends at the very end of a page. That way no runtime checks need to be made in software, but rather the MMU issues a page fault if the executable attempts to access an address beyond the end of the page. There are a couple of disadvantages to this approach however. Firstly, it makes very inefficient use of memory. Even allocations of a few bytes take up an entire page (typically 4KB). For a program with large numbers of small allocations this can result in running out of memory quite quickly. The second issue is that unlike other malloc debuggers, this technique can only check for boundary overruns at one end of the allocated block, typically the top. In fact, since almost all bounds overruns occur at the top of the block of memory (loop variables almost always start at zero and go upwards), this is not a problem. Periodically re-running tests with the boundary configured to be at the bottom of the block should pick up any unusual boundary overruns that are occurring at the bottom of the memory.

Full memory debuggers

Perhaps the two most readily applicable tools covered by this article are Valgrind and Mudflap. These take a step beyond traditional malloc debug libraries and provide diagnosis of a variety of memory problems. In fact Mudflap is descended from the classic gcc patch BCC (the bounds checking compiler patch – see [3]) but Valgrind works in an entirely different way.

Valgrind [4] is in fact a framework which allows a wide variety of tools to plug into it. The main tool of use, and the one most people are referring to when they say 'Valgrind', is the memcheck tool. Valgrind works by simulating the processor and runs your entire executable inside this simulation. The beauty of this solution is that it can work on your existing executable without having to modify the source code, instrument the object code, or even re-link the executable. To analyse your code with Valgrind, simply append your ordinary command line to the Valgrind command (much like using "time" or "rsh"). Of course if you analyse a stripped executable, the debug messages you get back might not be too helpful. If Valgrind shows up a bug, you really need to point it at a version of the code built with **`-g`** in order to get details of line numbers etc. You will of course notice that **`-g`** is specific to gcc and here comes the main and only real flaw of Valgrind: it only works on executables compiled with gcc and running under Linux. Of course this applies to a wide variety of developments, but clearly not all. In particular it does not work under Cygwin or MinGW. Valgrind checks to see if your program:

- Accesses memory it shouldn't (areas not yet allocated, areas that have been freed, areas past the end of heap blocks, inaccessible areas of the stack).
- Uses uninitialised values in dangerous ways.
- Leaks memory.
- Does bad frees of heap blocks (double frees, mismatched frees).
- Passes overlapping source and destination memory blocks to **`memcpy()`** and related functions.

Almost all modern Linux distros include Valgrind either as standard or as an optional extra. Such is the usefulness of these checks that there is a case for making a non Linux codebase multiplatform just so the memory analysis can be done.

Mudflap [5] performs all of these checks and more, and has slightly wider applicability. Mudflap is a gcc 4 plugin. Some platforms (e.g. SUSE) ship with it installed as standard, others will require you to upgrade your gcc installation (no mean feat). However, Mudflap is not limited to Linux, but operates on any platform where gcc has it installed. Mudflap executables also tend to run less slowly than executables running through Valgrind since it does not rely on a simulation of the processor. The killer bug detection feature Mudflap has over Valgrind is its ability to detect memory bugs on the stack. Valgrind on the other hand is restricted to the heap (due to the way it works). Mudflap is harder to use however. Even if Mudflap comes pre-installed on your version of gcc, it takes some configuring to get it to behave like you want it to (see sidebar). Once you're familiar with it, this is another strength – it is very configurable. The learning curve to get started is definitely steeper than Valgrind though.

If you are on a platform that supports Valgrind but not Mudflap, and you want a quick and dirty way to get a bit of stack memory checking without the hassle of rebuilding gcc, there is an easy solution. "Propolice" [6] comes installed as standard on version 4.1 and later of gcc. Compile with the flag **`-fstack-protector-all`** to enable stack checking. This feature is aimed at security rather than debugging and so is very fast. It is intended to be left in the final executable as a security measure to protect against malicious buffer overflow attempts. However it doubles as a handy stack overflow bug detector. The downside is, its security background means it doesn't give you any clues as to what went wrong - your code will just bomb out.

There are commercial alternatives to these tools such as Rational Purify, but they don't really add much and are in fact weaker in some areas than their open source cousins. The main reason for looking at commercial alternatives here is the desire to roll out such tools onto platforms that are not currently covered by Mudflap and Valgrind (in addition to the usual concerns such as training support etc.)

Lint tools

- PC-Lint/Flexelint (~£250 per seat C/C++ tool)
- Splint (FREE C tool)
- Sparse (part of FREE Linux kernel tools)
- Findbugs (FREE Java tool)
- Programming Research tools: QAJ, QAC and QAC++

Security checkers

- Klocwork
- Fortify
- Ouncelabs

Static Analysis

Static analysis tells you *why and where* problems exist as well as *what and how*.

Lint tools

There are various categories of static analysis tools, the widest used category being ‘Lint Tools’. These have evolved from the early UNIX lint command which used to sit alongside the compiler in order to carry out more thorough checks than the compiler had time to do. With increased computing power on the desktop, most of the checks that lint used to perform are now built into the compiler in the form of warnings. If you currently use default compiler options, a useful step forward is to read the manual and switch on the extra warnings which modern compilers provide (an easy one to remember for gcc is **`-Wall`**, which switches on a lot of the more useful warnings).

Modern lint tools check for three main classes of issues. Firstly and most usefully they check for potential bugs. These are almost exclusively bugs local to a single function. For example you may **`malloc`** some memory and neither free it by the end of the function, nor keep hold of the pointer to it – a simple memory leak.

Secondly lint tools often have facilities for checking code against particular rulesets and/or standards e.g. MISRA C [7], Scott Meyers’ recommendations [8]. These are particularly useful if your customer has specified compliance to these standards. Often the commercial tools are much stronger in this respect than open source alternatives.

Thirdly lint tools check for bad style – i.e. they pick up the fluff in your code that you really don’t want to be there. Some of these checks can be a waste of time in that the chances of them ever causing a problem are so low that the cost of fixing them outweighs the benefit. The danger here is that you end up swamped with all sorts of warnings which don’t get fixed and end up missing some of the real issues because they get lost in the midst of all the warnings. There are however some very good style checks in some of these tools. We’re not talking about whether your brackets are aligned properly or whether you’ve used the correct indent – use a code beautifier for that.

Things that might make code non-portable or easy to misunderstand are the kinds of style issues addressed by the tools. Take splint for example (an open source lint tool for C). One of the checks it can perform is for two variables which look visually similar on the screen. It would warn you, for instance, if you had variables named **`bal`** and **`ba1`** in the same scope (since a number 1 looks very similar to a letter l). In a similar vein it will warn you if you have used C++ reserved words as identifiers, since it makes your code non-portable for use with C++ compiler e.g. has anybody ever written a function with parameters named **`old`** and **`new`**?

There are not a lot of open source lint tools available, and in particular at the time of writing there appear to be no free C++ static analysis tools available. There is a Sourceforge project aiming to produce a C++ version of splint but it has shown no sign of emerging from the planning stage for years [9]. The good news is that the commercial tools do not cost the earth. They are certainly affordable by a small business and even just about within reach of a home developer, costing about the same as a good graphics card.

Security tools

The second category of static analysis tool is the security checker. These tools are sometimes entirely based on static analysis, but often also use dynamic techniques. Their raison d’etre is to discover possible vulnerabilities in your code base, typically in web applications. Unlike security tools such as Nessus[10] which take a black box approach, these security checkers analyse the source code and report use of unsafe constructs, API calls etc.

Checking for security issues is a big topic, beyond the scope of this article and won’t be covered any further here other than to note that there is a degree of overlap with the final category of tools we are going to look at.

Super Lint? Meet the big boys!

So what is the final category of static analysis tool? In parallel with the evolving of lint tools a new breed of tools has emerged over the past decade or so. These take code checking to a whole new level. The main difference between lint tools and these tools is the scope of the analysis they carry out. Lint tools almost exclusively operate on a function by function basis. However as we all know, the really nasty bugs are never local, they arise due to the interaction between multiple parts of the code. And that’s where these tools score highly – they perform path analysis. They use various different reasoning techniques to spot possible paths through your code where run time errors could occur (such as null pointer dereferences), and if found, explain to you how they could happen. If lint tools complement unit testing, these tools complement integration tests. They find bugs on paths which your tests don’t execute (but which your end user probably will – you know how it is).

lint tools check for bad style – i.e. they pick up the fluff in your code that you really don’t want to be there

Another difference with these tools is that they tend to have a centralised console and issue tracking system. These are typically accessed via a web interface. This makes the tools much easier to use in a team environment where any given piece of code may be edited by multiple individuals. A single central repository allows team members to leave notes on issues to say what has been done to address them and why. Often the central consoles provide summaries, graphs, breakdowns etc. which are all useful for getting a handle on the state of the code base.

Because of the larger scale of these tools they tend to be a lot less nitpicking than lint tools. They focus on major issues such as run time bugs, security vulnerabilities, architectural problems etc, rather than the fluff. This makes them a lot nicer to pick up and start using than lint tools which do seem to be over pedantic until you get used to using them. Unfortunately, also because of the scale of the tool, no open source solutions are currently available. The required level of development is just too high it seems.

Commercial tools have come from various sources. Klocwork sell a tool which was originally produced in house to improve code quality and save on testing effort. Coverity was originally developed at Stanford University by academics and used to be available for free until somebody realised they could make money out of it! One of the older players in this sector came not from academia or industry but from outer space: Polyspace

The catastrophic failure of the Arianne 5 rocket launch in 1996 was found to be due to a software issue [11]. In order to prevent a similar error occurring in the future, work was begun to find an automatic checking tool. Polyspace was the result of that work. The proof is in the pudding: all of the other checkers mentioned in this article will report all the errors they detect, but will not guarantee to have found all errors of a particular type. Because of its background, Polyspace aims for no false negatives – i.e. if it claims that you have no null pointer exceptions in a piece of code, then you really don’t. It guarantees to find them all. However the cost of this is a belt and braces approach which doesn't let you get away with code which might conceivably result in an error. You therefore end up jumping through a lot of hoops to get your code into an all clear state. Typically therefore Polyspace is most applicable to high integrity software development where the cost of jumping through the hoops is worth it because of the stringent reliability requirements the software has to meet. Although viewed as a static analysis tool, under the hood Polyspace actually uses dynamic techniques as part of the 'static' analysis (known as 'Abstract Interpretation'), though this is hidden from the user [12]. The thoroughness of this checking technique requires considerably more processing power than that used by other tools.

some of the vendors have provided their tools for free to open source projects

Hot on the heels of the three main players mentioned above is a new kid on the block: CodeSonar. The code sonar tool is more straightforward and to the point than the other tools on the market, and being fairly new is considerably cheaper.

And there's the rub. Because advanced static analysis tools like these require fairly serious development effort to produce, there is a price tag to match. Typically licenses cost £10000s (that's right, four zeros). The cost of these tools clearly puts them way beyond the home developer. However, some of the vendors have provided their tools for free to open source projects, so if you're involved with an open source project it's always worth an ask. For example see [13]. For commercial software development the benefit is fairly easy to see when you add up the cost of fixing bugs, especially the ones you accidentally ship to the end user. If such tools are used correctly they prevent many bugs from getting into the checked-in codebase in the first place, which hopefully saves money at all stages of testing as well as increasing code quality. This is true of any static analysis tool, but 'Super lints' come into their own during integration as the number of path combinations explode way beyond what is humanly testable.

How to use

Most of the time the sorts of tools listed above are only wheeled out in order to find some hard-to-track-down bug. However I'd like to suggest that using these tools in that way is similar to only running your unit tests once. What you need (for both) is a regression strategy. A simple and very effective approach is to run all of your unit tests under a dynamic analysis tool such as Valgrind and check that they come out clean.

Even doing this as a one off is certainly useful. Apart from anything else it improves the effectiveness of your unit tests. Your code may contain a local memory leak. There is no way to detect memory leaks using test cases alone, but a dynamic testing tool will warn you immediately it detects a problem. But why do it just once? My advice is to make it a standard part of your regression testing. If at some later point somebody introduces a memory leak into the code, or makes some other mistake that doesn't show up in the unit test results, your unit testing procedure will now pick it up. The beauty of this is that you're starting to pick up issues during unit testing that normally don't show up until integration.

Similarly I would recommend applying your static analysis tool at the same level. The feasibility of this second suggestion depends very much on what static analysis tool you are using and whether you are starting with a clean codebase. Lint type tools produce a large number of warnings, most of which are not related to genuine code defects. To get the value out of these tools you need to get used to their way of working and work with them from the outset rather than against them. The most effective use of lint tools is to treat their output as compiler warnings: always aim for a clean build. That way, as soon as any new issues crop up they are immediately visible. It does take a bit of getting used to. The payback however is worth it.

A final point about static analysis tools. Lint and static analysis tools in general are typically very configurable. Often when first using the tools the common gut reaction is to switch off any checks which are reporting false positives. Unfortunately this often robs the tool of its power. There are usually several levels of warning suppression you can go through before resorting to switching off a check completely, including modifying your code. Learn to use the different suppression techniques and maximise the advantage of using the tool. Don't fight it, work with it and it should pay you back handsomely.

Conclusion

The currently available tool offerings, both open-source and commercial, offer an impressive range of abilities and can save huge amounts of effort if used properly. Being too nitpicking can result in loss of productivity – learning to configure your tool to the optimum level is an art and takes some time to get right, but is well worth the effort. Happy debugging. ■

References

[1] http://www.gnu.org/software/libc/manual/html_node/Allocation-Debugging.html
[2] http://duma.sourceforge.net
[3] http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html
[4] http://www.valgrind.org/
[5] http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging
[6] http://en.wikipedia.org/wiki/ProPolice
[7] http://www.misra.org.uk/
[8] *Effective C++ - 50 Specific Ways to Improve Your Programs and Designs*, Scott Meyers, 1998 Addison-Wesley
[9] http://sourceforge.net/projects/splintpp
[10] http://www.nessus.org/nessus/
[11] http://en.wikipedia.org/wiki/Ariane_5_Flight_501
[12] https://tagteamdbserver.mathworks.com/ttserverroot/Download/42825_white_paper_abstract_interpretation.pdf
[13]http://scan.coverity.com/

Don't fight it, work with it and it should pay you back handsomely

Further reading

1. http://www.infoworld.com/article/06/01/26/73919_05FEcode_1.html
2. http://www.infoworld.com/article/06/01/26/74270_05FEcodelint_1.html
3. http://en.wikipedia.org/wiki/Memory_debugger

!(C ^ C++)

Matthew Wilson takes a considered look at C and C++.

When the CVu editor asked me to write this article (instead of the one I'd proposed), I immediately thought that the best approach would simply be to write from the perspective of someone who is an ardent fan of both languages, uses them both on a daily basis, and has strong opinions on each.

This article does not attempt to be a comprehensive look at the differences between the two languages; rather it's a list of the issues, trivial and substantive, that first occurred to me in response to this task. They will inevitably be steeped in my bias as a software consultant (who spends a lot of time helping teams out with projects that are in difficulty), as an avid library writer (who is trying to change the C++ world from the inside out), and as an author (who writes books that attempt to show how advanced techniques may be applied in practice).

In the two main sections I list the things I miss about C++ when writing in C and vice versa, and, where possible, I'll discuss techniques to emulate the missing feature. I also comment on techniques that can be used to ensure code compiles correctly in both languages, something that is essential in the creation of header-only libraries that support both languages. In the final section, I discuss why it is that I persist in using both languages, and discuss briefly some of my criteria for selecting one language over the other for a given task.

10 things I miss from C++ in C

In no particular order, here is a list of the facilities of C++ that I most miss when programming in C. Some are trivial; others less so. Some have workarounds; others do not.

1. for initialiser scope

In C++, I can write:

```
for(int i = 0; i != 10; ++i)
{
  printf("%d\n", i);
}
for(int j = 0; j != 10; ++j)
{
  printf("%d\n", j);
}
```

In C, I cannot. I must instead write:

```
{ /* Some scope: function, if-block, . . . */
  int i;
  int j;
  . . . /* Arbitrary amount of code . . . */
  for(i = 0; i != 10; ++i)
  {
    printf("%d\n", i);
  }
  for(j = 0; j != 10; ++j)
  {
    printf("%d\n", j);
  }
```

This adds more lines to the code, and violates the principle of locality of scope. There's a simple workaround, however: introduce a new scope:

```
{ int i; for(i = 0; i != 10; ++i)
{
  printf("%d\n", i);
}}
{ int j; for(j = 0; j != 10; ++j)
{
  printf("%d\n", j);
}}
```

2. Anonymous arguments

In C++, I can omit the name of an argument that's not used, and thereby be relieved of the compiler – whose warning-level is always set to maximum, of course – nagging me about not having used it.

```
inline int fn(int used, char /* notUsed */)
{
  return used;
}
```

In C, that's not possible, and one must instead reference it in some way or other:

```
inline int fn(int used, char notUsed)
{
  ((void)notUsed);
  return used;
}
```

The **((void)notUsed)** statement is a bit ugly and not all that self-documenting – at least I've met experienced developers who don't get it. Furthermore, being a C-style cast, it is something one likes to avoid in C++, it doesn't jump out of the page even at those developers who do get it, and it precipitates warning from those compilers who are generous enough to be able to warn us about them. (Digital Mars is one.) Changing it to **static_cast<void>(notUsed)** means that the code will not compile for C, and also leads to warnings about meaningless statements from other compilers. (Open Watcom is one.)

My approach is to use a macro, **STLSoft**'s **STLSOFT_SUPPRESS_UNUSED()**, which resolves to an appropriate construct whatever language/compiler is being used. For those *really* fussy C++ compilers, it resolves to a call to a suite of function templates, whose main actor is

```
// in namespace stlsoft
inline void suppress_unused_func(
   T const volatile   &)
{}
```

With this I can rewrite **fn()** in the following way, and have it compile correctly, at all warning levels, in both languages, and with all C/C++ compilers. It also stands out, and is readily **grep**able.

```
inline int fn(int used, char notUsed)
{
  STLSOFT_SUPPRESS_UNUSED(notUsed);
  return used;
}
```

MATTHEW WILSON

Matthew is a software development consultant, columnist, and author of *Imperfect C++* and *Extended STL*. He is the creator of the FastFormat, Pantheios and STLSoft libraries. Matthew is curently working on *Breaking Up The Monolith: Advanced C++ Design Without Compromise*.

