Contents

Copy Dates

C Vu 16.4: July 7th

C Vu 16.5: September 7th

Contact Information:

Editorial: Paul Johnson
77 Station Road, Haydock,
St Helens,
Merseyside, WA11 0JL
`cvu@accu.org`

Advertising: Chris Lowe
`ads@accu.org`

Treasurer: Stewart Brodie
29 Campkin Road,
Cambridge, CB4 2NL
`treasurer@accu.org`

ACCU Chair: Ewan Milne
0117 942 7746
`chair@accu.org`

Secretary: Alan Bellingham
01763 248259
`secretary@accu.org`

Membership Secretary: David Hodge
01424 219 807
`membership@accu.org`

Cover Art: **Alan Lenton**
Repro: **Parchment (Oxford) Ltd**
Print: **Parchment (Oxford) Ltd**
Distribution: **Able Types (Oxford) Ltd**

Membership fees and how to join:

Basic (C Vu only): £25

Full (C Vu and Overload): £35

Corporate: £120

Students: half normal rate

ISDF fee (optional) to support Standards work: £21

There are 6 issues of each journal produced every year.

Join on the web at `www.accu.org` with a debit/credit card, T/Polo shirts available.

Want to use cheque and post - email `membership@accu.org` for an application form.

Any questions - just email `membership@accu.org`

Reports & Opinions

Editorial

Around the start of the year, James asked if anyone fancied taking over the helm of the good ship C Vu as both his work and personal life had changed somewhat and it was becoming harder and harder to do the job as well as he had been doing it. A tough act to follow if ever there was one. In a moment of insanity, I stepped forward (or was that everyone else jumped back!) and that was that.

The last edition really marked the start of my stint at the helm and with the backup of everyone involved, we managed to get the issue out. Okay, it wasn't there in time for the conference, but that did mean that delegates would have a happy surprise for when they arrived home and everyone else could have a good read.

I would like to take this opportunity to thank James on behalf of everyone for his hard work on C Vu and wish him all the best for his future endeavours. I'm sure he won't be too far away.

C Vu and Overload

There has been some talk over the future of the two magazines (mainly at about 1am at the Conference, shortly before a fire alarm sounded! You'd be surprised at how many things get discussed in such a way). I'm happy to say that for the time being Overload and C Vu will be apart. While there are good reasons behind merging the two with a joint editorial team, it is felt that for now there is enough distinction between the two to merit keeping them as discrete entities.

Coming soon...

As with the famous BBC TV show Dr Who, a new editor brings in quite a few changes.

Pete Goodliffe has been asking for quite a few editions now for new writers to come forth and see their work in lights. His call has been answered. Not only do we have a new series on using Qt, but in the coming editions there will be tutorials on using C# and wxWidgets. The only GUI library we're now short of is GTK. Any volunteers?

There are also plans afoot to revise the reviewing system to enable us to have a more comprehensive (and consistent!) method of reviewing books. It will also mean a greater range of ratings instead of the normal four. More of that soon.

The Conference

There are many "views from the conference" in this edition. It certainly was a busy conference and in a number of ways, an eye opener for someone who is not based in a programming environment and for whom it was the first conference ever attended.

If I exclude the nightmare journey down (the M6 had been closed most of the day so instead of a nice easy 3 hour journey, it meant diverting from Merseyside to Leeds and then down the M1, cut across country to the M40 and onto Oxford – a not so nice 5 hour journey!), the aspect which has stuck with me has been the friendliness of everyone. It was so refreshing that those involved with the open source movement were happy to talk to the chap from Microsoft without the usual baiting which has been seen from time to time. The sheer professionalism of the whole set up was breathtaking.

As I've said, this was my first conference and I was also one of the first speakers (10 am session on the Wednesday). Given that I only had about 60% of my voice back (I had gone down with a rather grotty cold on the Sunday before the conference), the talk went well and to me raised something very interesting; how little industry knows about how education works and what was involved in teaching. It was a pity that the last edition of C Vu missed the conference as my editorial formed the basis of my talk. Oh well, not much can be done about that.

The venue was very good but it did have one drawback – the temperature. Being an old Victorian building, the bricks kept the heat in and so things became warm quickly. It was not helped by the lack of air conditioning. Temperatures in the keynote sessions could become unbearable. By the end of the week, I was taking in a bottle of water per half length session. If that one problem is addressed, then I'm sure that the conference next year will be an even bigger success. My other problem was the laptop didn't have a PCMCIA socket, so I couldn't make use of the wireless LAN connection provided by the hotel and there was no chance of me forking out the £70+ to PC World for a USB wireless modem. Pity, but there you go.

That may not sound very much, but in the world we live in, not having internet access for even something as simple as email when you're used to it really is something. Okay, I wasn't going cold turkey or anything as drastic as that, but not having access to the world outside (other than the very pedestrian news programmes on TV – gah! that's 5 hours old – I want my news NOW!!!!) was a pain, especially as it was so near. I think the worst part of not having email access was getting home to 1000+ emails. Not bad for 5 days (or so) of not being around. Oddly, there was also very little spam – it looks like SpamAssassin had done its job.