3. Dynamic struct initialisation (for local scope)

This one can be a real pain. Here's some code from an example that was recently added to the **Pantheios** logging library. I'd written it in C++, probably because Pantheios is primarily a C++ library, but once it was written I decided, with an hour to go before release time, to rename it to `.c`. Whenever possible I like to write in C to verify that libraries that purport support for both C and C++ actually do so. Here's an extract from the succinct C++ version:

```
{
  char buff[101];
  pan_beutil_time_t tm  = {
     STLSOFT_NUM_ELEMENTS(buff), 0, buff, NULL };
  size_t n = pantheios_util_getCurrentTime(
     &tm, 0);
  printf("time with 0 flags: %.*s\n", (int)n,
     buff);
}
```

When compiled as C, the compiler applies the rules of C and balks on the dynamic initialisation of the **tm** instance. So it has to be rewritten as:

```
/* Using local time with the default
representation for the platform */
{
  char                 buff[101];
  pan_beutil_time_t    tm;
  size_t               n;

  tm.capacity      =   STLSOFT_NUM_ELEMENTS(buff);
  tm.len           =   0;
  tm.str           =   &buff[0];
  tm.strftimeFmt   =   NULL;

  n = pantheios_util_getCurrentTime(&tm, 0);

  printf("time with 0 flags: %.*s\n",
         (int)n, buff);
}
```

The extra verbiage adds nothing to readability. In this case, there's no workaround: you just have to wear the extra SLOCs. Of course, I would want this only for local-scope: one of the nice features of C is that there is no dynamic initialisation of non-local structures.

4. Class/structure/union/enum name scope

That hasty rewrite had a positive effect, in that it revealed a defect in the definition of the structure **pan_beutil_time_t**. This is due to one of C's irritating, albeit obviatable, quirks, whereby the scope of the name of an enumeration, structure or union type is the scope of the type. Given the structure

```
struct pan_beutil_time_t
{};
```

any C code that references it has to use the name **struct pan_beutil_time_t**, rather than **pan_beutil_time_t**. In C++, you just simply refer to **pan_beutil_time_t**. The workaround, which I (believed that I) always use to define structures (and unions, and enumerations) for libraries that have to support both C and C++, is as follows:

```
struct pan_beutil_time_t
{
  . . . /* Members */
};
#ifndef __cplusplus
typedef struct pan_beutil_time_t
   pan_beutil_time_t;
#endif /* !__cplusplus */
```

The way to find out whether you've forgotten to do so is, as in this case, to compile in C.

5. Namespaces

I make sophisticated use of namespaces in C++, but in C the only thing I really miss about namespaces is protection from shortsighted C library writers (or giant software companies) who write libraries containing name definitions such as:

```
#define FOR        if(0); else for
#define BLOCK      (UINT)65536
#define INFINITE   0xffffffff
int GetClass(void);
```

and their ilk. Namespaces do, of course, have a great many powerful uses, but in C you're pretty ok as long as you simply choose sensibly and predictably unambiguous names, and then use those names to prefix all the globally visible names, as in:

```
#define ACMELIB_BLOCK_QUANTUM    (65536u)
#define ACMELIB_INFINITE_TIMEOUT (0xffffffff)
int AcmeLib_GetClass(void);
```

Note the absence of an **ACMELIB_FOR**: that's a stupid idea, even without the stupid name.

6. Overloading

Consider the function **b64_encode2()**, from the **b64** library.

```
size_t b64_encode2(
  void const* src
, size_t      srcSize
, char*       dest
, size_t      destLen
, unsigned    flags
, int         lineLen /* = 0    */
, B64_RC*     rc      /* = NULL */);
```

The function **Base-64** encodes a block of memory described by **src** and **srcSize** into a character buffer described by **dest** and **destLen** according to the given flags. Users can optionally specify a line length and/or the address of a variable to receive a return code in the case of encoding failure. In C++ we might implement four overloads, with five, six and seven parameters to cover the four permutations.

```
// in namespace b64
size_t encode(void const* src, size_t srcSize,
   char* dest, size_t destLen,
   unsigned flags); // 5
size_t encode(..., int lineLen); // 6
size_t encode(..., B64_RC* rc);  // 6
size_t encode(..., int lineLen, B64_RC* rc);// 7
```

In C, callers indicate their desire for the 'default' parameters by specifying **0** and **NULL**, respectively.

Contrast this situation with the function **cstring_createLenEx()** from the **cstring** library:

```
CSTRING_RC cstring_createLenEx(
  struct cstring_t* pcs
, char const*       s
, cstring_flags_t   flags
, size_t            cch
, void*             arena
, size_t            capacity
);
```

Amongst other things, this function allows a **cstring_t instance** to be created from a custom memory arena, such as a fixed block of stack memory. In this case, the arena and capacity parameters are *both* required when specifying the memory block.

Where such flexibility is not required, the arena parameter should be **NULL** *and* the capacity parameter should be **0**; specifying either without the other is invalid. Were **cstring** to have a C++ API, we would define two overloads, of four and six parameters, to clearly connote this relationship:

```
// in namespace cstring
CSTRING_RC create(
  struct cstring_t* pcs
, char const*       s
, cstring_flags_t   flags
, size_t            cch
);
CSTRING_RC create(
  struct cstring_t* pcs
, char const*       s
, cstring_flags_t   flags
, size_t            cch
, void*             arena
, size_t            capacity
);
```

In C, the best we can do to avoid any erroneous suggestion that one parameter is valid without the other is to specify an additional API function, **cstring_createLen()**, which takes only four parameters.

```
CSTRING_RC cstring_createLen(
  struct cstring_t* pcs
, char const*       s
, cstring_flags_t   flags
, size_t            cch
);
```

7. Destructors for guaranteed resource cleanup

If you're using C++, you know about *resource acquisition is initialisation* (RAII), even if that's not what you call it. It is made possible by C++'s support for deterministic destruction, arguably C++'s most important feature, and one whose lack is a fatal flaw to many, supposedly more modern, languages.

Even when there's not a pre-written class for a given resource, you can apply RAII in C++ with components such as STLSoft's **scoped_handle**:

```
int main(. . .)
{
  int panres = pantheios::init();
  if(panres < 0)
  {
    . . . // report and quit
  }
  else
  {
    stlsoft::scoped_handle<void>
       scoper(pantheios::uninit);
    . . . // application code
  } // pantheios::uninit() automatically
    // called here
  . . .
}
```

This code guarantees that **pantheios::uninit()** is called regardless of what 'application code' does (except if it calls **exit()**), at the end of the block in which scoper is defined.

In C, one must decide the exact point at which to invoke a cleanup function, and consequently be at the mercy of premature returns.

```
int main(. . .)
{
  int panres = pantheios_init();
if(panres < 0)
{
  . . . /* report and quit */
}
else
{
  . . . /* application code */

  if( . . . )
  {
    pantheios_uninit(); /* Don't forget! */
    return EXIT_FAILURE;
  }
  else
  {
    . . . /* more application code */

    if( . . . )
    {
      pantheios_uninit(); /* Don't forget! */
      return EXIT_FAILURE;
    }
  . . .
```

If any part of 'application code' executes a return without also invoking **pantheios_uninit()**, the library uninitialisation will not occur.

One approach to deal with this is to impose a single return call per function. In many cases that can work well, but the exceptions to the rule are cumbersome indeed. In the case above, we can easily imagine setting **panres** to **-1** at the start of **main()**, and then checking it at the end, and only invoking **pantheios_uninit()** if it's non-negative. But consider when we are using two, three, four, … ten libraries, all of which need to be initialised in the same manner. It's too easy to render such code defective during maintenance, either by forgetting the right uninitialisation sequence, or by adding in a premature return and bringing down the whole house of cards.

My approach in C for this problem is to use layered functions, as in:

```
/* file: my_application.c */
int main_application(int argc, char** argv);
/* . . . application code . . .*/

int main(int argc, char** argv)
{
  int res = pantheios_init();
  if(res < 0)
  {
    . . . /* report and quit */
  }
  else
  {
    res = main_application(char, argv);
    pantheios_uninit();
    return res;
  }
}
```

When multiple libraries must be initialised, more layers may be added by interleaving in more initialisation function layers. To be sure, it's more code. But it's a heck of a lot more proof from maintenance damage.

Naturally, if you've a lot of these, and they're all structurally and semantically conformant (they each take 0 parameters, and return a negative integer on failure), you can be a bit smarter and define a layering API initialisation library, to give something like the following main application code:

```
static const layered_api_entry apis[] =
{
  { pantheios_init, pantheios_uninit },
  { shwild_init, shwild_uninit },
  . . .
  { AcmeLib_init, AcmeLib_uninit }
};

int main_application(int argc, char** argv);
/* . . . application code . . .*/

int main(int argc, char** argv)
{
  return layered_api_main(&apis[0],
     NUM_ELEMENTS(apis), argc, argv,
     main_application);
}
```

I'll leave it as an exercise for the reader to implement it – including handling different error return semantics, initialisation flags, and calling conventions – and release it as open-source.

8. Strings and containers

Although a huge topic, the point to be made about these two items is simple: the lack of such libraries in the C standard library and their presence within the C++ standard library means that, absent other important factors, writing a library or application that makes much use of either is going to be a lot easier in C++. Furthermore, because they're available in the *standard* library, you can worry a whole lot less about coupling in such a C++ project than in a corresponding C project that also requires its users to download and build *AcmeLib's C Containers Library*.

I would remark on one neat (read: perverse) trick a colleague and I used recently. We're building a suite of tutorial material about writing applications in C and C++ using only existing standard and open-source libraries, and wanted to use C as the implementation language of an early example for pedagogical reasons. The rub was that we needed to store a sequence of file paths for later processing. The trick we came up with was to use the `cstring` library to store the paths as a single string, separated by the `|` character (something that cannot occur in a path). Then we processed the 'list' by tokenising it (`via strtok()`) and operating on each retrieved path in turn.

9. Transparency of application code

C++ is hugely more expressive than C. Consider the following two snippets of application code that log a function entry and its parameter values. First, in C:

```
int connect_to_peer(struct in_addr const* addr)
{
  pantheios_logprintf(PANTHEIOS_SEV_DEBUG
     , "connect_to_peer(%u.%u.%u.%u)"
     , (NULL == addr) ? 0 : (
      (addr->s_addr & 0x000000ff) >> 0)
     , (NULL == addr) ? 0 : (
      (addr->s_addr & 0x0000ff00) >> 8)
     , (NULL == addr) ? 0 : (
      (addr->s_addr & 0x00ff0000) >> 16)
     , (NULL == addr) ? 0 : (
      (addr->s_addr & 0xff000000) >> 24));
  . . .
```

Now in C++:

```
int connect_to_peer(struct in_addr const* addr)
{
  pantheios::log_DEBUG("connect_to_peer(",
     addr,  ")");
  . . .
```

For things like this, there's just no contest. The former is verbose and opaque, and as a consequence significantly detracts from transparency (the quality of how easy it is to understand a piece of code in order to change it). The latter is the opposite, intrudes minimally into the function's implementation, and is more efficient when the debugging level is switched off.

10. Beauty (aka expressiveness)

While it's certainly true that C++ gives you much broader capacities for writing inscrutable junk, it also affords a level of expressiveness that's simply not achievable in C (or in plenty of other C-family languages, for that matter). If you'll permit me the tasteless indulgence of taking content from my second book, *Extended STL, volume 1* [1], here's an example of a highly expressive piece of code that deletes files from the current directory.

```
using unixstl::readdir_sequence;
readdir_sequence  entries(".",
   readdir_sequence::files);
std::for_each( entries.begin(), entries.end()
             , ::remove);
```

It's almost possible to read this as natural language: 'for each entry in the sequence of files in the current directory, remove it'.

The alternative in C is around fifteen lines of code, almost all of which is boilerplate involving `opendir()`, `readdir()`, `closedir()`, `S_IFREG` and `S_IFMT`, and a whole lot more gunk that makes for very poor reading. (You can check out the code, and the accompanying discussion, by downloading the book's Prologue, which is freely available from http://www.extendedstl.com)

10 things I miss from C in C++

In no particular order, here is a list of some aspects of C that I most miss when programming in C++.

1. (Absence of) overloading

Overloading is great. Except when it isn't. (It's always better than default parameters, of course, but no-one's ever going to give back weapons once they've got them. Actually, they're necessary for a whole manner of techniques for working around compiler/language defects, so there's no question of giving them up.) Anyway, back to overloading.

This one can be a surprisingly inscrutable problem. Consider the following code:

```
// file: my_app_main.cpp
namespace
{
  void onBadArg(int bExit, char const* reason,
     int invalidArg, int argc,
     char const* const* argv);
  void showHelp();
}
int main(int argc, char** argv)
{
  // process the command-line
  if(1 == argc)
  {
    onBadArg(1, "insufficient options; use --help
       for usage", -1, argc, argv);
  }
}
namespace
{
   void onBadArg(int bExit, char const* reason,
      int invalidArg, int argc,
      char const** argv)
  {}
  void showHelp()
  {}
}
```

This compiles fine, but balks at linker time, rabbiting on about something like

```
"absence_of_overloading.obj : error LNK2019:
unresolved external symbol "void __cdecl `anonymous
namespace'::onBadArg(int, char const*,int,int,char
const* const*)"
(?onBadArg@?A0x85adb1b1@@YAXHPBDHHPBQBD@Z)
referenced in function _main"
```

What are you *talking* about? says the unfortunate time-poor developer, who goes on to spend many head-scratching minutes before asking a colleague to break the mental torment. Said colleague instantly spots the missing **`const`** in the definition of **`onBadArg()`**. C++'s overloading facility is a definite hindrance in cases such as this. What can we do?

Well, as readers of my first book, *Imperfect C++*[2], will know, there's a whole lot you can do with the magic of C++'s backwards-compatibility mechanisms. In this case, we can use the **`extern "C"`** linkage specifier to instruct the compiler that we don't want overloading of our helper functions, as in:

```
// file: my_app_main.cpp
namespace
{
  extern "C"
  {
    void onBadArg(int bExit, char const* reason,
       int invalidArg, int argc,
       char const* const* argv);
    void showHelp();
  }
}

int main(int argc, char** argv) ... // as before

namespace
{
  extern "C"
  {
    void onBadArg(int bExit, char const* reason,
       int invalidArg, int argc,
       char const** argv)
    {}
    void showHelp()
    {}
  }
}
```

Now when you compile, you're given a much more informative response, and by the compiler rather than the linker, along the lines of

```
"absence_of_overloading.cpp(24) : error C2733:
second C linkage of overloaded function
'`anonymous-namespace'::onBadArg' not allowed
    absence_of_overloading.cpp(23) : see
declaration of '`anonymous-namespace'::onBadArg'"
```

If you enclose the declaration but forget the definition, you'll still get 'successful' compilation, and only find out at link time. But at least it'll be clearer, along the lines of

```
"absence_of_overloading.obj : error LNK2019:
unresolved external symbol _onBadArg referenced in
function _main"
```

The absence of C++ name mangling should give you a better clue about what the problem is.

2. Compilation times

This one's a no-brainer: we all know that C++ has much longer compilation times, exacerbated of late with the increasing use of templates. When building the (legion) object libraries of Pantheios on my main 64-bit Linux box, it's impossible to follow the compilation of the C compilation units with the eye, while the C++ ones take at least a second each.

3. Portability

The C dialects supported by the various popular compilers differ very slightly in comparison to the dialects of C++. In all the libraries I write, and port across a large number of compilers and platforms, I have found very few problems in ensuring the portability of the C, as long as I stick with standard (C89 or C99, whatever the case may be).

By contrast, I'd guesstimate that 50% of my effort in writing and maintaining my main C++ libraries – **`FastFormat`**, **`Pantheios`**, **`STLSoft`**, **`VOLE`** – is spent ensuring portability and inventing cunning, fatuous, and philosophically worthless workarounds. That's pretty sad when you think about it.

With compilation times, these two explain why so much of Pantheios, a C++ logging library whose selling points include employing C++'s type-system to yield 100% type-safety (something of passing importance in a logging library), is written in C!

4. Interoperability (ABI)

In C (with very few exceptions), I can use any C compiler on a given platform and interoperate with C, C++, C#, D, Java, Perl, Python, Ruby and many other languages and technologies. By contrast, in C++ (with very few exceptions), I can interoperate with C (produced with *most* compilers) and with C++ written only with the same version of the same compiler.

Consequently, a host of C++ language features (some good, some not so much) are not available when writing/using libraries that are going to be connected together via link-time or runtime, rather than compile-time, mechanisms. Multiple-inheritance, exceptions, RTTI, static objects (local and non-local), and templates are all out. Virtual methods can be achieved only with considerable effort (see Chapter 8, 'Objects Across Borders', from *Imperfect C++*). Name-mangling schemes are different between different compilers. Calling conventions (and their concomitant symbol name decoration) are at best troublesome; at worst some compilers support a different set of calling conventions to others.

What this means is that you either have to be sure that you'll always be able to generate all your libraries (static and dynamic) from the same compiler (or one of the small set of compatible compilers), or, as I prefer to, eschew the use of C++ features (with the occasional exception of portable vtables, when the benefit warrants the effort) and stick with a C-compatible interface.

5. Discoverability and transparency of library code

Ok, let's get real here for a minute. Many C *and* C++ libraries lack something in discoverability (the quality of how easy it is to understand a piece of code in order to use it), and I'd guess that the vast majority leave much to be desired in transparency. (Discoverability tends to be more associated with the interface, and transparency with the implementation, although the demarcation is not absolute.)

In my opinion, however, C++ libraries tend to be considerably worse than C libraries of the same/equivalent complexity in both aspects. Of course, it's difficult to nail down exact causes with third party libraries produced by disparate groups of authors. However, if you take a look at the interfaces and implementations of any standard library implementations you'll probably see some merit in what I'm saying. And in reviewing my own work I can definitely see a difference in levels of transparency.

Given the portability issues in C++, particularly when templates are mixed in, I can see an argument to be forgiving of messy implementations in C++ libraries. However, no such quarter should be asked or given when it comes to crappy C libraries – if you can't comfortably inspect the C libraries you're using, chuck them and try another.

6. WYSIWYG: lack of exceptions

Ok, don't get me wrong here. I am a big fan of exceptions, as long as they're used sensibly, that is, for the indication of *unusual* conditions that are *within* the purview of the software design, not for the transition between common program states, nor for the indication of design violations.

Having said that, it's still something of a pain sometimes when one is spending so much mental effort attempting to see all the hidden exit paths from a given piece of source. Whenever you look at any non-trivial piece of C++, at minimum you have to ask yourself what's the consequence of `std::bad_alloc` being thrown. If that's your only concern, it's not that bad because an out-of-memory condition usually means that your process has entered a practically unrecoverable state. As long as you have some high-level catch clause to detect and attempt to report it, you can call your job done in that regard. (I use the word 'attempt' advisedly, since nothing's guaranteed when there's no more memory left. But you have to make a good-faith attempt, preferably using a very low-impact logging call.)

In most cases you have a lot more to consider than the low-likelihood, simple-response case of out-of-memory. Anytime you use a library (that uses a library, which uses a library) that might throw exceptions representing recoverable conditions, the complexity of your code rises dramatically. What exception(s) might be thrown? Where? Under what conditions? With what mitigations, or alternate actions?

What might seem a clear distribution of normal vs exceptional semantics with one component/library at one level of abstraction can be but a portion of a baffling spectrum of complexity when it is considered in concert with several others, from a vantage point of a few levels of abstraction above.

Conversely, in C the question of whether the execution path encounters a given code block can be reasoned based entirely on the code that one sees *at that level of abstraction*. To be sure, the manual receipt, translation and transport of return-code conditions between different layers of abstraction in C presents a challenge of similar magnitude, is a lot more work, and is all too easy to do incompletely. Neither situation is particularly good: just different negatives.