It was also good to finally put names to faces. Despite having been on the committee for the past year, there were so many names I knew but who I had never met. Francis Glassborow, Allan Kelly and John Merrells being just three of them. Happily, I now do know them from Adam now. Other than Francis, people were roughly what I imagined they would look like...

If you've not been to the ACCU conference before, I heartily recommend it to you for next year. It certainly will be remembered by myself for a good few years to come!

I would like to thank Paul Grenyer for his able assistance on this issue's editorial checking. Much appreciated.

Paul F. Johnson

View from the Chair

Ewan Milne `<chair@accu.org>`

I am very happy to be able to look back on the 2004 conference with a strong feeling of satisfaction. I would like to take this opportunity to thank once more everyone who helped make the event the success it was: the programme committee, sponsors, speakers, organisers and everyone who came along and contributed to the great atmosphere.

At the moment we are finalising the dates and venue for the 2005 conference, but it looks very likely that we will return to the Randolph, on the 20-23 April. It is worth saying a few words about the venue at this point: it was impossible to miss the renovation work that was still ongoing while we were there (even without the extra fun of the fire evacuations!). Needless to say, like all building projects, this was running late, and we hope that the inconvenience did not affect the enjoyment of the event too much. Once complete, the hotel will be equipped with much improved air conditioning, and car parking facilities. The feedback we have had is generally very positive about the venue and the central Oxford location, but I am always open to hearing your views on such an issue. Please mail me if you want to get in touch.

We can make the boast for this year's conference that we had a world exclusive announcement from Microsoft, made by Herb at the end of his keynote talk. Rather than repeat it directly here for those of you who were not present, I would like to point you towards `www.accu.org/conference/presentations` where you can find Herb's slides along with many other presentations from the conference. The presentations you'll find there are the ones that are, shall we say, so full of up-to-date cutting edge material that they couldn't be made ready in time to be on the conference CD.

And lastly, though I had more than one person expressing concern about how stressed out I looked during the whole four days, I can honestly say that I enjoyed every minute. See you all again next year...

A final note - please make sure you read Allan Kelly's piece on the most recent committee meeting, in which he reports on our decision to increase the membership fees. This was of course a decision not taken lightly, I hope you will recognize that while the fees have remained at the same level for as long as anyone can remember, there comes a time when an increase is inevitable.

Membership Report

David Hodge `<membership@accu.org>`

At this year's AGM a change was made to the membership year. Under the old system the membership expiry date for all members was fixed at 31st August. For existing members this will still be the case. For new members their renewal date will be set at one of 6 dates throughout the year. On joining they will be sent the most recent journal, a handbook and a welcome letter. For example, someone joining in August or September would have their expiry date set at 31st August and be sent the August issue (similarly for other pairs of months).

Making the changes has two main effects. The sending out of journals to new members is confined to one issue (at the moment the number of back issues being sent out increases as we go through the year), which will save on postage and time to do the job. We do not have to print (and pay for) extra copies to supply journals to future members.

We currently have 1068 members (end April) – 57 down on last years final total. I anticipate that we will be about 40 down by the end of the counting period.

ACCU Online Journals

Tony Barrett-Powell <tony.barrett-powell@blueyonder.co.uk>

If you have visited the ACCU web site recently you may have noticed a new link to 'Online Journals' on the home page. This area contains the initial fruits of the project to provide an online archive of the all journals printed by the ACCU. Currently only 20 or so journals are available in HTML, PDF or both. Pippa Hennessy renders the PDF files directly from the printed format (created using Quark Express) but the HTML is rendered from DocBook versions of the articles created by hand from the journal sources. DocBook is an XML DTD used for writing structured documents, for more information see: `http://www.docbook.org`

The DocBook versions have to be created manually, as it is impossible to automatically derive meaning from the printed format. For example, it is not possible to distinguish between a citetitle (the title of a cited work such as a book or journal) and italic text, or a class name and a variable.

Why generate the DocBook at all? Well, for me, it is how the DocBook version of the articles could be used that is important. I want to be able to search the articles for a particular subject say 'Java' and have all relevant ACCU articles provided in a format of my choosing, such as PDF. The DocBook version of the articles allows us to do this, but it would be difficult if not impossible with PDF or HTML versions.

The main hurdle to the completion of the project is the manual conversion of the printed format into the marked-up DocBook. The good news is that there is a small team of volunteers helping me to convert the journals: Giovanni Asproni, Richard House, Graham Johnstone, Chris Lowe, Clare Macrae, Chris Main. I hope you will join me in thanking the team for volunteering to do something that will benefit all the members of the ACCU.

Of course, we would be happy for anyone to join in, either to convert the articles to DocBook or to review the converted articles. If you would like to help out please email me at the above address and let me know what you would be able to do.

Finally, I hope you find the online journals as useful as I do, for having access to an article without needing to carry around the paper copy is already a great benefit to me.