7. WYSIWYG: Lack of dynamic initialisation (well, ... only a bit)

In C, the only actions that may be said to occur outside the scope of `main()` are the invocation of the termination functions registered at `atexit()`, and the closing of any open streams. And even these two are skipped if you invoke `_Exit()`, though that's not something that's recommended except in extremis.

What this means it that you don't have to wonder what kinds of things might be happening in your program far away from `main()` (and your ken). By contrast, in C++, you can find yourself in all kinds of link-time trouble because some library designer has thought fit to whack in a few globals for the sake of specious expediency.

This is one reason why writing libraries in C (with or without a nice header-only C++ API) is a **good thing**.

8. WYSIWYG: lack of invisible costs

C++ has a reputation for being inefficient, particularly with respect to its older brother. This does not have to be so; it's just that it usually is. One reason is that it's very easy to hide large amounts of functionality within seemingly innocuous statements, and lots of developers don't seem to want to think about what's going on inside the components that they're using (which is pretty reasonably, after all; library designers should be serving them better).

In C, I have to name functions in order to carry out actions. The things that happen are, with a few rare cases, right in front of me in my code. If I call a function, `Xml_LoadFileIntoDom()`, I have a pretty good notion of the scale of its costs. In C++, you are at the mercy of the library designers' whims (and experience), since it's quite possible for you to write innocuous statements in your library code and have huge amounts of stuff done without your having a clue from looking at the code.

Consider the `IOStreams`, one of my pet hates, for an example. The insertion statement in the following code incurs seven memory allocations (with VC++ 9):

```
std::stringstream ss;
std::string       journal = "CVu";
int               year    = 2008;
char const*       answer  = "yes";
ss << "Write for " << journal << " in "
   << year << "? The answer is " << answer;
```

This is ludicrous. Compare that to C's

```
char        result[100];
char const* journal = "CVu";
int         year    = 2008;
char const* answer  = "yes";

sprintf( &result[0]
       , "Write for %s in %d? The answer is %s"
       , journal, year, answer);
```

The not-inconsiderable downside to the latter, as you well know, is that it's not type-safe, and it can overflow the receiving buffer. Naturally, there are better alternatives to both, but that'll have to wait for another article.

9. WYSIWYG: lack of operator abuse

I'm as guilty as the next programmer about having abused the meanings of operators. Early on in my career, I did some truly terrible things with overloaded operators, and even now some path manipulation components in the `STLSoft` libraries overload the division operator for the purposes of path concatenation. The older I get, the less I like it, and I'm seriously considering deprecating/removing all operator overloading in all of my libraries in the near future. There's just no net benefit in looking at code such as the following

```
p = d / f;
s = t + u;
```

and not knowing whether these are arithmetic operations or a path composition (of directory `d` and file `f`) and a string concatenation (of string `t` and character `u`). Sure, it's enormous fun to write libraries that do this, and a fair amount of fun to use them. Until you forget how to use them, that is, and then the specious expressiveness quickly becomes obfuscation. Since I spend most of my commercial time looking at other people's code, and most of my 'free' time looking at my own code, I value transparency more highly.

We may assume that operator abuse was legitimised by the inclusion of the woeful `IOStreams` library into the C++ standard library. But now we also have to contend with a profusion of string concatenation, path manipulation, formatting, and all manner of expression templates; I know because I've done my fair share. But they just add to dialecticism and reduce maintainability. Ho hum.

10. WYSIWYG: lack of mutable (non-const) reference parameters

In common with the other four items, this one also has to do with C++'s propensity for delivering (unwanted) side effects. And, like most of the items in this section, it's a facility that I both abhor and use to good ends in producing C++ libraries that have the characteristics I deem necessary and appropriate. Such is life.

Nonetheless, in 'normal use', mutable reference parameters are just a pain, and they dramatically *reduce* code transparency. Consider the `integer_to_string` function suite from STLSoft that performs the conversion of any integer type into a string with very high efficiency. The four-parameter overload provides the number of converted characters used in the given string, allowing for the following:

```
std::string fast_i2s(long long value)
{
  char     num[21]; // 21 is large enough for
                    // any 8-64 bit integer value
  size_t   n;
  char const* r =
     stlsoft::integer_to_string(&num[0],
     STLSOFT_NUM_ELEMENTS(num), val, n);
  return std::string(r, n);
}
```

There's one problem with this. When looking at the code it's not immediately obvious that **n** is an out-parameter of **integer_to_string()**. Sure, we can look at the fact that it's not initialised, and make an inference from that. (Some compilers aren't so clever, and you may find yourself having to initialise it to **0**, which totally blows that out of the water.) Or we can look at the documentation of **integer_to_string()**, which costs us a couple of minutes and takes our attention away from the code we're trying to understand/modify.

The issue is that an out-parameter codified as a mutable reference is a pointless thing. It reduces discoverability because when you look at the interface of a function such as **integer_to_string()** you must work out (through the documentation or the implementation) whether the parameter is in-out, or just out. And it reduces transparency of client code in the way described above.

All that for the specious feeling of safety you get from 'knowing' that a reference can't be **NULL**. (Which of course it can, as quickly and easily as you can type **static_cast**.) I believe that out-parameters in C++ should always be coded as a pointer, because the benefits to transparency are obvious:

```
size_t      n;
char const* r =
   stlsoft::integer_to_string(&num[0],
   STLSOFT_NUM_ELEMENTS(num), val, &n);
```

Now we don't need to look any farther afield to understand what's going on.

Why I use both

When I'm creating application code, I always select the C++ option in my code generation tools, for the following reasons:

- Expressiveness
- Resource management
- Transparency (of application code)

The situation is less clear-cut when it comes to library code, and is much more of an 'it depends' situation. Absent any overriding reasons, I'm going to choose C. But there are quite a number of such reasons. Table 1 lists the details for my main open-source libraries. The main reasons why I've used C++ for implementing some of them are:

- Need of strings and, especially, containers
- Need to have complex internal data structures
- Resource management
- Use of existing façades for non-trivial APIs

Some final things to think about:

- If you use exceptions, you cannot properly manage resources without RAII. (This is of relevance to users of the half-baked younger siblings of C and C++.)
- Exceptions for unrecoverable conditions are the absolute best choice mechanism, but the case is less clear in other cases: you're left to choose between certainty that something will be reported even though you might not know what that is while you're designing your software, or clarity in manual error-handling with the likelihood that you'll be introducing defects for any that you overlook.
- Highly expressive C++ libraries must be efficient.
- The discoverability of C++ libraries tends to be lower than C libraries, meaning that users will be looking inside them more often, meaning that the transparency needs to be higher. (But, C++ libraries also tend to have lower transparency. Which is a problem.)
- If you want C programmers to use your libraries, just providing a C API over a C++ implementation might not be enough. They may want '100% C'.

I love both these languages, and have (so far) made a challenging career from knowing and applying them well. The key is in knowing the appropriate language, and features, for the given circumstance. I hope that this article has helped you a little with that. ■

References

[1] Wilson, Matthew (2007), *Extended STL, volume 1*, Addison-Wesley (see also http://www.extendedstl.com).

[2] Wilson, Matthew (2004), *Imperfect C++*, Addison-Wesley.

Library	Language(s)			Reasons for choice of implementation language
	Client	API(s)	Implementation	
b64	C, C++	C, C++	C	Portability, Transparency, Efficiency
CLASP (not yet released)	C, C++	C, C++	C	Portability
cstring	C	C	C	Portability, not a library used by C++
FastFormat	C++	C++	C++	Containers, Complexity in implementation (parsing, memory mgmt, etc.)
Pantheios	C, C++	C, C++	C, C++	C: Compile Times C++: Use of existing libraries (for synchronisation, OS façades, memory)
recls	C, C++	C, C++	C, C++	Use of existing libraries (for synchronisation, file-system enumeration, memory)
xCover	C, C++	C, C++	C++	Containers, Complexity in implementation (classes, memory mgmt, etc.)
xContract	C, C++	C, C++	C	Very simple implementation
xTests	C, C++	C, C++	C++	Strings, Use of existing libraries (printf-traits)
UNIXem	C, C++	C	C	Portability
VOLE	C++	C++	C++	COM => C++ (unless you're insane)

Python: New Thinking in the Teaching of Programming

Nick Efford and Tony Jenkins recommend Python as a practical first language.

The teaching (or, probably more accurately, the *learning*) of programming is a perennial problem in Computing Higher Education. Here is a subject that lies at the very heart of our discipline, but it is a subject that always seems to cause our students misery and distress. Every year we see students leave Computing for different subjects and, more often than not, it turns out that their struggles with programming are the root cause.

It might seem obvious that the programming language that students learn first sits at the heart of the problem. Many languages have been tried over the years, but few, if any, have had any noticeable impact. There is something of a Catch-22. Some languages, like Pascal or even BASIC, have been designed with the needs of learner programmers in mind. But these languages are not taught because there is no demand in industry for Pascal programmers. So Universities tend to teach languages that are in demand in industry, notably at present Java; unfortunately these are languages that were designed for professional programmers, not for learners. The ideal language would be, obviously, one that is both easy to learn and in demand in the industry.

We believe that Python is that language.

Universities do not change their first programming language lightly. The phenomenon of a 'Language War', where factions of academics debate the merits of their preferred language long into the night, is well known. Surprisingly, we were able to adopt Python without the need for a war. We report here on what happened when we adopted Python as our introductory programming language.

A brief history

The School of Computing at the University of Leeds has taught programming for many years. It has always been one of the first courses that students take, and programming has always been regarded as fundamental to all the degrees offered in the School.

The progression of language has followed a predictable pattern. In the mists of time, Algol was the choice. In the mid-1980s the language of choice became Pascal. This was replaced by C++ in the mid-1990s, and then by Java in about 2002. There is very little evidence that any of these changes altered the students' learning of programming.

In 2004 we dabbled with a new language, Python. This was used as an introduction to the course, before the students moved on to Java. The results of this were promising but, for various reasons, the experiment was dropped. A complete redesign of the degree programmes in 2007 gave the opportunity for a change, and Python was adopted.

We were most pleased with the results. We will not be changing back to Java any time soon.

TONY JENKINS

Tony Jenkins is a Senior Teaching Fellow in the School of Computing at the University of Leeds. A confirmed devotee of all things Pythonic and becoming known as a Python advocate in the UK higher education community, he is really glad that he doesn't have to teach Java any more.

NICK EFFORD

Nick Efford leads the teaching of software engineering to undergraduates in the School of Computing at Leeds. He also teaches computer security at undergraduate and postgraduate level and has researched and taught in the area of image processing and computer vision.

Python in industry and academia

Python is a mature, well-established, very high-level programming language renowned for its clear, readable syntax and for the significant productivity gains [1] that it brings to programmers brought up on lower-level languages such as C or Java. It has been used successfully in countless real-world business applications, including many large, mission-critical systems [2], by companies such as Google, IBM, HP, Disney and Nokia. It is also increasingly popular in academic and industrial research, where it has been used for space shuttle mission design [3], biomolecular modelling [4], the control of large-scale physics simulations and the analysis and visualisation of weather radar data, to give but four examples.

Python has a long history as a language of interest to educators, dating back to 1999's DARPA-funded CP4E project [5] and beyond. Web-based resources have sprung up, dedicated to the teaching of Python in high schools [6], and the language plays a central role in the One Laptop Per Child project [7].

Within the university sector, interest in using Python to teach programming to students of computing is growing rapidly. Besides Leeds, the Universities of Coventry and Glasgow here in the UK are using Python extensively for this purpose, as are a number of institutions in the USA – e.g., UC Irvine, Michigan State University and, notably, MIT [8]. Python is also finding favour amongst those who teach more advanced topics from the computer science curriculum (e.g., natural language processing at Leeds and at the University of Toronto [9], or the semantic web at the University of Maryland).

Rationale for adopting Python as a first language

Programming is unusual among 'academic' subjects in that it does not form a single coherent body of knowledge. To illustrate this, consider the question of what should be the very first thing that a new programmer learns. Arguments could be made in favour of any of:

- How a computer represents data;
- The programming methodology;
- How to declare a variable;
- The mechanics of compilation or interpretation.

and many, many more.

A key issue is the number of concepts that a new programmer must learn before they can produce something [10]. The first program that any novice programmer *sees* is, of course, always the same; it prints **`hello, world`** on the screen. In C that program might be something like:

```
#include <stdio.h>
int main (void)
{
  printf ("hello, world!\n");
  return 0;
}
```

The line that does the printing is obvious enough, although a novice would require some explanation of the need for the mysterious `\n` at the end. But what of the rest of the program? The novice will want to know what `stdio` is all about. Why is `main int` and what is `void` all about? Why is this program returning `0`?

As a small aside, consider for a moment the semi-colon at the end of the `printf` line. Without that small piece of punctuation this program is completely worthless. Edsger Dijsktra [11] has pointed out that this feature, whereby the 'smallest perturbation' renders the program completely useless, is something that makes programming especially difficult to learn.

Of course, few universities now teach C as their main language; Java has occupied that role in most institutions for many years now. The first program in Java can be written like so:

```
public class Hello {
  public static void main (String args [])
  {
    System.out.println ("hello, world!");
  }
}
```

(Note that we are adopting a simplified approach here; the Java purist would argue that we should define a 'proper' class, with a constructor and a method that prints the greeting, and that we should then create an instance of the class in `main` and invoke that method on the instance.)

It is difficult to know where to start in counting the concepts here, even in this simplified, object-free example. As before, spotting the line that does the printing is easy, but the rest of the program is full of mystery. What is a `class`? What are `String args`? And what on earth is `public static void main` all about?

The solution to this conceptual overload is naturally to tell novice programmers not to worry about such things; just copy the program blindly, and assume that everything will work. We find this most unsatisfactory, especially as the smallest error in copying the code can provoke all sorts of arcane messages from the compiler. We would much rather that our students understood all of their programs from the start.

Here is the first program in Python:

```
print 'hello, world!'
```

That's it. There is one concept to understand: a 'command' with an 'argument'. A student who cannot grasp what is going on here is probably doomed to failure from the outset.

Obviously there is far more to choosing a language for teaching than the simplicity of this one program. Having a small and uncomplicated 'Hello, World' isn't useful if more realistic tasks require code comparable in size and complexity to Java code. Fortunately, however, Python remains reassuringly clear and concise as we move on to more advanced topics. Chou [12] has found, for example, that Python implementations of sorting algorithms look almost identical to the pseudocode representations appearing in a standard computer science textbook. Chou has also shown that data structures such as directed graphs have almost trivial representations in Python, in stark contrast to the situation with Java, C++, etc.

Python in practice

In our experience, programming language complexity is not the only factor influencing students' motivation. We therefore established four principles for teaching programming with Python:

1. 'Cool' – programming is cool, and Python is cool. Writing programs to find the average of test scores is not cool; writing a game with some neat graphics is cool.
2. 'Useful' – programming is useful. Programs to carry out artificial tasks should be avoided; programs should address real problems.
3. 'Real' – our programming should use real languages, libraries and environments, that are being used by professional programmers in industry applications.
4. 'Fun' – above all, programming should be fun. If the students found the programming experience fun, hopefully they would be motivated to practice, and hopefully they would not give up.

With these principles in mind, we taught the course in a fairly conventional way. We started with the traditional first program and then introduced concepts such as input/ouput, conditional statements and loops before moving on to look at the structuring of programs using modules and functions. Everything seemed to go rather well, and by the end of the seventh week we found that we had covered most of the basics.

At this point we wanted to enable the students to write something interesting (cool, useful, real and fun), either using the facilities of Python's standard library or those of a separate package. As a good example of something both useful and real, we chose PySQLite – a module that supports use of the SQLite [13] embedded database from within Python programs, conveniently available as standard with Python 2.5. We also chose Pygame [14], a third-party, cross-platform framework for game and multimedia programming, to fulfil the requirement for something that was cool and fun.

After some lectures and exercises covering the use of both PySQLite and Pygame, students were asked to do a four-week project, to be done individually or in pairs, using one of these options or something else entirely. Although most opted to use Pygame, we also had a few students developing networked applications using the Twisted [15] framework, or web applications using Django [16], plus one student who created an application for his Nokia mobile phone (discussed below). Just before Christmas, students took part in a 'programming showcase' where they demonstrated their finished projects to tutors and members of the School's Industrial Advisory Board.

Teaching for the second semester proceeded along similar lines, albeit covering different topics (object orientation, test-driven development, debugging and some forays into basic web and GUI programming). Once again, students were given the opportunity for an extended piece of project work, to be completed over the Easter break.

Results

Here, we look at some specific examples of project work from the first semester of the course and consider the outcomes of switching to Python.

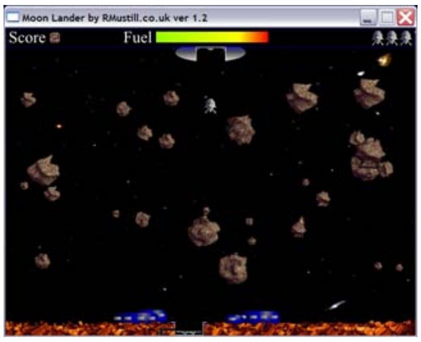


Figure 1

Figure 1 is a screenshot from a lunar lander game. The author of this game had considerable prior experience of programming and therefore had little difficulty in implementing the necessary game logic. This left him with plenty of time to polish the game by adding music and implementing impressive visual effects such as spinning boulders and dramatic explosions.

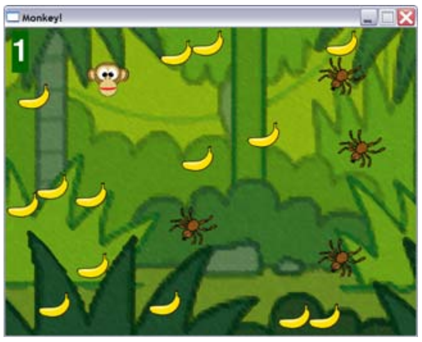
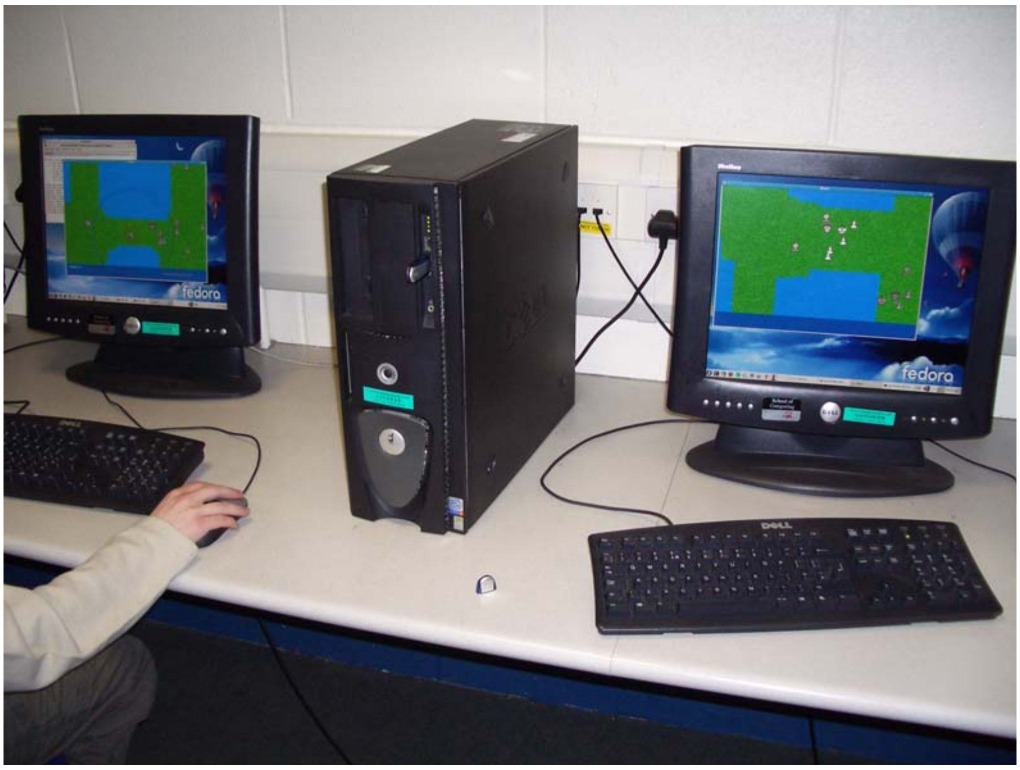


Figure 2

Figure 2 shows the monkey game, a much simpler offering from a pair of less confident students with no prior experience of programming. Despite their relative lack of confidence they were nonetheless able to implement a largely functional game in which the monkey can be moved around the screen using the cursor keys, eating bananas and avoiding the spiders in order to score points. These students would not have fared well in earlier, Java-based incarnations of this course, and would certainly not have been able to reach the level of achievement seen here.

Figure 3



Most students were proud of what they had accomplished in their projects after only ten weeks of learning Python. Colleagues and members of our Industrial Advisory Board were visibly impressed by the work that was demonstrated to them, and by the level of enthusiasm shown by the students. By the end of the year, fewer students were failing the course and fewer were complaining about the difficulty of programming.

Figure 3 shows a turn-based, two-player strategy game running over the network. This was implemented by a pair of very keen programmers and was perhaps the most impressive project from a purely technical standpoint. The details of the map on which the game is played are passed over the network in compressed form, and the students developed their own network protocol to handle this and the movement of player pieces.

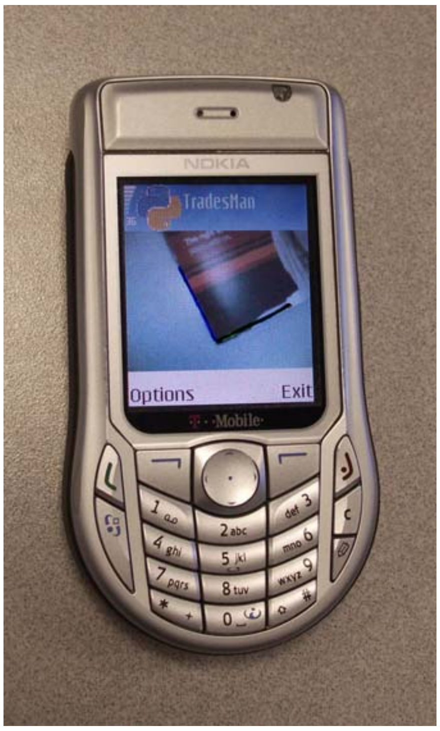
Finally, Figures 4 and 5 show a simple Python application running on a Nokia Series 60 mobile phone. The application captures an image using the phone's built-in camera, allows the user to draw two lines over the top of this image, then computes and displays the angle between the two lines. The author of this application was repeating his first year, having struggled with programming originally. The project work was highly motivating for him and, thus stimulated, he went on to achieve good results at the end of the year.

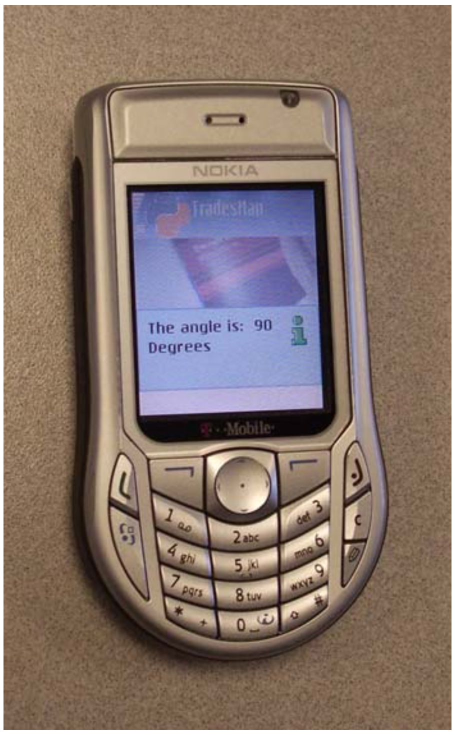
Conclusions

Python is no 'silver bullet', and we certainly do not claim to have solved the perennial problem of teaching students how to program. Those who struggle with the level of discipline and precision required in programming, or who cannot grasp the concept of breaking down a problem into smaller pieces in order to solve it, will continue to find programming difficult. But too many students simply give up when learning Java or C++ because they struggle with these issues in addition to struggling with the language itself. Python succeeds here because it puts far fewer syntax-related obstacles in the way of the novice.

The high-level nature of Python also allows ideas to be turned into working code with much less effort – making it possible for the keen programmer to do truly impressive things in a short space of time, but also allowing the less confident student to accomplish something satisfying. Nowhere was this illustrated more clearly to us than in the standard reached by our students in their project work. We are seeing greater confidence and a higher degree of motivation in our students as a result of switching from Java to Python, and we hope that other institutions will follow our lead. ■

Figure 4





Notes and references

1 http://mail.python.org/pipermail/python-list/2004-February/249707.html
2 http://www.python.org/about/quotes/
3 http://www.pythonology.com/success&story=usa
4 http://www.pythonology.com/success&story=mmtk
5 http://www.python.org/doc/essays/cp4e/
6 http://openbookproject.net/pybiblio/
7 http://laptop.org/
8 http://www-tech.mit.edu/V125/N65/coursevi.html
9 http://www.cs.toronto.edu/~gpenn/csc401/
10 We have the concept of 'something that they would be happy to show their mother' to illustrate the minimum standard that we would like new programmers to reach as quickly as possible.
11 EW Dijkstra, 'On the Cruelty of Really Teaching Computing Science', *Communications of the ACM*, 1989
12 PH Chou, 'Algorithm Education in Python', *Proceedings of the 10th International Python Conference*, 2002 (http://newport.eecs.uci.edu/~chou/py02/python.html)
13 http://www.sqlite.org/
14 http://www.pygame.org/
15 http://twistedmatrix.com/trac/
16 http://www.djangoproject.com/

A Groovy Example: Mail Merge Made Easy

Peter Pilgrim demonstrates a Groovy way to automate your admin tasks.

In this article, I will explain how the Groovy language can act as a really neat scripting language tool, which you can use to perform systems administration tasks. I'll demonstrate with an example program that sends a farewell letter to a list of recruitment agencies. I will explain how to read data from a Comma Separated Value file (in this case, the recipients' address details) and use the default template engine to insert it into the standard text. Groovy, like Java, can connect to the Google Mail SMTP server, so the finished letter is sent using the JavaMail API.

Ingredients

For this recipe, you will need the following ingredients:

- Groovy Development Kit 1.5 or above
- JavaMail and Bean Activation Framework JARs
- A Google Mail Account
- Spreadsheet CSV file
- Java Development Kit 1.5 or above
- Groovy Console for Editing (included with GDK)

Groovy is based on Java, so therefore it helps to already have the Java Development Kit installed on your machine. Point your browser to http://groovy.codehaus.org, in order to locate and download the Groovy Development Kit. Follow the instructions in the package to install and run it. For Windows XP, download the ZIP file and my advice would be to unzip the distribution under folder called `C:\opt`, in order to avoid problems with folder names that contains spaces e.g. `C:\Program Files`. Download the JavaMail 1.4 and the associated JavaBeans Activation Framework jars separately. If you already have the GlassFish application server then you can get JARs from that distribution. You also need a Google Mail account in order to send the resignation email using the scripts here. You need a spreadsheet that acts as a list of contacts (e.g. recruitment agents), first and last names, email, street, address columns in Comma Separated Value format.

Configuration

Once you have Groovy up and running, we need to configure the start up in order to find the extra JARs. We alter the default classpath for Groovy start-up by creating a new file. Under Windows XP this is called `C:\Documents and Settings\YOUR-USER-NAME\.groovy \postinit.bat`.

```
REM postinit.bat
set STARTER_CLASSPATH=%STARTER_CLASSPATH%;
\%GLASSFISH_HOME%\lib\mail.jar
set STARTER_CLASSPATH=%STARTER_CLASSPATH%;\
%GLASSFISH_HOME%\lib\activation.jar
```

In the Groovy 1.5.4 on my machine, currently I can also install the JARs under the folder `C:\Documents and Settings\`**`YOUR-USER-NAME`**`\.groovy\lib`, without using the above post initialisation script. Whatever works for you, as long as it works.

About Groovy

The Groovy language is a dynamically typed scripting language and it is a lot more flexible than Java. The language was designed to give developers like you and me seamless integration into existing Java libraries and code. Being a dynamically typed language, like JavaScript, Python and Ruby it supports the notion: if it walks like a duck and if it quacks like a duck, then it probably is a duck!

Linux and Mac

The automatically included JARs are set up a differently to the Windows version described in the main article.

Create a `.groovy/lib` directory in your home directory, any JARs found in this directory will be automatically included in the **`CLASSPATH`** when you run Groovy from the command line. The `.groovy/lib` directory is disabled by default; please enable it in the configuration file `$GROOVY_HOME/conf/ groovy-starter.conf`

Hopefully this advice also sorts out Mac users.

There are some other similarities with scripting language brethren, not surprisingly. Everything is an object in Groovy. Java primitives are silently converted into their equivalent immutable Wrapper objects, at least most of the time. Groovy assumes you want accurate decimal numbers in your mathematics and defaults to **`java.math.BigDecimal`**. So allow me to quickly demonstrate this with two other very important Groovy features: closures and GStrings.

```
def val = 0.0                                 // 1
10.times( { it -> val += 0.1 } )              // 2
println "This is the answer val=$val";        // 3
```

The first line, above, defines a untyped variable **`val`**, which is a decimal number. The second line calls an extended method, **`times()`**, on the integer object **`10`**. The **`times method()`** accepts a lexically scoped anonymous function, our closure. Do not worry too much about the syntax here of the closure. This is full version of closure syntax. The left hand side of the arrow (**`->`**) denotes closure parameters. There is only one and in this case, it is the variable called **`it`**, which happens to be the default name of a single closure parameter. The right hand side of the arrow (**`->`**) denotes the closure's body, which is to add a literal number **`0.1`** to the decimal variable **`val`**. (Groovy designers chose BigDecimal by default to avoid floating point accumulator errors, obviously to be programmer friendly at the command line.) The third line outputs the result of the accumulation exercise. Groovy uses GStrings behind the scenes to enhance the basic Java String. If you have experienced Bash or Korn Shell programming then you will recognise the style of variable substitution immediately.

To send mail with JavaMail API we need to create a SMTP authenticator. This is a simple class that implements the Authenticator interface and allows the calling program to send back the login credentials to the mail account.

We also require a way of templating the letter. We substitute text such as the subject and sender address at placeholders, key positions in the letter. The concept is similar to the way office word processing applications

PETER PILGRIM

Peter is a Java EE software developer, architect, and Sun Java Champion from London. By days he works as an independent contractor in the investment banking sector. Peter can be contacted at peter.pilgrim@gmail.com

Listing 1

```
class SMTPAuthenticator extends
   javax.mail.Authenticator
{
  private String d_email;
  private String d_password;
  public SMTPAuthenticator( String d_email,
     String d_password )
  {
    this.d_email = d_email;
    this.d_password = d_password;
  }
  public PasswordAuthentication
     getPasswordAuthentication()
  {
    return new PasswordAuthentication(d_email,
       d_password);
  }
}
```

substitute text at placeholder positions in a document in order to perform a mail merge.

Groovy SMTP authenticator

You can define classes in Groovy and extend interfaces and other (abstract) classes coming from the world of Java. Listing 1 is the code for SMTP authenticator class for the JavaMail API.

In Listing 1, we have defined a Groovy class implementation of the JavaMail authenticator. It is simply an object that accepts your Gmail email address and password and stores them to private data members. The JavaMail library framework will call back on the authenticator object in order to get credentials for the Google Mail SMTP service, at the appropriate time. Like all good frameworks, this design concept follows the *Hollywood Principle*, 'Don't call us, we'll call you!' By the way, you will also need to have a couple of Java imports in order to make all of this work successfully.

```
  import javax.mail.*;
  import javax.mail.internet.*;
  import com.sun.mail.smtp.*;
  import java.util.Properties
```

Listing 2

```
def mailout = '''
$firstName $lastName
$street
$address
$company
Dear $firstName
I am writing to inform you that I am no longer
actively looking for employment.
I have recently accepted a very generous offer.
Thank you for your past support.
Regards
--
A.N. Other
a.n.other at gmail dot com'''
```

Listing 3

```
def engine = new
groovy.text.SimpleTemplateEngine()
def template = engine.createTemplate(mailout)
def binding = [
  firstName: "Scott",
  lastName: "Liddle",
  company: "Agency One",
  street: "555 Liverpool Street",
  address: "Big ACME Building, London, EC1X 7EE"
  email: "scott.liddle@agencyone.acme.com" ]
def text = template.make(binding).toString()
println text;
```

The letter template

In scripting languages like Bash, there is a concept of a HEREDOC (or rather a *here-document*). A heredoc allows strings to contain arbitrary characters e.g. XML syntax and/or embedded new lines and tabs. Groovy has its own take on these heredocs with triple apostrophe characters. Here is the text of our resignation email, which we will eventually send to all recruitment agents.

In Listing 2, we have placeholders in the `GString` definition. The place holders will be substituted with values from the said named variables, if they exist – otherwise they will be empty. Listing 3 is a sample Groovy script to make the template engine actually work and produce output. It will work in the Groovy console.

Notice how, unlike Java, Groovy allows map collections to be literally defined in place. The simple template engine is also a default part of the GDK and there are a couple of other useful engines. You can find more information on the groovy.text.SimpleTemplateEngine inside the Groovy Development Kit manual, which is available online.

The MailMerge class

Being decent object oriented programmers, we define an object class for a mail merge (Listing 4). Sorry about this, but it is a bigger class. Groovy

Listing 4

```
class MailMerge {
  private String host;
  private int port
  private String senderAddress
  private String org
  private SMTPClient smtpClient;
  private String mailText;
  def exclusion = [];
  def password = 'PASSWORD'
  private Properties props;
  MailMerge( String host, int port,
     String password, String org,
     String senderAddress, String mailText )
  {
    this.host = host
    this.port = port
    this.org = org
    this.password = password
    this.senderAddress = senderAddress
    this.mailText = mailText
    props = System.getProperties();
    props.put("mail.smtp.user", senderAddress );
    props.put("mail.smtp.host", host);
    props.put("mail.smtp.port",
       Integer.toString(port));
    props.put("mail.smtp.starttls.enable",
       "true");
    props.put("mail.smtp.auth", "true");
    props.put("mail.smtp.socketFactory.port",
       Integer.toString(port));
    props.put("mail.smtp.socketFactory.class",
       "javax.net.ssl.SSLSocketFactory");
    props.put("mail.smtp.socketFactory.fallback",
      "false");
    props.each{ entry ->
      if ( entry.key =~ /^mail\./ ) {
       println entry.key + " = > " + entry.value
      }
    }
  }
  //...
}
```

Listing 5

```
class MailMerge {
  // ...
  public String trimQuotes( String s ) {
    if ( s.length() > 0 && s.getAt(0) == '\"' ) {
      s = s.substring(1)
    }
    if ( s.length() > 1 &&
       s.getAt(s.length()-1) == '\"' ) {
      s = s.substring( 0, s.length() - 1)
    }
    return s;
  }
  // ...
}
```

supports a great deal of the Java syntax without change, so I will gloss over the easy details. One can have private, protected and public data members. The mail merge class defines a list (untyped) collection variable called **`exclusion`**. It also stores a copy of the system properties for further use. Properties are mapped collections in Groovy too.

The constructor for **`MailMerge`** class follows the same style as Java. We set the data members of the object from the constructor parameters. Next, we get the system properties and append additional properties. We need to do this in order to initialise the JavaMail framework correctly for the GoogleMail SMTP service. Examine the last statement of the constructor, because it contains a closure that iterates through the property collection and dumps the key and value to the console for debugging. It also filters on the property name with a literal regular expression!

Listing 6

```
class MailMerge {
  // ...
  public void mailMerge( String inputCsvFile )
  {
    def handle = new File(inputCsvFile);
    def counter = 0;
    handle.eachLine {
      ++counter;
      def rawCellLine = it.split(",");
      def cellLine = [];
      rawCellLine.each{ cellLine.add( trimQuotes( it )) }
      def firstName = ( cellLine.size() > 1 ? cellLine[1] : '' );
      def lastName = ( cellLine.size() > 3 ? cellLine[3] : '' );
      def company = ( cellLine.size() > 4 ? cellLine[4] : '' );
      def street = ( cellLine.size() > 6 ? cellLine[6] : '' );
      def address1 = ( cellLine.size() > 7 ? cellLine[7] : '' );
      def address2 = ( cellLine.size() > 8 ? cellLine[8] : '' );
      def city = ( cellLine.size() > 9 ? cellLine[9] : '' );
      def postcode = ( cellLine.size() > 11 ? cellLine[11] : '' );
      def country = ( cellLine.size() > 12 ? cellLine[12] : '' );
      def email = ( cellLine.size() > 14 ? cellLine[14] : '' );
      if ( !exclusion.contains( firstName+" "+lastName ) &&
         email.length() > 0 &&
         email =~ /\b[A-Za-z0-9._%+-]+\@[A-Za-z0-9._%+-]+\b/  ) {
        def Authenticator auth = new SMTPAuthenticator( senderAddress, password );
        def Session session = Session.getInstance( props, auth);
        // set sender and recipient
        println "["+counter+"] Sending mail to "+ firstName+" "+lastName+" ( "+email+" )"
        def engine = new groovy.text.SimpleTemplateEngine()
        def template = engine.createTemplate(mailText)
        def binding = [
          firstName: firstName,
          lastName: lastName,
          company: company,
          street: street,
          address: (address1+" "+address2+" "+city+" "+postcode+" "+country),
          email: email
        ]
        def text = template.make(binding).toString()
        // send body of message
        def MimeMessage msg = new MimeMessage(session);
        msg.setText( text );
        msg.setSubject('Accepted Job Offer - Unavailable For Contracts');
        msg.setFrom( new InternetAddress(senderAddress) );
        // Test line to send to your own ISP account
        msg.addRecipient( Message.RecipientType.TO, new InternetAddress('A.N.Other@YOURISP.co.uk') );
        // Uncomment this line to actually send mail with GMAIL!
        // msg.addRecipient( Message.RecipientType.TO, new InternetAddress(email) );
        Transport.send(msg);
      }
    }
  }
  // ...
}
```

Trim the double quotes

With the constructor definition over, we need to have a handy function to tidy up those troublesome CSV files. I don't know about you, but the CSV file output by MS Excel that arrives from most of the business folks I know tends to have embedded quotes around the values, which we do not need in the mail merge program. Listing 5 is a useful Groovy function to remove leading and trailing double quotes:

The big operation

The `mailMerge()` is a large operational method. Essentially, it reads the CSV file one line at a time. (See Listing 6.) The bulk of the work happens in long closure statement. For each text line from the file, we break it into an equivalent list collection of Strings, which represent the row cells from the spreadsheet. We sanity check for an email cell, in this case the column number of electronic mail address in my CSV data file is 14. If this email cell exists and is valid, then we can move to the text templating and the sending of email, otherwise we skip to the next line.