Standards Report

Lois Goldthwaite <standards@accu.org>

ACCU members, and the organisation itself, contribute in many ways to the development of international computing standards. Here is a brief recap of what has been happening in the year since ACCU hosted (and Francis Glassborow organised) the semi annual meetings of the ISO C and C++ committees (known officially as WG14 and WG21) in Oxford. This happened in conjunction with last year's conference, so attendees had a chance to meet and talk with committee members.

Another milestone that has direct relevance to ACCU members is the publication last fall of the C and C++ standards, in hardback books from John Wiley and Sons Ltd. The complete official text of both standards is now available at a well-stocked bookshop near you, handy for a little light reading on the train or in the bath.

Three ACCU members (Francis Glassborow, Alisdair Meredith, and myself) represented the UK at this spring's recently-concluded committee meetings in Sydney. The same three delegates also participated in some additional meetings devoted to developing an international standard for an extended C++ for use in the CLI (or .Net) environment.

In regard to the standards themselves, the C committee has officially decided to stabilise the current C standard for the next several years. Committee efforts on new work items, such as hardware-supported formats for decimal floating point numbers, or additional library functions with less opportunity for allowing security breaches, will be issued as Technical Reports and will be optional for implementers of standard C.

The C++ committee, in contrast, is working hard to define the next version of the C++ standard, which is still about five years away. Their stated objectives are to make C++ easier to learn and teach, to remove inconsistencies and ugly warts, and to provide better support for generic programming and library creation, while at the same time retaining its traditional efficiency for systems programming. A Technical Report on new features to be added to the standard library is approaching final form, so programmers and implementers can gain experience with these features before they are set in standard concrete.

Let me mention another completed Technical Report from the committee here, on obtaining the best space and time efficiency from C++ code, especially when these resources are constrained. Efficiency is a primary goal of the C++ language and the standard library, and the techniques discussed in this paper will help you achieve it. I am honoured to be the project editor for this TR.

Nearly all of the committees' papers are publicly available from their websites, at `http://std.dkuug.dk/jtc1/sc22/wg14/` and `http://std.dkuug.dk/jtc1/sc22/wg21/` Open discussion takes place on newsgroups such as `comp.std.c`, `comp.std.c++`, and `comp.lang.c++.moderated`. If you would like to join the UK panels for these languages, accredited by our national standards body BSI, please write to `standards@accu.org` for more details. There is no fee to participate and no reward, other than the enjoyment of deeply technical discussions and a much better understanding of C and C++. It is emphatically NOT necessary to become an internationally-recognised expert before joining!

I mention only the international C and C++ committees in this report, but BSI sponsors panels for other standards too, including Posix, C#, Ada, Fortran, Cobol, and language-neutral domains like language independent arithmetic, design languages and internationalisation. There is also a shadow group for SC22, the parent committee of all the working groups like C and C++. If these interest you, I can supply details on joining them also.

Newbie Report from a Committee Member without a Post

Allan Kelly

To Nottingham they came. From far and wide, from the south coast, Bristol, London, Oxford, Cambridge, and closer by. They came for the ACCU quarterly committee meeting, nine of us all told, far fewer than the full committee but still enough of us to fill Alan Griffiths' living room.

I guess that most readers of this page are unaware of the amount of work it actually takes to run the ACCU; certainly I was until fairly recently. Yes I knew about the journals, I knew a little of the work behind the writing, the editing, the production, printing and distribution. And the conference too. But the actual running of the ACCU: the membership administration, the web site, the mentored developers, the money, the t-shirts, the bank accounts, the actual doing of things. There is a lot of work and very little of it involves programming - actually, none of it does.

I learnt a lot at my first ACCU committee meeting. However, there was another reason I was there, the reason I agreed to join the committee in the first place. I think there are some great opportunities for the ACCU in the next few years.

We are already expanding our standardisation work to include C# but away from programming there are quite a few members (myself included) who want the ACCU to offer a home to those moving up the career ladder. It is not enough for us "programmers" to sit back and blame management when things go wrong, most of us get involved in management sooner or later, the ACCU should grow to support our members as the grow.

And talking about growing: where is the ACCU growing? The simple answer is internationally. We need to service our international membership better.

Which brings me to the website. Yes, we all know it looks dated, needs a revamp, but did you know it also has technical problems? So another challenge for us is to update it. We're looking at our options but we've also decided we will need to spend money to get a professional website.

Talking about challenges, Neil Martin challenged the ACCU at the conference. He asked out loud in a keynote talk if the ACCU could help the development community to come up with some means of "certifying" developers' abilities. Alan Griffiths in particular seems to be rising to this.

All of these changes would be challenging enough but at the same time the Association is hitting financial problems. Our advertising revenue is down at exactly the time we need more money for magazine production, the website and new opportunities. We have decided to raise the membership fees.