So this code is not rocket science by any stretch of the imagination. Groovy easily allows a file to read line by line using a closure. Here is a simple Groovy script to read a source code line by file and prefix with the line number. Try it in the GroovyConsole.

```
def file = new File("Foo.groovy);
def count = 1;
file.eachLine {
   line -> println "${count} ${line}"; ++count }
```

Let us get back to the `MailMerge` program and the `mailMerge()` method. We make use of the `trimQuote()` function to tidy up the cell text. Essentially, Groovy Strings extends the `java.lang.String` with some useful functions of its own. The `split()` method breaks the supplied string into list collection of Strings according to the delimiter.

Remember our list of `exclusion` data; in the mail merge method we use it to avoid sending email to certain agents. This feature allows us the freedom to give those privileged agents our exclusive personal touch! Groovy supports regular expressions directly in the language. We use this feature to verify the contents of the `email` cell is actually a conforming electronic mail address.

The rest of code is split between the Groovy template engine, which I have already shown you above, and the code to send email using the JavaMail API. You can learn about the latter on-line, because there are tons of articles. The only part to explain is that an SMTP authenticator object is created each time. The authenticator instance and the stored system properties create the JavaMail session handler for sending email.

Listing 7

```
class MailMerge {
  // ...
}
def host = 'smtp.googlemail.com'
def port = 465
def org = 'YOURORG.com'
def emailAddress='YOURACCOUNT@gmail.com'
def secret='XXXXXXXX'
def mailout=''' as before ... '''
def x = new MailMerge(
 host, port, secret, org, emailAddress, mailout )
def inputfile = "C:\\Documents and Settings\\
   Peter\\My Documents\\
   RecruiterDatabase_2008.csv"
// Agents To Exclude
x.exclusion = [ 'Sally Martin',
   'Charles Underwood', 'Margaret Fitzgerald' ];
x.mailMerge( inputfile );
```

Make it work

We have a finished our mail merge Groovy class, which we can now instantiate and send news-update emails to those recipients!

The code in Listing 7 defines the SMTP server and port for Google Mail service. You will need to write your own Gmail address and secret password. Obviously, you need to edit the `mailMerge()` function in the `MailMerge` class to set up how to process your own CSV spreadsheet data file. For example to change the column cell numbers in each iteration. Once you have a working CSV file then you are ready to test the mail with a dummy target address. After testing then you can perform the mail merge for real.

Conclusion

In summary, you now have a program that demonstrates a mail merge with Groovy. I wrote this pretty quickly because Gmail certainly did not work correctly in Open Office 2.3! Writing the equivalent code in Java would be a lot more complex and in my opinion have taken longer to get right. The mentality of a Java programmer starts off with proper object oriented techniques. A dynamic scripting language allows one to be, how should I say, more lazy. There is a question of maintainability with dynamic languages over static typed ones. The benefits for this new Groovy programmer are clear however. I wrote a simple mail merge with a text templating, authenticator and SMTP authenticator in less than a day. I simply had no extra time to devote to writing the code in the Java traditional OOP way. So I apologise if this code is a little funky.

This code should be tidied up. For instance it would be better to make some of the configuration information such the SMTP hostname "smtp.googlemail.com" and port "456" as final static constants. Ideally they should be properties of the mail merge and be loadable from a properties file. Of course that is an exercise for the reader. ■

References

http://groovy.codehaus.org/ (Groovy Home Page, Groovy Development Kit)

http://www.onjava.com/pub/a/onjava/2007/03/23/using-groovy-to-send-emails.html (Using Groovy send Email)

http://forum.java.sun.com/thread.jspa?threadID=668779 (Using Java to sent mail using Gmail Provider)

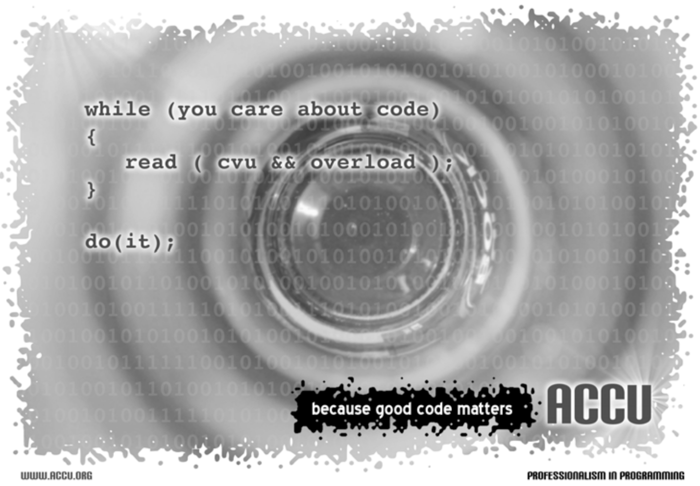
http://blogs.sun.com/apanicker/entry/java_code_for_smtp_server (Java Code for SMTP server)

http://java.sun.com/products/javamail/downloads/index.html (Javamail 1.4 Download)

http://java.sun.com/javase/technologies/desktop/javabeans/jaf/index.jsp (JavaBeans Activation Framework)

http://groovy.codehaus.org/groovy-jdk/ (Groovy Development Kit)

http://groovy.codehaus.org/User+Guide (The very well documented, Groovy User Guide)


while (you care about code)
{
read (cvu && overload);
}
do(it);
because good code matters
ACCU
WWW.ACCU.ORG
PROFESSIONALISM IN PROGRAMMING

Code Critique Competition 54

Set and collated by Roger Orr.

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

Last issue's code

I'm trying to write a program to see if a triangle is right angled. I've got part way there, but the program seems a bit unreliable – sometimes it crashes and sometimes it says triangles are right angled that aren't. Can you help?

Can you help answer the question? The code is shown in Listing 1.

Listing 1

```
#include <iostream>
#include <map>
#include <string>
#include <sstream>
using std::cin;
using std::cout;
using std::endl;
typedef std::map<int,int> triangle;
// will have 3 points!
// Read integer: DD (base10) or 0xDD (base16)
int readInt( std::istream & is )
{
  std::string s;
  is>>s;
  int result(0);
  char x;
  std::istringstream iss(s);
  if ( ! iss>>result )
    cout << "Not a number!" << endl;
  else if ( result==0 && iss>>x && x=='x' )
    iss>>std::hex>>result;
  if ( ! iss.eof() )
    result = 0;
  return result;
}
triangle readTriangle( std::istream &is )
{
  triangle vals;
  cout << "Enter triangle coordinates: ";
  for ( int idx = 0; idx != 3; ++idx )
  {
    int key = readInt( cin );
    vals[key] = readInt( cin );
  }
  return vals;
}
int main()
{
  triangle vals = readTriangle( cin );
  triangle::iterator it = vals.begin();
  int ax = it++->first;
  int bx = it++->first;
  int cx = it++->first;
  double slope1 = ( vals[bx] - vals[ax] ) /
    double( bx - ax );
  double slope2 = ( vals[cx] - vals[bx] ) /
    double( cx - bx );
  if ( slope1 * slope2 == -1 )
    cout << "Right angled" << endl;
}
```

Critique

Robert Jones <robertgbjones@gmail.com>

I have identified two specific errors in this code, but they may be masking other issues as well. The whole approach of the code is bizarre, but that is presumably because it is a contrived example.

Firstly, the `readInt()` function contains the line,

```
if ( ! iss>>result )
```

which actually binds as

```
if ( ( ! iss ) >> result )
```

when what was required was

```
if ( ! ( iss >> result ) )
```

Secondly, the use of a map structure means that if any of the points of the triangle have the same x-coordinate only one of the points will be recorded. So when presented with

```
00 04 04 00 00 00
```

which represents {0,4}, {4,0}, {0,0}, what is seen is {0,0}, {4,0}. The use of a fixed array of points would be more appropriate.

Frances Buontempo < frances.buontempo@gmail.com>

This is an interesting problem. It got my attention because I had to think for a while about the test for a right angle.

```
double slope1 = ( vals[bx] - vals[ax] ) /
  double( bx - ax );
double slope2 = ( vals[cx] - vals[bx] ) /
  double( cx - bx );
if ( slope1 * slope2 == -1 )
  cout << "Right angled" << endl;
```

A line from (0,0) to (x,y) is rotated 90^o anti-clockwise if and only if it ends up at (-y,x). See figure 1.

The gradient or slope of the first line is y/x. The slope of the second line is y/-x. This gives us:

```
slope1 * slope2 = (y/x)*(x/-y)
                = (y*x)/(x*-y)
                = (y*x)/(y*x*-1) = -1
```

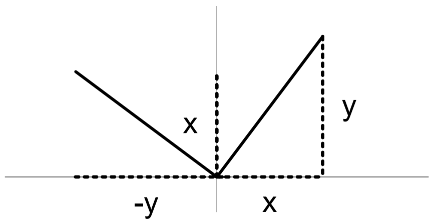


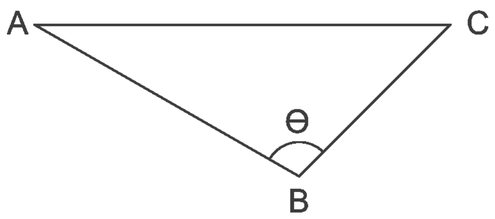
Figure 1

Similarly, for a line translated 90^o clockwise (x,y) will end up at (y,-x), again giving us `slope1 * slope2 = -1`.

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

Figure 2



Now consider a triangle, shown in Figure 2.

The angle between the line AB and BC is the angle we are testing. The code is writing **(ax, vals[ax])** for the coordinates of A, **(bx, vals[bx])** for the coordinates of B, and **(cx, vals[cx])** for the coordinates of C. This took a moment to spot, and is unintuitive. We call **vals[ax]** because the data is stored in a map. How very odd. All will be revealed. We'll get back to the code in a moment.

Imagine rotating line BC round to BA through θ degrees. We can treat point B as the origin (0,0). Point C is then at **(cx - bx, vals[cx] - vals[bx])**, and point A is at **(bx - ax , vals[bx] - vals[cx])**. Now, provided **bx-ax** isn't zero and **cx-bx** isn't zero, we can use the test of the product of the slopes above. If the product is -1, we have a right angle.

Of course, if **bx-ax** is zero, meaning the x-coordinates are identical, or **cx-bx** is zero, again meaning the x-coordinates are identical, the test won't work since we will be trying to divide by zero. At this point we have a choice: we can either decide the test does not work for a whole load of triangles, tough, or find another test. In fact, the test in the code won't work for any triangles with a vertical side. The first triangle I wanted to test was (1,0), (0,0) and (0,1) but this is not possible.

We could use a different test. Now, the inner product of two (mathematical) vectors (a1,a2) and (b1, b2) is **a1 . b1 + a2 . b2 = lengthAB . cos** θ. This is also known as the dot product. See http://en.wikipedia.org/wiki/Dot_product.

Since cosine 90^o is 0, we could change the test at the end of the code to:

```
int inner_product = ( bx - ax )*( cx - bx )
                  +( by - ay )*( cy - by );
if ( inner_product == 0 )
  cout << "Right angled" << endl;
```

This provides one way to avoid the potential divide by zero. Adopting this approach allows any triangle to be tested.

If we stick with the test adopted in the code, we must avoid the potential divide by zero. This can be achieved by forcing users to only enter triangles that are free from vertical lines. How do you do that? Well, avoid identical x co-ordinates. And how do we do that? Obvious, isn't it? Store the co-ordinates in a **std::map**. Ah! I see now. When I first looked at the code, my immediate question was why are we using a map?

Of course, with the code as it is, the poor user can still try to enter my favourite right angled triangle (1,0), (0,0), (0,1). Let's explore what happens when we do this. Starting at the top of **main**, we call **readTriangle**. Actually, before I get as far as my favourite right angled triangle, I can just type "RUBBISH" in can't I? OK, I will get the helpful message,**"Not a number!"** but the **readInt** function will return 0. Then **readTriangle** will try to add this to the map, but the code doesn't check the point has been added. If I keep typing "rubbish" it tries to add (0,0) to the triangle co-ordinates, but fails, since we have chosen to use a map, to avoid identical x co-ordinates. But no clue is given to the user.

When we get back to **main**, we increment the iterator into the map under the assumption, stated in the comment:

```
// will have 3 points!
```

(By the way, why is this an **iterator** rather than a **const_iterator**?). Unsurprisingly, regardless of the imperative expressed in this comment, at runtime we are in trouble if there aren't enough points in the triangle. The code could test there are three points in the 'triangle':

```
if( vals.size() == 3 )
{
  //do incrementing and test here
}
```

This may avoid some of the crashes. Alternatively, in **readTriangle**, instead of just calling

```
int key = readInt( cin );
vals[key] = readInt( cin );
```

it may be more informative to provide some feedback, and only increment the index when a point has been added:

```
std::pair<triangle::const_iterator,bool> ret =
  vals.insert(std::make_pair(key,val));
if( ret.second )
{
  cout << "inserted [" <<
    key << "," << val << "]" << endl;
  ++idx;
}
else
{
  cout << "Failed to insert ["
    << key << "," << val << "]" << endl;
}
```

Now, at least when I try to test my favourite triangle for right angledness, I am told that I can't add (0,1). Which is very disappointing, but at least it won't crash when it tries to iterate through non-existent points.

Is everything OK now? No. Let's try the program again with the modification to **readTriangle**. I will forget my favourite triangle and try (1,1), (0,0) (-1,1). This is a right angled triangle, and should get inserted into the map because the x co-ordinates all differ. Oh dear! This happens.

```
Enter triangle coordinates: 1 1
inserted [0,0]
0 0
Failed to insert [0,0]
-1 1
Failed to insert [0,0]
```

Before, the code would carry on regardless, and we would only have (0,0) in the map despite the comment that it will have three points. What on earth is going on? Ah. Found it.

```
if ( ! iss>>result )
  cout << "Not a number!" << endl;
```

! has higher precedence than **>>**, so this is calling

```
if ( (!iss) >> result )
  cout << "Not a number!" << endl;
```

In other words, bit shifting (**!iss**), which is false, by result, which is 0, always giving us **0 >> 0**, zero. What we actually need is

```
if ( ! (iss>>result) )
  cout << "Not a number!" << endl;
```

It would probably be better to throw an exception here, so we know we haven't read an int, rather than return 0 and carry on regardless.

With the brackets around **(iss>>result)** we now get

```
Enter triangle coordinates: 1 1
inserted [1,1]
0 0
inserted [0,0]
-1 1
inserted [-1,1]
Right angled
```

Much better! I still can't test my favourite right angled triangle, but I'll get over it. I know that's right angled. How about my second favourite triangle? (3.3, 3.3), (0.0, 0.0), (-3.3, 3.3)?

```
Enter triangle coordinates: 3.3 3.3
inserted [0,0]
0 0
Failed to insert [0,0]
-3.3 3.3
Failed to insert [0,0]
```

So, I can't have a triangle with non-integer co-ordinates, in addition to the ban on triangles with vertical lines? No fair! This disallows an infinite number of triangles. Oh well. There is code there to accept integers represented in hex, so I can input the same triangle in a couple of ways to make up for things. Let's try it.

```
Enter triangle coordinates: 0x01 0x01
inserted [1,1]
0x00 0x00
inserted [0,0]
-0x01 0x01
Failed to insert [1,1]
```

Oh, really. First, I have typed the numbers in as hex, but the code I added to figure out what was happening in here doesn't send them back in the same format. That's my fault though. Now, what about these `-0x01` values becoming 1? The `readInt` function needs some work.

```
int readInt( std::istream & is )
{
  std::string s;
  is >> s;
  int result(0);
  char x;
  std::istringstream iss(s);
  if ( ! (iss>>result) )
    cout << "Not a number!" << endl;
  else if ( result==0 && iss>>x && x=='x' )
  {
    iss>>std::hex>>result;
  }
  if ( ! iss.eof() )
    result = 0;
  return result;
}
```

OK. In order, from the top. We put the contents of the stream into a string. We put the contents of the stream into a stringstream. I'm getting dizzy already. We put the contents of the stringstream into an int. For "0x01" we will be OK. Result will be 0, then we read the next character into a char, and if it's 'x', we set the stream to be hex and read the remains into result. Result! What happens with "-0x01"? We read -0 into result, which means result is zero and we forget about the minus sign. And then continue as before, read the 'x' and treat the remaining "01" as hex, incorrectly giving 1 rather than -1.

If we remember the minus sign at the start we can make this work:

```
int readInt( std::istream & is )
{
  std::string s;
  is >> s;
  int sign = 1;

  int result(0);
  char x;
  std::istringstream iss(s);
  if ( ! (iss>>result) )
    cout << "Not a number!" << endl;
  else if ( result==0 && iss>>x && x=='x' )
  {
    iss>>std::hex>>result;
    if (!s.empty() && s[0]=='-')
      sign = -1;
  }
  if ( ! iss.eof() )
    result = 0;
  return sign*result;
}
```

Why the user would prefer to be allowed to type in the values in hex, but not allowed to use doubles or triangles oriented horizontally is curious.

These suggestions will stop the code crashing and make it correctly identify if a triangle, with non-vertical sides, is right angled. It is far from perfect. Of course, if I type "Rubbish" as an input it will assume a value of zero and carry on regardless. I am tempted to re-write the `readInt` function, but will leave that as an exercise for the reader.

Joe Wood <joew@aleph.org.uk>

The following conversation takes place between two post-graduate students one evening.

Derek sat down, a rueful smile on his face "Joe's been to see me again today."

Brian "Oh, he tried to nab me in the lab, but I made my excuses. Is it a mess?"

Derek "Some ideas are interesting. But I think his solution is ideally fitted for the yokel joke about 'If I was going there I wouldn't start from here.'"

Brian "So what's wrong?"

Derek "Very briefly. Joe wants to determine if a set of points represents a right angled triangle. He starts by representing a triangle as a map between `int`s and `int`s."

Brian "Why use a map of `int`s? Triangles always have three points, even if they degenerate to a line or even a point. And why use integers at all?"

"Good questions. A map is particularly unfortunate because two points like (1,2) and (1,3) will degenerate to a single point, probably (1,3)."

"As regards to the use of `int`s, I was puzzled. From the one line comment above a function called `readInt`, I think part of the exercise is to handle both decimal and hexadecimal integer numbers. Frankly, however, I don't get Joe's `readInt` at all."

"Hold it, I want my laptop to set up some test cases."

Derek "Right, let's see (3,4,5) is a valid right angled triangle, as is (5,12,13). So that's [(0, 0) (0, 3) (4, 0)] and [(0, 0) (0, 5) (12, 0)]."

"However, the triangles [(0, 0) (0, 3) (3, 1)], [(0, 3) (0, 3) (4, 0)] and [(0, 0) (0, 0) (0, 0)] definitely are not. The first is not right angled, the second is really a line and the third is a single point."

"You knock out an initial test file. Later, we should test for horizontal and vertical shifts, say (+6, 13), including negative cases, say (+6, -13)."

Brian "OK, let's suppose that we can fix `readInt` later, for now we just use."

```
int readInt ( std::istream & is ) {
  int result = 0;
  is >> result;
  return result;
}
```

Brian "We need a hexadecimal test case as well, hmm, the positive shift case converted to hex."

"Let's use a pair of `int`s to represent a point, with three points in a triangle."

```
// need <utility>
typedef std::pair<int,int> point;

struct triangle {
  point a;
  point b;
  point c;
};
```

Derek "Why use a `struct` for triangle? We could use a plain array, or a vector or even a class?"

Brian "Well, arrays quickly degenerate into plain pointers, with absolutely nothing to check their size. We could use a vector, but their size can change, and there is nothing to indicate how many points it should contain. A class seems a little overkill, it's a small program and the default constructors and destructor work fine in this case."

Derek "OK, anyway we can always revise this decision later. Given that we have introduced a point abstraction, I think we should add a new function `readPoint`."

```
point readPoint ( std::istream & is ) {
  const point p ( readInt(is), readInt(is) );
  return p;
}
```

Brian "Hmm, but you can't use **`point(readInt,readInt)`** because C++ makes no statement about the order of parameter evaluation, we must use."

```
point readPoint ( std::istream & is ) {
  point p;
  p.first  = readInt(is);
  p.second = readInt(is);
  return p;
}
```

Derek "Good point. Now that makes **`readTriangle`** clearer."

```
struct triangle readTriangle
    ( std::istream & is ) {
  struct triangle vals;
  cout << "Enter triangle co-ordinates: ";
  vals.a = readPoint(is);
  vals.b = readPoint(is);
  vals.c = readPoint(is);
  return vals;
}
```

Brian "By the way, I see Joe didn't use the input parameter. What does he think compiler warnings are for?"

"What about determining if it's a right angled triangle?"

Derek "Well, Joe had an interesting idea. As you may recall from calculus, two perpendicular lines have tangents that multiply together and make -1."

Brian "You know maths is not my strong point, yeah, yeah, we're still looking! But I see two problems. Firstly, which two of the three tangents are you going to use? And secondly, if one of the sides is parallel to the vertical axis, the gradient will overflow, another undefined C++ area."

Derek "Both true. So we will ditch the tangents and use Pythagoras' theorem, which you will remember said 'The sum of the square of the hypotenuse is equal to the sum of the squares of the other two sides'."

Brian "Yes, I remember Pythagoras, but what if it's not a right angled triangle?"

Derek "Well, the cosine rule, generalises Pythagoras' theorem, and it means that if Pythagoras' theorem holds it must be a right angled triangle."

Brian "Good job we using integers, or we'd have to worry about equality."

Derek "True. Let's add a function to find the length of a line."

Brian "We can be smarter. Your function would need to take the square root, and later square it to apply Pythagoras' theorem. If instead, we just calculate a line metric using"

```
typedef long metric;
metric lineMetric ( const point & p1,
                    const point & p2 ) {
  const metric dx = ( p1.first  - p2.first );
  const metric dy = ( p1.second - p2.second );

  return dx * dx + dy * dy ;
}
```

Brian "We needn't take the square root and lose the precision."

Derek "Perhaps the maths is beginning to rub off. We still need to calculate the three line metrics and apply Pythagoras."

Brian "Well if we put the line metrics into a local vector, and sort it, if Pythagoras holds we get."

```
v[0] = v[1] + v[2]
```

Derek "Have to sort the vector using greater than, but that's easy. Not too happy about adding **`v[1]`** and **`v[2]`**, but is a local action, so it's safe enough."

Brian "What if somebody enters a line, not a triangle, then we always get, **`v[0]=v[1]`** as **`v[2]`** is zero."

Derek "We must test for that. So our main function becomes"

```
int main()
{
  struct triangle vals = readTriangle ( cin );
  // need <vector>
  std::vector<metric> sides;
  sides.push_back(lineMetric(vals.a, vals.b));
  sides.push_back(lineMetric(vals.b, vals.c));
  sides.push_back(lineMetric(vals.a, vals.c));
  // need <algorithm>
  std::sort(sides.begin(), sides.end(),
    cmpMetric );
  if ( ( sides[0] != sides[1] ) &&
       ( sides[0] == sides[1] + sides[2] ) ) {
    cout << "Right angled" << endl
  } else {
    cout << "NOT right angled" << endl;
  }

  return 0;
}
with
int cmpMetric ( metric a, metric b )
{
  return a > b ;
}
```

Derek "I need another pint."

Brian "Hold on. What about **`readInt`**?"

Derek "Get the drinks and I'll have another think."

A little later.

Derek "Joe reads the number into a string, which seems a good idea. How come you have to tell C++ to handle a hexadecimal number, some languages know that '0x' starts a hex string and behave correctly?"

Brian "No idea, just one of those things. Suppose we split the problem, is it a hex string and correctly process the remaining input string."

Derek "OK, but your have to modify the input string to take off the initial 0x if required. So **`readInt`** becomes something like"

```
int readInt ( std::istream & is )
{
  int result(0);
  // Test for good input stream
  if ( is ) {
    std::string s;
    is >> s;
    const bool hexString = isHexString ( s ) ;
    std::istringstream iss(s);
    if ( ! hexString ) {
      iss >> result ;
    } else {
      iss >> std::hex >> result ;
    }
    // reset the input stream
    is.clear();
  }
  return result;
}
```

Brian "And then we have."

```
bool isHexString ( std::string s )
{
  const std::string hexStr("0x");
  bool isHex = false ;
  if ( s.size() >= hexStr.size() ) {
    std::string format =
      s.substr(0, hexStr.size());
    for (size_t i=0; i != format.size(); ++i){
      // need <cctype>
      format[i] = tolower(format[i]);
    }
    if ( format == hexStr ) {
      isHex = true;
      s = s.substr( hexStr.size(), s.size() );
    }
  }
  return isHex;
}
```

Derek "Oh, I see, test for the format identifier, and if present strip it off the input string and set the return flag."

Brian "OK, that works, I hope Joe appreciates all our hard work."

Derek "I'm sure he will until his next assignment is due."

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

Looking at the code, I see three major issues:

1. Because triangle is just a **std::map**, it only contains points which have unique x-coordinates (since x-coordinates are the key, and keys of maps are unique). For example, the triangle described by (0,0),(0,3),(4,0) will only have two of its vertices stored, and the third dereference of the iterator has undefined results.
2. Slopes can have values of negative-infinity and positive-infinity. Again, the example of (0,0),(0,3),(4,0) shows this.
3. The inexactness of floating point mathematics. For some calculations, the multiplication of the calculated slopes may not result in exactly -1. Read the paper 'What Every Computer Scientist Should Know About Floating-Point Arithmetic' at <http://dlc.sun.com/pdf/800-7895/800-7895.pdf>. The layman advice I usually give to people about floating point is that casual use is for display purposes only when exactness doesn't matter.

Now, we can mitigate (1) by using a **std::multimap** instead of a **std::map**. But more to the point, it just isn't the right data structure. It has far too complicated an interface to use for storing something as simple as a triangle. Let us go back to fundamentals.

What is a triangle? A collection of three points. What is a point? An x-coordinate and a y-coordinate. I will start there:

```
struct Point
{
  int x;
  int y;
  friend std::ostream& operator<<
    (std::ostream& os, Point const& p)
  { return os << '(' << p.x
              << ',' << p.y << ')'; }
  friend std::istream& operator>>
    (std::istream& is, Point& p)
  { is.unsetf(std::ios::dec);
    return is >> p.x >> p.y; }
};
```

Because I created a data type for **Point**, I can give that type a stream inserter (**operator<<**) and extractor (**operator>>**). Of note in this extractor is the **unsetf()** call, which sets the stream up so that the **int** extractors automatically handle both integer and hexadecimal numbers. This takes the place of the work done in the original **readInt()** function.

Now that I have **Point**, let me define **Triangle** in terms of it:

```
struct Triangle
{
  Point   vertices[3];
  friend std::ostream& operator<<
    (std::ostream& os, Triangle const& t)
  { return os << t.vertices[0]
              << ',' << t.vertices[1]
              << ',' << t.vertices[2]; }
  friend std::istream& operator>>
    (std::istream& is, Triangle& t)
  { return is >> t.vertices[0]
      >> t.vertices[1] >> t.vertices[2]; }
};
```

Note: while this program as is doesn't use the inserters, I left them in anyway, as they are very useful during debugging.

To mitigate issues (2) and (3), I'll use other mathematics to both avoid division and to keep things in the integer realm. Namely, I'll use the Pythagorean Theorem: the square of the hypotenuse of a right triangle is equal to the sum of the squares on the other two sides.

How does one calculate the square of the side of a triangle where that side is specified by two points? That's easy:

```
int Distance(Point const& a, Point const& b)
{ return (a.x - b.x) * (a.x - b.x)
       + (a.y - b.y) * (a.y - b.y); }
```

Note: If the x-coordinates or y-coordinates are sufficiently large, this and other calculations can cause overflow, which real-world code should consider. For purposes of this critique, I am assuming that the values are not that large.

Applying the Pythagorean Theorem:

```
bool IsRightTriangle(Triangle const& t)
{
  int sides[] =
  {
    Distance(t.vertices[0], t.vertices[1]),
    Distance(t.vertices[0], t.vertices[2]),
    Distance(t.vertices[1], t.vertices[2]),
  };
  std::sort(sides, sides + 3);
  return sides[2] == sides[1] + sides[0]
      && sides[0];
}
```

The way this algorithm works is as follows:

A. Calculate the square of the length of each side.

B. Sort them by the square of the length, so that the square of the hypotenuse is in **sides[2]** and (as I'll need in (ii) below) the the square of the shortest side is in **sides[0]**. One might ask: is **std::sort** overkill for sorting three values? No, as it already works and therefore is code that I don't have to write, debug or maintain. Plus, most implementations optimize it for small data sets anyway.

   Consider degenerate cases, where the three vertices do not describe a triangle:

   (i) All three points are distinct but on the same line. It turns out that this implementation will correctly return **false** in this case.

   (ii) Two or more points are the same (for example, (0,0),(0,0),(0,0)). This will result in the smallest side being 0, and needs to be tested for explicitly.

C. Return whether or not we are a right triangle by applying the Pythagorean Theorem.

   Putting it all together, using as close as possible the same formatting, input and output as the original program:

```
int main()
{
  std::cout << "Enter triangle coordinates: ";

  Triangle vals;
  if (!(std::cin >> vals))
    std::cout << "Not a number!" << std::endl;
  else if (IsRightTriangle(vals))
    std::cout << "Right angled" << std::endl;
}
```

Note: Instead of the original **readTriangle()** function, I just use the extractor. However, the output may not be exactly the same in the case where the user provides data that isn't an integral or hexadecimal value.

Michal Rotkiewicz <michal@michalhr.ehost.pl>

In this program there are three problems:

1. Input reading
2. Data storing
3. Algorithm

Let's see what is going on at each point.

1. Input readings

There is an issue with **readInt** function. At first sight the instruction **if (! iss>>result)** looks innocent, but there is a problem in fact. **std::istringststream** class has **operator!** function that returns **true** if either one of the error flags (**failbit** or **badbit**) is set on the stream. Otherwise it returns **false**. So in fact our **if** behaves like this:

```
if (iss.operator!() >> result)
```

As on the left side of **operator >>** there is no **istringstream** object any more **>>** is interpreted as a right bit shift.

A simple example will prove that:

```
int main() {
  std::string s;
  cin>>s;
  int result(0);
  char x;
  std::istringstream iss(s);
  iss>>result;
  iss>>result;
  cout<<(!iss>>result)<<endl;
  return 0;
}
```

If I run this program and enter a value greater than or equal to 1 the output is 0, but if we enter 0 the output is 1. How does it work? My intention was to setup error flag on the stream. To achieve that I used:

```
iss>>result;
iss>>result;
```

While the first one reads value from the stream the second one sets error flag on the stream as the stream is empty at this point. Thanks to it **!iss** returns **true**. When I run the program and enter value 1 the **!iss>>result** becomes **true>>1**. **True** is converted to **int** and I have **1>>1** which is 0. But if I enter value 0 the expression **1>>0** is evaluated to 1 as 1 shifted right by 0 is still 1.

To solve the problem we have to enforce that **iss>>result** is done first. Instead of **if (! iss>>result)** we write **if (!(iss>>result))**.

Now the **readInt** function works correctly. To make this function shorter we may get rid of

```
if (!iss.eof() )
    result=0;
```

as we do only one **iss>>result** operation so the **eof** bit won't be set.

2. Data storing

The next bug is in **readTriangle** function. This function works only if the first coordinates are different. Map class allows to keep only one element with given key. If we provide the second element with the same key the first element's value gets overwritten with the second element's value. To keep such data we need a multimap container.

It comes from "map" header as well. Comparing to map, multimap doesn't have **operator[]** so we can't use the instruction:

```
int key = readInt(cin);
vals[key] = readInt(cin);
```

Instead we may insert elements like this:

```
int key = readInt(cin);
int val = readInt(cin);
vals.insert(std::make_pair(key, val));
```

3. Algorithm

Ok – at this point we have all coordinates collected. Finally we have to determine whether given triangle is right-angled. Unfortunately existing algorithm doesn't answer to this question correctly for two reasons:

1. It checks if angle only at vertex B is right,
2. It leads to 'division by zero' error if **bx-ax** equals 0 or **cx-bx** equals 0. What I suggest to do is to calculate the square of each side length. If triangle is right-angled we are able to find side **a** so that **a^2 = b^2 + c^2**.

Last but not least: so far we have had silent assumption that we have a triangle. But it's possible to enter coordinates that don't form a triangle (eg. they are colinear). Therefore I suggest to check it. In many cases we calculate sides' length and check if sum of any two is bigger that the length of the third.

Instead of calculating lengths and introduce floating point values I decided to use another approach: if we have a triangle its area must be greater than 0. Area may be easily calculated using determinant:

```
                   | Ax Bx Cx |
Area = 0.5 * abs   | Ay By Cy |
                   | 1  1  1  |
```

that may be presented as:

```
Area = 0.5 * abs(
  (Cx-Ax)*(By-Ay) - (Bx-Ax)*(Cy-Ay) );
```

To check if area is not equal to zero it's sufficient to check that **(Cx-Ax)*(By-Ay) - (Bx-Ax)*(Cy-Ay)** is not equal to zero.

Finally the program looks like:

```
#include <iostream>
#include <map>
#include <string>
#include <sstream>
using std::cin;
using std::cout;
using std::cerr;
using std::endl;

typedef std::multimap<int,int> triangle;
int readInt( std::istream &is) {
  std::string s;
  is>>s;
  int result(0);
  char x;
  std::istringstream iss(s);
  if (! (iss>>result))
    cout<<"Not a number !"<<endl;
  else if (result==0 && iss>>x && x=='x') {
    iss>>std::hex>>result;
  }
  return result;
}
triangle readTriangle(std::istream &is) {
  triangle vals;
  cout<<"Enter triangle coordinates: ";
  for (int idx=0; idx !=3;++idx)
  {
    int key = readInt(cin);
    int value = readInt(cin);
    vals.insert(std::make_pair(key,value));
  }
  return vals;
}
```

```
int main() {
  triangle vals= readTriangle(cin);
  triangle::iterator it = vals.begin();
  int Ax = it->first;
  int Ay = it++->second;
  int Bx = it->first;
  int By = it++->second;
  int Cx = it->first;
  int Cy = it++->second;
  int double_area = (Cx-Ax)*(By-Ay)
                  - (Bx-Ax)*(Cy-Ay);
  if (double_area == 0) {
     cout<<"Not a triangle!"<<endl;
     return 0;
  }
  int a_sqr = (Bx-Ax)*(Bx-Ax)
            + (By-Ay)*(By-Ay);
  int b_sqr = (Bx-Cx)*(Bx-Cx)
            + (By-Cy)*(By-Cy);
  int c_sqr = (Cx-Ax)*(Cx-Ax)
            + (Cy-Ay)*(Cy-Ay);
  if (a_sqr == b_sqr + c_sqr or
      b_sqr == a_sqr + c_sqr or
      c_sqr == a_sqr + b_sqr)
    cout <<"Right angled"<< endl;
  return 0;
}
```

Colin Grant <cgrant@gtsoftware.co.uk>

So, I'll start by skipping over the lack of definition for how it should be used, e.g. test cases, and get on to the starting with the first problem – the function to read in an integer is broken. Firstly the unary operator `!` grabs the string stream so round brackets can 'mend' that

```
if ( ! (iss>>result))
```

Generally the `readInt` function smells – I don't see any advantage in passing in a stream when the guts use `cout` anyway (and the interim `readTriangle` ignores its stream parameter), and it just doesn't fill me with any confidence that it is efficient or sane. In a code review I would suggest the author considers if it is very clear and whether the error handling is sufficient because bad values will return a result – exceptions spring to mind for exactly this.

Moving on to the data representation of a triangle – using an associative container is never going to work since having the same first coordinate means there are fewer entries in the map. Extensibility is a good goal but since a triangle is never going to change from having exactly three points, I suggest another representation that captures the three pointedness of a triangle explicitly would be better.

Finally the algorithm to work out the right angledness of the triangle cries out to be separated from the control code, but disregarding that it's easy to see that it needs to be revisited to avoid division by zero errors.

As a user I would prefer positive feedback about the right angle nature of the triangle, but that's just me (I guess it depends on the unstated requirements).

Please tell me that the code is a made up example! Which begs the question, how did you manage to make up the horrid code?

Roger: yes, the code is made up; but the constituent pieces do all come from real code. I am still surprised by the strange uses to which people put the STL collection classes!

Commentary

I think the entrants between them have covered pretty well all the issues with the code. The main point about the code is that this is one of the cases where using the STL was a bad choice – or at least using `std::map` was. I was pleased to see that Nevin and Michal both mentioned the `std::multimap` as my impression is that the multixxx associations deserve to be better known; although in this particular case I think it was right to prefer a simpler data structure.