Yes, I can hear the screams of pain now but please don't think the committee took this decision lightly. The fact, nobody present could actually remember when the fees where last raised. Is this true of other professional societies? The BCS? The ACM? And what about other development journals? Dr Dobbs? The C++ Users Journal?

I think if you look at the value the ACCU offers you'll find that the £10 a year extra we are asking is entirely justifiable. To lessen the pain we've decided that we will offer a £5 a year discount for anyone who agrees to pay by standing order. I know this will disadvantage overseas members who cannot set up standing orders to a UK bank account, however, overseas members actually cost the Association more because we have to pay for international postage. Don't get me wrong, we love having you as members but it is not fair for the UK members to subsidise all your postage.

So there you go, an eventful committee meeting for my first one! I don't intend to write a report every issue but I thought that as I was new to the committee it would be interesting to give you my first impressions and let you know how the ACCU is preparing for the future.

Dialogue

Letters to the Editor

The letters page seems to have vanished for quite a while, but after a comment in the last edition (as part of Francis's Scribbles column), it seems to have prompted someone to write in:

I must take issue with Francis's comment in C Vu (Vol 16, no. 2) in the paragraph just before item "More on Spam". It looks as if Francis is having a dig at James Dennett. Now, I'm pretty sure that this is not the case, however, I did overhear when the editor-to-be and Francis were talking at the conference that one thing Francis did want back was the letters section; something which had all but vanished over recent times.

When I arrived home, I read avidly Francis's comments and could hardly believe it when I saw what looked to be a dig. I am still sure that I am wrong, but if I'm not, then Francis should be thoroughly ashamed of himself for being so unprofessional.

I've put this to Francis and had the following response:

My view of being professional does not always agree with everyone else's but I think that what I write should be interpreted on the assumption that I am professional in my writing.

As a part time language lawyer I am used to people looking for unintended interpretations of text but I consider this case to be unfriendly without real justification. My aside (as indicated by the parentheses) was addressed to the Editor, who is not required to do more than proof-read my deliberations. The final part of the sentence is outside the parentheses and is, therefore, addressed to the actual reader.

It is a fair question. Over the years those that write for C Vu [note the space, it is and always has been the correct way to spell the title of this publication] have often been ill-rewarded for the time we spend writing. It has got worse over the last couple of years, but it has never been good. The simplest contribution you, the reader, can make to C Vu is to write a reasonably considered response to something that has fired your imagination or raised your ire. In this context the writer of the above at least scores over the vast majority even if I think his complaint is founded on a deliberately unfriendly reading of my words.

Francis

It seems that Francis' column has provoked more than one email... The following came from Mark Grimshaw

Dear Editor,

After reading your column in April's C Vu I decided to download the MinGW software as I to have been on the look-out for a decent (free!) development environment that runs on Windows and Linux platforms.

I downloaded version 2.05 (build date 12-12-2003) and installed it on my PC at home. On first appearances the IDE seems "polished" and at first glance it felt as though I was looking at MS Visual Studio. The software does provide most of the bare essentials one would need to start writing C++ programs although I did find that the lack of a stack trace and suitable debugging facilities (apart from a basic Quick Watch) a bit annoying. Still the product is under development so I think I may have another look in the near future. I may also have a look at the Linux version to see if it has a little bit more functionality.

Project repository

I think the idea of a project repository is an excellent one and something which I myself would be keen to use for learning. After using C++ about 3 years ago and feeling like my skills were improving quite well I have since been consigned (not through my own choice) to using other languages in the course of my daily work, most notably Visual Basic. To maintain my career prospects I thought about how I could go about getting back up to speed with C++ in my own spare time.

After revisiting my personal library of C++ texts (e.g. Effective/Exceptional C++ series) I found that my interest was wavering somewhat. I realised that my interest in the past was driven more by the prospect of delivering software to satisfy real users needs and not necessarily just by learning programming techniques. The best approach for me is to combine the two approaches to learning programming – i.e. the acquisition of techniques and their application to some real software solution. Coming up with suitable ideas to maintain interest in one's spare time is very challenging, to me at least. Not only that but different individuals' learning motivations can vary widely – some people are driven purely by the act of writing software for customers or creating software that relates to another (domain) activity and others purely because they find programming a satisfying mental exercise.

Others, such as college/university students, may be motivated simply by the prospect of gaining qualifications. Working as part of a team can also be quite inspiring. What might be boring to one person is not necessarily so to someone else. I think this means that to cater for as wide an audience as possible, any repository should ideally cater for different learning motivations – in the first instance it might be possible to list projects that serve to highlight good practice and techniques but as you pointed out in your article, challenging, practical projects are also invaluable. Although I haven't looked at them for some time, the mentored developers projects might be good avenues for people to learn. Although I haven't had a thorough look at it, Sourceforge seems to be a another possible outlet for people to work on something interesting and as part of a team.

In sum, I for one would be only too willing to pass on any ideas for projects that I come across.

Mark Grimshaw

If you have any comments or questions, please feel free to email them to `editor@accu.org`

From the Coalface

A friend approached me with a sudden desire to learn C++ in one week. It was for a job interview. She didn't know anything about programming, and she wanted to pick it up in such a short time so that she could say she had the skills when she went for the interview.