The Winner of CC 53

I had a couple of goes at deciding who should get the prize in this issue. Although I appreciated Frances' pictures and also enjoyed the dialog between Derek and Brian in Joe's entry, I eventually decided that Nevin's critique was the best one; one of its strengths seems to me the way that the data structures and the code in `main` so clearly express intent (and he also got a 'bonus' mark for using `unsetf` to allow reading of hex).

Code Critique 54

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

Listing 2

```
// ---- logging.h -----
enum { ERROR, WARN, DEBUG };
int level;
std::string now()
{
  time_t timeNow = time( 0 );
  char buffer[ 20 ];
  strftime( buffer, sizeof( buffer ),
    "%d %b %H:%M:%S", localtime( &timeNow ) );
  return buffer;
}
#define LOG(LEV, X) \
{ \
  if ( level & LEV ) { \
    std::ostringstream oss; \
    oss << now() << " [" << #LEV << "] " \
        << (X) << std::endl; \
    printf( oss.str().c_str() ); \
  } \
}
#define LOG_ERROR(x) \
LOG(ERROR, x)
#define LOG_WARN(x) \
LOG(WARN, x)
#define LOG_DEBUG(x) \
LOG(DEBUG, x)
// ---- example.cpp ----
#include <ctime>
#include <iostream>
#include <string>
#include <sstream>
#include "logging.h"

int main()
{
  level = DEBUG | WARN | ERROR;
  LOG_DEBUG( "This is a test" );
  LOG_WARN( "This is a warning" );
  std::ostringstream oss;
  oss << "An example msg";
  LOG_ERROR( "Problem: " + oss.str() );
}
```

I'm trying to write a simple logging header file, but it doesn't seem to be doing the right thing – my errors aren't being logged. Can someone help me sort this out?

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. ■

Regional Meetings

A round-up of the latest ACCU regional events.

ACCU Bristol

Report by Ewan Milne (ewan@calenture.org)

On a demi-semi-regular basis, a group of Bristol-based ACCU members meet up for an evening of curry and coding (Patia and patterns? Jalfrezi and Java? Oh, okay then). Sometimes we are joined by other members visiting the area: on one previous occasion, a group of bemused waiters was even treated to a full double-speed preview of a conference keynote talk (complete with slides!).

This time around (15th September) there were five of us present. I was joined by Tony Barrett-Powell, Tom Guest, Rob Jones and Pavol Rovensky, while apologies were received from Jon Jagger (family illness) and Kevlin Henney (too busy jet-setting). The conversation was relatively light on geek topics, covering parenthood (both recent and impending), dangerous locations for customer site visits (a tie between Tony's nuclear power station and my North Sea oil rig), and Pavol's hobby of competition-level model aircraft flying.

In finest Bristolian 'why rush?' tradition, the establishment of a more formal local group has been discussed on and off in recent times. But I'm now going to commit in print to getting the ball rolling early in 2009. We are looking for somewhere to meet, so if you can help find a location please contact me at ewan@calenture.org. And also get in touch if you are based in or near Bristol and would be interested in attending a local group. Hopefully there will an announcement of the first meeting on accu-general in the new year.

ACCU Oxford

Report by Jim Hague (jim.hague@acm.org)

The copy deadline for the last CVu meant that my first report on the nascent ACCU Oxford had to be filed shortly before our second meeting on July 30th.

As promised, Chris Jefferson, BSI C++ committee member, gave what turned into a full-length presentation on what's likely to be in C++0x (as of that week). It was an excellent session, enjoyed by both C++ gurus and those like myself with rusty C++.

After a break for the monsoon season (UK readers will understand), we returned with Pete Dillon presenting the IBM zSeries mainframe world. A foreign land to most, Pete's observations on where Unix and other systems go wrong gave plenty of food for thought. We'll return to Pete and his thoughts on Cobol at a later date.

Our next meeting will be on October 29th, when we're planning an open discussion of what qualities developers and their managers look for in each other. Keep an eye on ACCU-general for details, or if the volume there overwhelms you check http://www.lunch.org.uk/wiki/ACCUOxford

ACCU Cambridge and ACCU South Coast

Report from Ric Parkin (ric.parkin@gmail.com) and Pete Hammond (peter.hammond@baesystems.com)

Finding speakers is a major sticking point which has made it a quiet period for both these groups. Ideas for how to get speakers have been explored at an informal pub-meet in Cambridge, but more suggestions are always welcome from the wider membership. If you can recommend a speaker, propose an algorithm or a pattern for finding one, or simply have an idea for a topic you'd like to hear about, please discuss it on your regional mailing list: accu-southcoast or the newly created accu-cambridge. And if you're not already on the list, please join it; it's low volume and the ideal way to hear about – and influence – what's going on in your area

ACCU North East

Report from Ian Bruntlett (ianbruntlett@hotmail.com)

The 17th ACCU NE meeting saw 4 attendees (Ian, Steven, Michael & Jeff) looking at and discussing some Haskell code. We also had a look at some of Michael's C++ course work examples and decided the best thing to do next month would be to install C++ Builder on a PC and show Michael what C++ is all about.

If you didn't already know, Morpeth, home of Contact, where ACCU NE regularly meets, was flooded. Here is some footage of the main bridges into Morpeth being stress tested by the flood waters:

- http://uk.youtube.com/watch?v=WIQ5d0yCxmM
- http://uk.youtube.com/watch?v=dWPDfF-CE8g

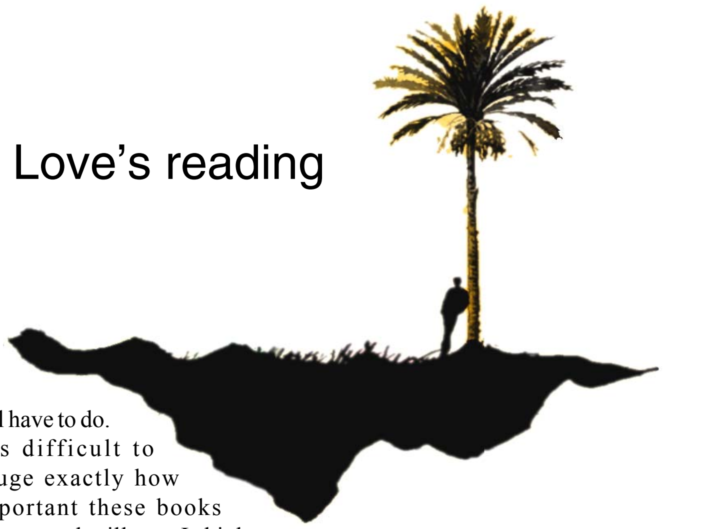
Contact was unlucky in the flooding. It happened on a Saturday meaning that our downstairs offices were flooded before anyone could get to them to move our Information Point's resources. We lost two offices and a kitchen, two laptops, a tower PC and an expensive photocopier. The net result of that is that the computer access project, that makes PCs & the internet available to Contact's members, has had to be scaled down.

If anyone would like to help, the kinds of things we need are 1) memory modules of 256MB or better, 2) windows licences (pref XP or 2000 but others are also viable) 3) redundant laptops. Or if you have something else you think we might have a use for, send an email to IanBruntlett@hotmail.com. Thanks in advance.

Despite the flooding it's business as usual and we didn't miss a single meeting. We meet at Contact on the third Saturday of the month ACCU NE; looking forward to seeing you at the next meeting.

Desert Island Books

Paul Grenyer introduces Steve Love's reading selection.



I have known Steve Love a good few years. We met at an ACCU conference (well, strictly speaking I think it was in the Dirty Duck in Stratford-upon-Avon), we've done a conference presentation together and consumed more alcohol than I care to mention. In fact with the possible exception of Alan Lenton and a couple of others whom I won't mention, I think I've consumed more alcohol with Steve Love than any other ACCU member and the amount of curry we've consumed together doesn't bear thinking about.

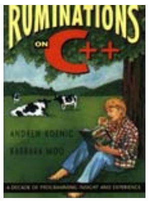
I've always felt Steve was like my formula one team mate. We have many of the same skills and have had similar experiences in our professional careers, feel the same way about a lot things and I always felt we were on a similar level. In recent years, especially when I went though my dark mobile phone and banking years, I feel Steve has pulled ahead. And interestingly enough, Steve now works for the bank I wanted to and never did.

Steve often talks of the time he filled in for the guitarist in a Whitesnake covers band. Never managing to master the guitar (probably lack of practice) I am quite jealous. Having seen Whitesnake recently get completely blown away by Def Leppard, I find myself wondering if they'd be better off replacing at least one of their (now American) axe wielding duo with Steve. Refreshingly Steve hasn't chosen any Pink Floyd, but I would of course argue that Trash and Hey Stoopid are the only Alice Cooper albums worth listening too. But that's a subject for another curry and long drinking session.

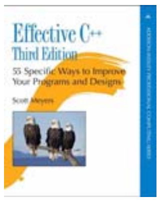
Steve is a regular conference goer, CVu and Overload writer and accu-general participant. He is part of the backbone of the organisation and we, as ACCU members, would be a lot poorer without his contributions.

Steve Love

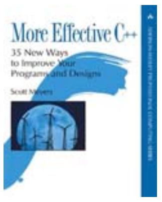
I reckon choosing technical books with which you'd be happy being washed up on a desert island is a pretty hard task. Apart from anything else, quite a lot of the tech books in my library are quite big – take the obvious titles from Stroustrup or, say, Josuttis – and would likely prevent me being washed up anywhere. Fortunately, Paul's specification for this little diversion has a get-out clause which I'll avail myself of: books that have made a big impact, or that I would take to a desert island.

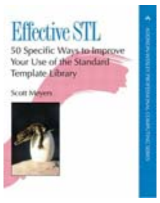


Having mentioned large books, first up is a much smaller one. *Ruminations on C++* by Andrew Koenig and Barbara Moo is so full of interesting anecdotes and enlightening experiences with C++ that its age is soon forgotten. Although much of the material pre-dates the first C++ ISO standard by nearly 10 years (some of it longer than that), it's no less relevant to today's C++ because it is as much about thinking about programming and problem solving as it is about C++ specifics.



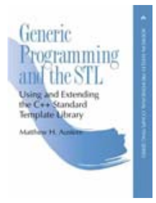
The content is based on articles and columns written over several years for such journals as JOOP and C++ Report, and those articles themselves drew on years of experience the authors had in using and teaching C++ and programming in general. It's just plain hard to find that much knowledge and insight all between the covers of one book anywhere else!





Is it cheating to cite Scott Meyers' *Effective C++* and *More Effective C++* as one selection? (Paul: No, but I'd include *Effective STL* or just take the CD instead) Well, since the copies I first read were on a single CD, and they liberally cross-reference each other, it'll have to do. It's difficult to gauge exactly how important these books were – and still are, I think – because they distill so much knowledge, and make it accessible to newbies as well as providing insights relevant to more experienced programmers. I was certainly new to C++ and Object Oriented Programming when I first got hold of these two books (the CD was borrowed) and some of the more advanced topics were, to me, revolutionary.

Having the CD with links between the different items in EC++ and MEC++ was really useful to me as someone trying to learn as much as I could really quickly. But, I still can't quite get on with reading detailed stuff on a screen, and a real book is still more comfortable to read. The CD might not cause me to sink on my swim to the desert shore, but a laptop with a large enough screen might not survive the experience.



Next stop, bigger and newer, but still 10 years old, is Matt Austern's *Generic Programming and the STL*. When I first read this, it was a bit of a revelation, talking about abstract requirements and not using inheritance <gasp!>. Although by no means a gentle introduction to STL, Austern's book explains its underpinning concepts in a way I could grasp, and put to use almost immediately. Although this is largely a reference book, covering each aspect of the STL in intricate detail, its more fundamental impact on me as a user of STL was its coverage of the abstract principles used to enable STL to work at all – for example, the idea of iterator concepts.

This book is about much more than using STL; it's about understanding enough of its design principles to extend it properly, and live within the rules. Matt Austern covers that aspect with real panache and clarity, and the fact that 'Generic Programming' is a comprehensive reference manual for the STL is almost an afterthought.

Between them, these four books probably had the most influence on my early programming career, and are all books I still refer to frequently enough that I keep them close at all times! When I first started using C++

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.

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)

Running Windows on your Mac

By Dwight Silverman, published by Peachpit Press (2008), ISBN: 978-0321535061

Reviewed by David Sykes

Recommended for people with little knowledge of the subject.

The book is split into three parts: Installing and Running Windows on the Mac, Macintosh for Windows users, and Windows for Macintosh Users.

Part 1 begins with a brief explanation of the history and differences between the Mac and PC, and why you would want to run Windows on your Mac, before outlining the three main routes available: Boot Camp, VMWare and Parallels. Chapter 1 finishes with a list of features and functionality for each option to help you to decide which route is best for you.

The rest of part 1 is split to cover each of the three options. One chapter covers Boot Camp installation. Parallels and VMware have three chapters each, covering installation, running and advanced items. Installing and running Parallels and VMWare is very simple, and the instructions are very clear and detailed, with pictures and explanations of each dialog box. Perfect if you have no knowledge of the software, or if you want something to give you confidence that you are doing things correctly. People who are confident with computers, e.g. most people in ACCU, will probably find the instructions unnecessary, although the book does mention a few setting changes that are valuable but not immediately obvious.

Part 2 covers the Macintosh for Windows Users, and is split into four chapters: 'Mac Basics', 'Inside System Preferences', 'Advanced Mac',

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

Desert Island Books (continued)

in a job, I was lucky enough to be in a team of really excellent developers, all genuinely interested in what they were up to, and eager to share their discoveries and skills with the new kid on the next desk. Not only did they share their library with me, they introduced me to ACCU, too, so a round of applause for them, please!

There have been so many other sources of information I've found indispensable that choosing just a few was difficult, but ones that really stand out include the ACCU itself – the journals, the mailing lists, the conferences and the community – the original Guru of the Week from Herb Sutter, most of which made it into the *Exceptional C++* series of books, but which started on the comp.lang.c++.moderated news group, and the on-line version of the C++ FAQ.

Time for a break, then: the novel.

There have been a few favorites over the years, but I guess the one that must take the prize is *Tai-Pan* by James Clavell. It describes and dramatises the founding of Hong Kong in the 19th century by the Opium pirates of the South China seas, and has at least one foot in the truth. Romance, swashbuckler, political thriller, historical tale and plain ol' adventure story all rolled into one, it's a novel I've lost myself in for a few days, more than once, and will again and again. The *Tai-Pan* is the big chief of a company (it turns out to be a bit of a joke made by the Chinese at the expense of the 'foreign devils', but you'll have to read this book and its successor, *Noble House* to discover why if you don't already know!) and the story revolves around the fortunes of two China Traders at logger-heads. *Tai-Pan* is part of Clavell's Asian Saga, which includes the probably better-known *Shogun*, another close contender for the choice to take to a desert island.

Finally, some music. This is particularly difficult, especially to choose just two albums from the hoards of great ones I've heard and owned. In the end, I've settled on two that, instead of being particularly indicative of my usual taste in music, hold emotional significance for me.

The first is *Da Da* by Alice Cooper, a definite piece of left-field from a guy who's made his name being, well, a bit out of the ordinary! From that album, 'Former Lee Warmer' stands out as a quite unsettling track, conjuring images of Vincent Price or Christopher Lee in some grainy black-and-white Lovecraftian creep show, and admirably captures the Alice Cooper penchant for the theatrical.

The second album is *Aqualung* by Jethro Tull, featuring some startling virtuosity from the one-legged flautist Ian Anderson. 'Cross-Eyed Mary' in particular is a rich, funny and sharply sarcastic song, although quite how Anderson manages playing a flute with his tongue wedged so firmly in his cheek remains his secret.

Next issue: Alan Lenton picks his desert island books.

and 'Mac Apps: An Overview'. Like the rest of the book it provides a basic introduction to the Mac in a clear and detailed way. This would be great for anybody new to the Mac, but is unlikely to contain much new to anybody with much previous experience. The book is written for Leopard, and includes a few items that are not relevant to previous versions.

The book finishes with a short section on Windows for Macintosh users. I found the book easy to read, and for somebody who is new to the subject, or would like something to give them a bit of confidence, then this book is great. If, however, you have any knowledge already then much of this book will cover things you already know.

Visual Studio Team System: Better Software Development for Agile Teams

By Will Stott and James W Newkirk, published by Addison Wesley (2007), ISBN: 0321418506

Reviewed by Ed Sykes

VSTS: BSDFAT is essentially a rehashing of Kent Beck's work on extreme programming as presented in *Extreme Programming Explained: Embrace Change and Test Driven Development*. The authors frame XP in the context of a project with a broken process which adopts the Team Foundation System (TFS) in order to fix it. It focuses on the usual agile practices of version control, continuous integration, automated builds, Test Driven Development, Automated Customer acceptance tests and continuous planning. All the advice for these subjects is useful and accurate and will set novices in agile practices heading in the right direction. There are also some anecdotes that help to lend legitimacy to the advice given.

This book succeeds as a worked example of a team adopting agile practices, however for someone with experience of working in an agile environment the book is not weighted enough towards team foundation system. At the end of reading this book I was disappointed to find that I hadn't learnt anything new from this book. Admittedly my appreciation of the book was tainted by my level of proficiency with my old toolset and my skill level. I was looking for something similar to Laura Wingerd's *Practical Perforce* that would explain how to use the building blocks of TFS to get more out of it. This is definitely not that kind of book. Going straight to the Kent Beck books will give you a better grounding in agile practices and the TFS documentation grounds the reader as well as this book.

This book is not going to help you to understand the Team Foundation philosophy. It's not going to help to mould you into the Team Foundation tool set or help you become a power-user. I can't recommend this book, not even to someone with a limited budget, because you can buy both of Kent's books for the same price and the TFS documentation is free.

Semantic Web for the Working Ontologist

By Allemang & Hendler, published by Morgan Kaufmann (2008), ISBN: 978-0-12-373556-0

Reviewed by Seb Rose

Recommended.

This book fills the gap between vague waffle about 'Web 3' and the very detailed W3C specs.

It starts by examining the motivation for adding semantics to the web and then dives into the stack of technologies available for modeling information, reasoning about that information and making inferences based upon that reasoning. Starting with RDF, it works coherently through RDFS, RDFS-Plus and OWL with concise and comprehensible examples and challenges. These examples amount to a pattern catalogue (a view that is further reinforced by special purpose FAQ / Index) that is invaluable to new (and not-so-new) ontologists.

Along the way you also get introduced to the three concepts that make working with the Semantic Web so different from (for example) OO modeling:

1. Anyone can say Anything about Any topic (AAA)
2. Open World Assumption (OWA) – just because you haven't seen a piece of information yet, you can't assume that it doesn't exist
3. Unique Naming Assumption (UNA) – the same entity might be known by more than one name.

A couple of 'In The Wild' chapters describe aspects of actual implementations – from Friend Of A Friend (FOAF) to the National Cancer Institute Ontology (NCIO). These chapters are interesting because they demonstrate some of the concepts described in earlier chapters and root the discussion in practical applications.

A chapter on good and bad modeling practices is also indispensable, as it lists some of the common antipatterns that crop up when OO practitioners begin working with semantic modeling. The final chapter discusses the different dialects/species of OWL and begins to scratch the surface of the formal underpinnings of the technology. It also looks forward to what might be included in the next version of OWL, which is currently being designed.