I explained to her that it would be much better to learn Python in such a short time. I showed her the Python website (`www.python.org`) and we looked through their tutorials to decide which one was best for her. Installing Python was out of the question, since she did not own a computer and we had to use the library computers (which don't let you install anything). Fortunately, though, they had some machines running Mac OS X, which has various command-line programming tools as standard, including Python and Emacs. It took me a while to persuade my friend to use Mac OS instead of Windows, and there were some teething troubles, but in the end she was able to go into the Terminal and write simple programs using Python and Emacs. I left her working through the exercises in the tutorial.

Later she called me and asked me to explain format strings and padding, which I did (while thinking that perhaps I could have picked a better tutorial), and then asked if I could write and explain a quick simulation program (it didn't take long in Python to write a simple monte-carlo queuing simulation). Meanwhile she had somehow found herself an old copy of Stroustrup's "The C++ Programming Language" and asked me to show her how to invoke a C++ compiler. "Oh dear", I thought, "now she will need a lot more help".

Needless to say, her first C++ program would never compile. She'd apparently got C++ muddled up not only with Python but also with higher-order logic (she had a slightly mathematical background) and the result was an "interesting" programming language but it wasn't C++. She'd tried to define a factorial function by writing something like this:

```
#include <iostream>
1! = 1;
for(n = 2; n < 10; n++)
  n! = n * n-1!;
cout << 10!
```

I gently explained what was wrong, gradually put the code right and compiled it, only to find that this Macintosh was somehow missing the C++ streams library (it must have been a mistake made by their remote system administration software).

"Stick to Python", I wanted to say, but no, she was determined to work through the exercises even without trying her code on a compiler. As I was quite busy, I left her to it, but not before showing her ACCU.

I don't know what happened since then. I suppose she must have lost interest after the interview.

Francis' Scribbles

by Francis Glassborow aka 'The Video Guy'

The ACCU Spring Conference 2004

At the very successful ACCU Spring Conference 2004 (Yes, the committee that replaced me did a pretty good job, apart from ensuring that I could not participate in the only item that I did not want to miss) I took on a new mantle by volunteering to video the keynotes.

The first of these was 'World Domination: The First Five Years' by Eric Raymond. He is one of those speakers who presents off the cuff and does not use slides. I have a lot of sympathy for that and have been known to opine that 'if I need slides to hold your interest, I cannot be doing a very good job.' Anyway back to the point.

We had obtained Eric's permission for videoing and I set up the camera and continued with my apprenticeship for yet another trade. About twenty minutes in Eric opened up for questions from the audience. I had several (well when don't I?) and put up my hand. Seeing it, Eric asked if there was a problem. I said no, I just wanted to ask a question. He said he would get back to me in a moment and then explained to the audience that he thought the video guy needed a moment to fix some equipment problem but it seemed he wanted to ask a question.

Clearly the ACCU Conferences are of such high calibre that even the video technicians ask technical questions about software development.

If you missed the event, you missed another extremely enjoyable one with a vast range of presentations and many other activities. On the second day we were interrupted by the fire alarm and had to make an orderly exit. Fortunately it wasn't raining and we all got a few minutes of fresh air and Spring sunshine. Apparently a workman had but a drill through some critical wiring.

The grandiosely titled "Speakers' Banquet" was a very enjoyable evening that I very nearly missed; I did not realise that the person I was talking with in the bar wasn't going. I just managed to get in after all the other speakers had been seated. It was after one o'clock in the morning when another fire alarm dragged sixty or so of us that were still socialising out into the street. I am not sure whether the hotel decided to annoy its more normal guests in order to get us out of the ballroom but I never did hear an explanation for the alarm going off.

I realise that the cost is fairly high for those that have to pay their own way but next year treat yourself to a special four days by attending the ACCU Spring Conference 2005. You will find a wide range of people who are not only intelligent and very knowledgeable about software development but are often great fun and potentially good friends.

Which leads me to:

ACCU in Melbourne

My intensive three weeks of Standards meetings started in Melbourne. Those of you who have flown to Australia will know that the most common flights to Melbourne and Sydney start at around nine in the evening and reach their destination some time before 6 in the morning. In our (my wife and I) case we left on Sunday evening to arrive at 5.45am on Tuesday (flying against the sun means that you lose a day somewhere). Alec Clews, a long-term member of ACCU, now lives in Melbourne with his wife and two children. He had noticed my itinerary and had volunteered to pick us up at the airport and take us to our hotel. This he did despite being in the throes of having his employer considering sending him for a six week job somewhere in the US. Had that come off he would have been flying out of Melbourne on the Friday. I do not know about you, but I would have been preoccupied by all the necessary arrangements. As it happened the job fell through so we had a pleasant dinner with Alec later in the week. He then took time out to reschedule our flight from Melbourne to Sydney (I had not been paying attention in the Travel Agent's and had managed to accept a 6am flight thinking it was 6pm – and had not been able to change the booking from the UK end) and then drive us out for our plane.