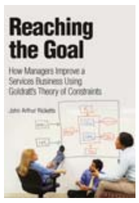
The book has its fair share of typos, but is generally well laid out and easy to follow.

Some things that you won't find in this book:

- there's no 'code'
- there's no discussion of toolsets that implement the technology
- you get introduced to N3 notation, but not RDF/XML
- there's no detail about query languages such as SPARQL.

In short, this is a book that gives you the understanding to work with the technology, not a book that describes low level details or specific implementations.

Reaching the Goal – How Managers Improve a Services Business Using Goldratt's Theory of Constraints



By John Ricketts, published by IBM Press (2007), ISBN: 0132333120

Reviewed by Frances Buontempo

Recommended

Disclaimer: I am not a manager, so cannot comment on the accuracy of any of the technical details in this book.

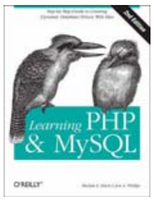
In a nutshell, Goldratt's theory of constraints (TOC) [see http://www.goldratt.co.uk/] concerns identifying bottlenecks in a process, such as a machine that has a fixed throughput, and using that as a positive way to organise the work flow. Conventional wisdom might see this as a bottleneck which must be overcome in order to increase profits. Buying more machines would allow you to produce more and thereby make more money. Instead, Goldratt's theory of constraints treats this as a basis for reorganising a service industry business's process, which according to some measures could increase profits. Pushing more products onto the market may not be the best way to go. The organisation must decide their purpose first, and can use the bottlenecks to drive what happens when. Instead of pushing products onto the market/production line, things are pulled as required. This, apparently, flies in the face of traditional approaches. The impact on sales and marketing as well as supply chain, logistics, accounting and internal organisation is also considered. The analysis, with clear, specific examples, was easy to read and thought provoking, given I have no background in this area.

The book briefly considers the theory of constraints in other types of business, and draws attention to particular problems in the service industry. In particular, predicting work flow is hard if your raison d'être is to help/provide a service on demand. Special consideration is given to these problems. For example the author shows how various measures should be adopted in order to take these problems such as unpredictable work flow into account. The author briefly mentioned agile, but emphasised TOC is a different approach, relevant when the requirements cannot be changed. TOC deals with fixed requirements.

This book is easy to read with clear examples. Frequently, the approach dictated by Goldratt's theory of constraints differs from the usual business approach taken, and this is clearly spelled out each time. It was interesting to consider how this different approach could change organisations where I have worked and if it would change things for the better. I suspect it also made me consider things from a manager's point of view and learn some of their lingo, which is always a good thing.

The only difficultly I had with the book was a periodic, in your face, evangelical tone of voice adopted to emphasise how TOC could solve almost everything. That aside, it is a good book, assuming a minimal level of background knowledge. If you need to learn about TOC this is probably a good place to start.

Learning PHP and MySQL (2nd ed)



By Michele E Davis & Jon A Phillips, published by O'Reilly (2006), ISBN: 978 0 596 51401 3

Reviewed by Alan Lenton

Not recommended.

This is an adequate, though uninspired, look through the basics of using PHP and MySQL to build dynamic web sites. Because it is trying to cover two major topics from a starter level it is unable to treat either in the depth needed for the reader to become fluent in either the use of SQL databases, or PHP.

However, the book also suffers from a serious flaw which renders it unfit to be used by those wishing to learn the subjects involved. The sample code is frequently incorrect, and this will cause endless confusion for newcomers. It is clear that insufficient attention has been paid to making sure the code is correct, which would seem to indicate that the authors haven't even tried to run the code they present. What the editors at O'Reilly were thinking of when they let this go through, I really don't know.

Implementation Patterns

By Kent Beck, published by Addison Wesley (2007), ISBN: 03211413091

Reviewed by Ed Sykes

Reading *Implementation Patterns* will help you write highly object orientated, intention revealing code. Unfortunately there isn't a simple trick to reveal. Instead values, principles and patterns are described that will get you thinking about your minute by minute implementation decisions. Sometimes, reading and writing code generates mental friction. Other times the process is effortless. If you want to understand why then buy this book.

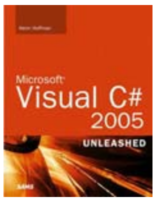
This book is short but full of useful ideas. The language used to describe the patterns is informal and easy to digest. The examples are in Java that anyone with OO experience can follow. Since you must think deeply whilst reading this book it takes longer to read than the 143 pages would suggest.

If you have invested in a language bible, the practice of programming and a design patterns book, then *Implementation Patterns* would be a good investment. You will get three things for your investment. Firstly, whatever your experience, an immediate improvement in the quality of the OO code that you write. Secondly, a set of thinking tools and a language to describe how code works at a micro level. Thirdly, an insight into the mind of Kent Beck, one of the most influential software engineers of recent times.

Whilst reading *Implementation Patterns* I began to understand why I have adopted certain conventions when crafting software. Now I also understand better why I don't like parts of the code bases that I work on. I've read the book twice and the second time was more rewarding than the first. I also expect to reference the book whilst writing more of the kind of code that I like.

This book is worth the money.

Microsoft Visual C# 2005 Unleashed



By Kevin Hoffman, published by SAMS Publishing (2006), ISBN: 0-672-32776-7

Reviewed by Omar Bashir

Recommended only as a broad reference.

This book attempts to explain the C# language as well as the complete .Net Framework using the C# language. The book initially implies that it is not an introduction to programming but later at a number of locations attempts to explain fundamental programming concepts. For example, chapter 1 explains the concept of a variable from a very basic level. The book then proceeds while assuming previous experience or knowledge of Visual Studio and a rather hasty initial introduction to the language.

A number of key terms are used without explaining what they mean. For example, the keyword base is used in chapter 5 without any previous reference. Examples are inadequately explained and so are some key concepts, which require the reader to refer to other text on the subject. The organization of the book could also have been better. The chapter on collections might have followed the chapter on generics to avoid duplication of explanation of a number of related concepts.

A number of other concepts are either vaguely or at times inaccurately explained. A striking example is the explanation of string immutability in chapter 3. It suggests that string immutability refers to storage of each string in a fixed space to optimize CLR runtime requiring operations on strings to return modified copies of the original string. Thus, in a round about manner it is suggested that strings are read-only, as precisely suggested by the MSDN topic on strings.

However, there is some interesting explanation of various aspects of the language and the framework. For example, the chapter on optimization of .Net code explains the performance impacts of autoboxing and auto unboxing in method calls. This chapter also gives an interesting introduction to the Performance Wizard and FxCop (.Net static code analysis utility).

I would recommend this book with a caution. It is a reasonable broad overview of the .Net framework using the C# language. However, any detail, even at an intermediate level requires referring to some other text on the subject (e.g., *Programming C#* by Jesse Liberty) as well as the MSDN. The book is not adequately indexed making it difficult to refer to topics using keywords, a common practice used by most developers.

Advanced AJAX

by Shawn M. Lauriat, published by Prentice Hall (2007), ISBN: 0131350641

Reviewed by John Lear

The tag line for this book's title is 'Architecture and Best Practices' and it is this that drew my attention. I have been developing web applications using AJAX technologies for almost a year and so was looking for something that would expand my knowledge beyond the basics of the `XMLHttpRequest`.

The book itself is divided into 11 chapters, each covering a different aspect of web development. Reviewing the chapter titles revealed my first disappointment. There is very little about architecture in this book. What is covered is limited to nothing more than the Model-View Controller design pattern. One of the most useful sections of this book covers the debugging tools available for the most popular browsers.

This brings me to my second complaint; a lot of the content of this book is not what I would call advanced, some of it is even what I would call basic programming skills. Three of the chapters cover Accessibility, Debugging and source code documentation. Worthy topics in themselves but not what I would expect to find bulking out an Advanced Level book.

There are some omissions as well. I would expect an Advanced book to deal with topics such as session management and to also cover some of the libraries that can be used to abstract away the many differences in how AJAX is implemented in browsers.

Despite these complaints *Advanced AJAX* is not a bad book, just poorly named. If you don't already own a book that covers this topic, you will find lots of useful information here. However, if you already own something on the subject, I would recommend that you give this book a miss.

Cross-Platform Development in C++

By Syd Logan, published by Addison-Wesley (2007), ISBN: 0-321-24642-X

Reviewed by Alan Griffiths

There are a lot of features of C++ that the standard defers to the implementation (the size of ints, the sign of char, ...) and these can trip up the unwary when moving between platforms. Further there are variations in the libraries available and the tool chain used in building software.

Syd bases his book on his experience working for Novell on various projects but most of his solutions derive from Netscape/Mozilla. In particular the development of the user interface abstraction layer. In covering this he addresses language variations, availability of libraries and portable build tools including **`make`**, **`Autoconf`**/**`Automake`** and **`Imake`**. In the latter category I'd also recommend looking at **`SCons`** and **`bjam`**.

I enjoyed reading the book – it has a pleasant conversational style and it exposed me to options I'd either not been aware of previously or had no real life information about. I was particularly pleased to see a treatment of the team dynamics of ensuring that the integration build works for all supported platforms (the examples used are OS-X, Windows and Linux).

My interest did wane somewhat when, after presenting a couple of the existing portable user interface libraries (wxWidgets and XUL) he moves on to promoting a new one he's developed to illustrate the book. I guess Syd's interest also waned as the book ends not with concluding remarks but with a listing of the build process for this Trixul project.

Out of interest I visited the Trixul website – like many open source projects this shows little activity. Similarly the book website currently shows no errata (I noticed a few discrepancies between listings and text, but they don't spoil anything).

With a few reservations (indicated above) I'd recommend this book as what it is: one man's experience of cross platform development.

Expert MySQL

By Charles A Bell, published by APress (2007), ISBN-13: 978-1-59059-741-5

Reviewed by Tim Pushman

Highly Recommended.

This book is somewhat mistitled, I was expecting a guide to installing and working with MySQL with a focus on best practices. In fact, the book is a guide to modifying MySQL at the code level to provide facilities that aren't included with a standard installation, for example, adding other storage engines or modifying the query engine. A better title may have been 'Hacking MySQL' or 'Inside MySQL'. I suspect that these titles have been taken by O'Reilly or others.

The book starts with an overview of MySQL and how it fits into the open source software paradigm. This is followed by a high level guide to the MySQL source code and guidelines on compiling and installing the software. Part one ends with setting up a test environment for MySQL.

Part two sets off into some serious code work, with a chapter on debugging changes that you might make, moving on to using MySQL in an embedded environment, creating a custom storage engine and extending MySQL with custom functions and commands. There is full code supplied, with a focus on writing code in the same style as the existing code.

Part three takes us on a guide to some of the internals of the MySQL database engine, with an good description of how an SQL query gets parsed, its internal representation and execution. There are many ideas on how to optimise the database engine, possible modifications you could make and ideas on how to test the effectiveness of any changes.

C A Bell is an excellent writer. Although I have no intention (or time) to play around with any of the ideas here, I read most of the book just out of interest in how a big project is handled and how to work with it. Bell is a senior programmer at MySQL and clearly knows his way around the source code. He also has an infectious enthusiasm for hacking on the code that had me tempted once or twice to drop everything and load up the MySQL source!

If you are involved in customising MySQL, or need to use a version of it in an embedded environment, or are simply interested in how a database engine works at the code level, this book is almost essential and highly recommended. The APress web site (http://apress.com/) have full source code and two sample chapters available.

Pascal for Science and Engineering

By James J McGregor and Alan H Watt, published by Pitman (1983), ISBN: 0273018892

Reviewed by Colin Paul Gloster

I recently have joined a group which I had preached to about Ada. I noticed that we have a large collection of Pascal books, so I planned to review some of these, but more in the context of would these help someone to learn the fundamentals of Ada instead of are they still accurate for users of current Pascal. It transpired that the person responsible for these Pascal books is only 'sort of' involved in my main project, so I might not review any more Pascal books in the near future unless that changes.

One difference I note between the languages is that though they both support providing alternative compatible names for a type, Pascal lacks the excellent facility of Ada to make otherwise identical types incompatible. Another disadvantage of using Pascal instead of Ada is that Pascal does not require an exhaustive static check that all values of a **`CASE`** condition lead into a branch (it was vaguely stated that the program would 'fail' so I checked another two Pascal books before I found a specific explanation: the result of an unaccounted for **`CASE`** condition is undefined). No Pascal function can return something which is not of a simple type. To check records for equality in Pascal, one must check on a field-by-field basis. A less important advantage of Ada is that its standard generic input/output routines can be instantiated with any enumerated type. Newer dialects of Pascal support operator overloading, but they were released much later than this book. I suspect that some of the other problems in this paragraph have also been overcome in newer Pascal dialects.

The book contains good advice against variable names only one letter long. Unfortunately, this advice is not adhered to in the book. For example, a squareroot program contains just three variables viz. **`e`**, **`r`** and **`a`**. Overall, this is a nice book but it does have a number of problematic quirks. Programming techniques tend to improve in later chapters but magic numbers can unfortunately be found in many parts of the book. I found the magic numbers 1..2001 for a range of years to be amusing. On page 87, symbolic names were replaced with hard-coded numbers (sic).

This book does not seem to be intended for financial science (that is an established term), but I wish to remark that the use of real on page 88 for a financial program is inappropriate. I know of someone who claimed to have 'seen a banking project using float for the money (under C++) because the team have been working on scientific projects before'.

It was (correctly) claimed that iterative methods are interesting for solving simultaneous linear equations, but why this is is not explained, nor is it explained that iterative is a synonym of indirect (inexact). In fairness, this is not a book on numerics and it does provide a gentle introduction with a better language than used in many other books, in order to prepare people to read an advanced numerical book but to still use a fairly decent language regardless of what is used in the numerical book.

Some programmers dislike unnecessary parentheses, but I prefer the advice in this book to have slightly more than necessary than not enough.

I rarely see other people use them in practice, but I like **`REPEAT-UNTIL`** statements (which are liberally promoted in this book) (and of course their counterparts in other languages (**`do-while`** statements in C++ and loops in Ada with **`EXIT WHEN`** at the end)). Unfortunately in Spring 2008 I encountered C++ code which showed to me that the syntactic layout of these statements in Ada, C++ and Pascal is bad. This is because the aforementioned C++ code contained **`do-while`** statements which were so large (over a screen) that I forgot that I was in a

ACCU Information

Membership news and committee reports

accu

View From The Chair

Jez Higgins
chair@accu.org

Sometimes I feel like a bit of fraud in this job. People sometimes shake my hand or clap me on the back and congratulate me about the conference, how useful they found a particular article, or some such. I smile weakly, thank them and say "well, it's really nothing to do with me".

CVu and Overload arriving through my letterbox is as much a surprise to me as to you. I have no prior knowledge of what is in each issue, and finding out is a pleasure. (I could ask in advance, I suppose, but where would be the fun in that?) If you feel so inclined the hands to shake are listed at the front - Overload editor Ric, CVu editor Gail and other guest editors, and the writers of our various articles, columns, and book reviews.

Still exhausted from last year's conference? Ready to gear up for next April? Shake the hand of Giovanni and his conference committee, and of our speakers past and future. Giovanni announces next year's keynote speakers elsewhere in this CVu and I think it's an extremely exciting line up.

If you enjoyed a particular article or found an email on a mailing list particularly helpful, drop the author a note. Better yet, nudge your colleague on the next desk and tell them about it too. As a last resort, though, I'll be happy to have my hand shaken. I might not be directly responsible, but I'm proud to be part of it.

Membership Report

Mick Brooks
accumembership@accu.org

The annual re-subscription rush is now over, and I'm happy to report that on the whole it has gone well: most members were able to renew without incident using the website. The membership numbers look about as expected: we currently have around 810 subscriptions, which is down from around 900 just before the end of August. A drop in membership of this size looks to be consistent with the turnover rates from previous years.

There were a few issues that came up, which Tim Pushman and I will be working on fixing for future renewals. Currently, people who have opted out of receiving 'announcement emails' on their website profile only receive one email message about the renewal, which arrives just after their membership expires. Those who are opted in get reminders in advance of their expiry date. It seems that most of you would expect to receive reminder emails even if opted out of announcements, which is what I suggest we do in future.

Others had trouble with the 3DSecure systems (aka Verified by Visa and Mastercard Secure) which are now being used by Worldpay, who process credit and debit card payments for us. Many banks are now making these systems mandatory[1], and Worldpay do not allow us to switch it off. As implemented on many sites, including ours, the system can look like a phishing or cross-site scripting attack, as you're redirected to your bank's (often well disguised) servers for authentication. I can't help feeling there must be a better way. Some of you may use these systems in your work: if you know how to make better use of them, then please get in touch.

Some people were left in the dark about whether their standing order payments had been processed. Some of you even paid twice in the confusion! We've improved the text of the reminder emails, so things should be clearer in the future.

We'd really love to be increasing the membership numbers, not just holding steady. If you have ideas about how to do so please get in touch, and tell friends and colleagues about us.

And if you have any questions or queries about your membership, please contact me at accumembership@accu.org

[1] http://www.theregister.co.uk/2008/08/07/verified_by_visa_compulsion/

Conference Report

Giovanni Asproni
conference@accu.org

The preparations for the next ACCU spring conference (22nd-25th April 2009) are well under way, and, I'm sure, it's going to be another great event.

For a start, we have a world class line-up of keynote speakers:

- Robert Martin (Uncle Bob). Leading software development expert, author and speaker
- Frank Buschmann. Worldwide authority on software architecture and patterns
- Baroness Susan Greenfield. Well known scientist, writer, broadcaster, and member of the House of Lords
- Allan Kelly. ACCU member, author, and expert on software project management, agile development, and patterns

The organization of the pre-conference tutorials is still going on, but I can confirm that Linda Rising, the renowned patterns expert, author, and speaker, will be teaching one.

At the time of writing, the programme is not ready – the call for proposals is not over yet – however, looking at the proposals received so far, I can already anticipate that it is going to be extremely interesting and exciting.

I suggest to check the conference web-site (www.accu.org/conference) regularly for updates because there is more to come.

I'm looking forward to seeing you all in Oxford next April.

Bookcase (continued)

`do-while` statement by the time I reached the end of one of them. This of course is a symptom of a more important problem, but a language whose syntax required the condition to be stated near the top would have helped me.

Some people might dismiss the following as pedantry, but computers do not make exceptions for people who speak too sloppily to be programmers. Mass was called 'weight' on page 15 and on page 12 the phrase 'area of the circle' was used, though of course no circle has an area. The intended readership might mistakenly accept 'multivalued function' from page 230 (a relation can be multivalued, in contrast to a function which can not).

Page 7 contains a program which is called one thing in the source code and which is incorrectly called something else in the execution example. This is a flawed book, but it is much better than other programming books intended for scientists and engineers. As for my personal aim of suggesting a Pascal book as a shortcut to beginning Ada, the lack of a facility in Pascal to make structurally equivalent types incompatible reduces how useful this book could be in the absence of an Ada textbook.