This kind of friendship is common in such communities as Science Fiction Fandom and it is nice to find that it has transplanted itself into the programming community as exemplified by ACCU. Indeed a couple of conference delegates independently commented that our conferences were more like Science Fiction Conventions than most technical conferences. I am glad that is so because that was my model for ACCU during my decade at the helm.

Sydney

Our two weeks in Sydney started out with a day spent with one of my oldest friends, David, who was my best man and his wife Sheila who was my wife's best friend at college. Entirely by chance their two months touring Australia happened to place them in Sydney at a time that overlapped our visit by a few days. I had to get on with Standards meetings on Sunday evening but David and Sheila took my wife off to stay overnight with the friends who were putting them up before going to visit the Blue Mountains. I gather they had a great time.

I did get some time to look around Sydney and it's a pleasant enough place with pleasant weather but I remain unimpressed by Bondi Beach (nothing that special, there are many better beaches round the world) and perhaps it is just ennui but give me Clifton Suspension Bridge over Sydney Harbour Bridge.

Anyway we did get a lot of work done in both WG21 and WG14. Several of the six papers I presented to the C++ evolution group were very well received and I was basically told to go and add some more and then bring them back for further consideration. The paper on forwarding constructors that I co-authored with Herb Sutter was interesting in that Herb and I did not agree on one important issue: when an object should be deemed to be alive. Herb wanted it to be only on completion of the whole chain of constructors whilst I was representing the UK view that it should be when the final delegated to constructor had completed.

When it came to our usual straw poll (strongly in favour, weakly in favour, weakly against, strongly against) the UK view prevailed by a substantial margin. The interesting point was that Herb was one of those giving strong support to the UK view. Technical arguments do win the day if put clearly.

That paper is the first Evolution Work Group item to go to ready status. That is, we think we know what we want and what changes need to be made to the Standard to achieve the objective. There is now a six month cooling off period after which, if there are no objections, the working paper will be modified to incorporate the proposed wording.

Now I have to work hard at my other two major proposals (unifying the initialisation syntax for object definitions, and providing an 'explicit class' in which the compiler is not allowed to implicitly generate code for such things as default constructors.)

My second week in Sydney was spent at WG14 meetings. As I have said before, this is a much smaller group with a somewhat different work structure. This is further highlighted by the fact that at the five-year review point it chose to reaffirm its Standard. That means that currently there is no work being done on a new C Standard. While WG14 could start work on a new Standard any time that it wanted to it is actually unlikely to do so for several years.

However this does not mean that there is no work to do. Quite apart from handling defect reports (i.e. fixing bugs in the Standard) it is also working on a number of technical reports. These do not have the force of a Standard but are strong statements about how we would like to see specific things done, and attempt to provide a path for extensions so that compilers remain close. We have just moved the second Technical Corrigendum (formal errata for a Standard) to ready status (i.e. we think it is right but want to take a few more months before putting it out to vote). We intend to ask that TC1 & TC2 be incorporated into the Standard and that a new corrected edition be published (as did WG21 with the C++ Standard). Note that Wiley published the C Standard corrected by TC1 though this is officially the collation of two distinct ISO documents. The difference is that next time ISO will publish a new document that is the result of applying both TC1 and TC2 to C99 as a single document ISO/IEC 9899:2005 (or perhaps 2006).

I was tasked with drafting the WG14 response to the longest outstanding Defect Report. Having listened to the Committee discussion it was clear that their well considered opinions were actually different from a commonly held view. The core of the issue is whether the bit-patterns representing a value can change without explicit access. The simple surprising answer is that WG14 is intending to stipulate that they can as long as the value does not change. The most extreme case is where an object has indeterminate value. In such a case all bit patterns are of equal status. WG14 intends to state that using arrays of unsigned char to examine the bit pattern of an indeterminate value twice does not require that the two patterns be identical.

Their reason is that it is an unnecessary burden to require that the bit patterns of indeterminate values be preserved (e.g. when paging out memory).

What about determinate values? Well there are times when these can have alternate representations; the implementation is allowed to arbitrarily change between representations (e.g. it can normalise a floating point value whilst idling for input, or more specifically, at a time when it is only accessing the object for reading a value.)

None of this should affect the working C programmer but it does have implications for compiler developers.

Worse is Better

During a newsgroup discussion of how to study C++ we got into a few exchanges on the subject of Dijkstra prognostications, in particular EWD 498, ‘How do we tell truths that might hurt’. (for more start at: `www.cs.utexas.edu/users/EWD/indexEWDnums.html`)

I have some fairly strong opinions on some of the things Dijkstra said and a feeling that despite his good intentions the way he addressed the problems often put people off. During the course of the dialogue I wrote:

> Yes, and it does not work. Practically every successful language has been developed pragmatically. Algol (both versions) were probably better designed than the contemporary FORTRAN. I hear all kinds of reasons why functional languages, Pascal, Modula (all versions), Ada, Scheme etc. are better tools but they have mostly remained in academia.
>
> On the other hand, languages designed by those who wanted to program may be theoretically flawed but they are used. Instead of telling us (or researchers) that they should be doing a better job’ it might be more effective to determine why those actually doing productive work choose something else.

One respondent gave the following url:
`http://www.dreamsongs.com/WorseIsBetter.html`

I think many of you might find a little surfing starting there would challenge your thinking. Please take a little time to visit the site and follow some of the links and then, if you find them thought provoking share your thoughts with the rest of us. Unless you want to settle down to writing a full scale article on the subject I think that a letter to the editor would be appropriate and would give our new C Vu editor a helping hand.

Making Contact

Once again lost email has surfaced, this time from Brett Fishburne, a long-term quiet contributor to ACCU. Here is some of what he had to say:

Spam and Personal Data

I’m so very disappointed. In one section you call for our ISPs to parse all of our mail and in the next you decry a loss of privacy. Well, as GMail has shown, anyone parsing everyone’s mail (for any reason) would be considered “invading privacy.”

Now, suppose that all these ISPs got busy and started parsing mail and pulling out viruses. What is their liability if they miss one? What if they don’t meet the requirement for “identifiable” for some reason (perhaps they use Macafee instead of Norton and are an hour behind)? The ISP’s job is to deliver the mail. This is like expecting the post-office to intercept chain letters. The presumption is that it would be OK for them to peruse our mail in the hopes that chain letters are removed. Now, you could make the argument that it is the post office’s responsibility to prevent bombs from going through the mail, but my understanding is that they do not take this responsibility. They generally prevent bombs from going through the mail out of the interest of protecting postal workers. Similarly, if you wrote a mail program that forwarded all of your SPAM to your ISP requesting that the sending offender be blocked, perhaps all people using that mail program could effect a “denial of service” to your own ISP and (assuming that alone wasn’t sufficient for your ISP to cut you off personally), perhaps your ISP would determine that, in the interest of preserving itself, it would prevent such viruses from ever getting to your mail program. A long-shot at best. Perhaps, if a government were to make it illegal for an ISP to forward a “known” virus...but there, again, we trip over privacy. *[Privacy is not an absolute.]*

I certainly don’t understand why companies don’t do this at the firewall. It makes no sense. Given that there are lots of people with corporate accounts and corporations are the ones who get dinged when networks get trashed, I would think that corporations would be very, very touchy on this issue and not leave it to each individual PC.

Perhaps the reason this doesn’t work at ISPs is due to performance. Can you imagine how bad Internet telephony would be if every packet was parsed for a virus? What about innocent applications (like telephony) which may contain byte codes that could be a virus if executed, but are never intended to be executed (ditto for any binary source)?

Tools

I really like TextPad. I use it for everything and have gladly paid the price requested. I like that I don’t give up EXREGEXPs for my searching and replacing and I especially like the “block select” mode which lets one select columns of data instead of just rows!

Commentary on Problem 13

You make so many obvious mistakes that I can only assume that this was done for effect. Nonetheless, I will address them:

a) You treat the pointer to the structure in the `compare` function like they are structure objects. You would need to use an arrow “`->`” instead of the period, “`.`”.

b) In the call to `qsort`, you refer to `sixeof(X)` which has both a typo and an error and should be `sizeof(struct X)`.
The original was in C++ and I forgot to change that line.

c) In the version of `qsort()` I use, it is necessary to supply the number of elements in the array as the second parameter. I can only assume that this was an oversight on your part.
Again a careless conversion problem from a C++ `sort()`.

d) `compare` is required to return an `int`, but the addition and subtraction in the final return statement could possibly exceed an `int`.

e) The `compare` statement doesn’t handle a NULL pointer (although that shouldn’t happen).

Cryptic

Here is my clue: The millenium problem in 4 digits in the correct ascending order and, of course, bookending but not naught.

The millenium problem centered around using the last two digits of a year and the problem that occurred when these were sorted. 01 was sorted in front of 99 when the desired sort was 99 then 01. The millenium in two digits is “00” or naught, but the years we want are just before “99” and just after “01” (bookending).

Another clue: What you get when you see and take away one followed by the reflection of a Downing Street address (4 digits).

“See” is a pun for the roman numeral “C” which is 100. 100 - 1 = 99. The Downing Street address is, of course, #10, which can be reflected in a mirror to “01” if the number one is written as a single line (implied by the reflective symmetry you mentioned).

Thanks

Thanks, Brett, for the time spent and I am only sorry that it did not arrive in a timely fashion. Now perhaps a few more of you could get into the dialogue. By the way, Brett also sent me 3 items for my project repository. Isn’t it time you wrote up and sent in your contribution?

Commentary on Problem 14

Somehow under the pressure of getting half a dozen papers proposing future developments for C++ it got left out. Which is fortuitous because I am about out of space.

Problem 15

```
#include<iostream>
#include<cstdio>
using namespace std;
main() {
  int n;
  int waste;
  char name[51];
  cout << "Enter any integer number...\n";
  cin  >> n;
  cout << "Enter your name...\n";
  cin  >> waste; // ‘gets’ does not read the
                 // name if this line is omitted
  gets(name);
}
```

A little examination will tell you that the above should not work. Why did it?

No space for cryptic clues this time so try:
Sounds like a perfect result when a score dine together. (2 digits)

Francis

Student Code Critique Competition 28

Set and collated by David A. Caabeiro <caabeiro@vodafone.es>

Prizes provided by Blackwells Bookshops & Addison-Wesley

Please note that participation in this competition is open to all members. The title reflects the fact that the code used is normally provided by a student as part of their course work.

This item is part of the Dialogue section of C Vu, which is intended to designate it as an item where reader interaction is particularly important. Readers' comments and criticisms of published entries are always welcome.

Before We Start

Given the amount of entries received so far (I hope this attitude doesn't decay as time goes by), I'll be as concise as possible. Please remember that you can get the current problem on the ACCU website (`http://www.accu.org/journals/`). This should be enough motivation for people who don't participate due to the short time span between the date they receive C Vu and the contest deadline. As Francis stated in the last column, other languages such as Java, C# and Python are also very welcomed, so if you have some snippet you consider suitable for publishing, drop me a line at `caabeiro@vodafone.es`

Late Submission For SCC 26

From Ian (surname not provided)

The simple answer to the student's problem is arrays in C++ being indexed from 0 he does not need the + 1 in his loop which would then read:

```
for (int i = 0; i <= len; i++)
cout << str[i];
```

However this leaves a few problems. Addressing the most obvious first `main` should return an `int` rather than a `void` (most obvious because my version of GCC will not compile the code until this is fixed). *[gcc 3.3.3 will allow* `void main()` *to be compiled. A warning is given rather than an error - Ed]*

A lot of the remaining problems stem from the use of C-style strings and changing these for STL strings will address a number of issues. In terms of the current issues related to the C strings the `arraySize` constant is unnecessary and dangerous, if `"Need help with C++."` becomes `"Need no help with C++."` in a moment of hubris and the `arraySize` value is not increased then there are potential problems, `char str[] = "Need ..."` will suffice instead. The loop given is printing the final null character which whilst not harmful is not necessary.

Initially changing to use C++ strings tidies these up a bit.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  string str = “Need help with C++.”;
  int len = str.size();
  cout << "The sentence \"Need help with
       C++.\" has " << len << “ characters.”
       << endl << endl;
  for (int i = 0; i < len; i++)
    cout << str[i];
}
```

Even this is correct but could still be improved in a few ways. The `"Need help with C++."` string is hard coded in two places and the `len` variable is could be replaced with the `str.size()` call where it is used. In a program of this size it also not necessary (though not harmful) to flush the output buffers with `endl` repeatedly, `"\n"` will suffice. (Incidentally the original loop version dropped the carriage return after each character, which behaviour is required is unclear.)

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  string str = "Need help with C++.";
  cout << "The sentence \"" << str << "\" has "
       << str.size() << " characters.\n\n";
  for (int i = 0; i < str.size(); i++)
    cout << str[i];
}
```

Finally the loop could be made more idiomatic by using string iterators rather than indexing (which would also handle the case if the string size happened to overflow the `int`).

```
  for (string::const_iterator i = str.begin();
       i != str.end(); ++i)
    cout << *i;
```

But even more so by using an STL algorithm rather than an explicit loop to perform the job, the final version becoming

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

using namespace std;

int main() {
  string str = "Need help with C++.";
  cout << "The sentence \"" << str << "\" has "
       << str.size() << " characters.\n\n";
  std::copy(str.begin(), str.end(),
            ostream_iterator<char>(cout, ""));
}
```

Student Code Critique 27 Entries

There are at least two issues with this issue's problem code. The first is finding the immediate logic error in the code that results in false positives. The second, to my mind more important issue, is the student's approach to solving the problem. It is relatively easy to learn to write syntactically correct code but that is not programming any more than writing grammatically correct English is writing poetry or a novel.

Please submit a critique of the code as it is and then append a critique of the student's approach to the problem s/he was set.

In this exercise an ugly number is defined as a compound number whose only prime factors are 2, 3 or 5. Note that a factor can repeat so 20 (2*2*5) is an ugly number. The following program outputs ugly numbers correctly, but also outputs prime numbers. I can't understand why this is happening because the pointer returned by `pf()` points to a zeroed 1st element of the array `flist` when the input to `pf()` is prime. It should fail the `while()` test and the next number should be factored. I can kludge this with a separate test for primes, but I would like to understand why it's not working as is.

```
/* find "ugly numbers": their prime factors
    are all 2, 3, or 5 */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20
/* SIZE is the max number of prime factors */
#define TWO 2
#define THREE 3
#define FIVE 5
int *pf(int number);

int main(int argc, char *argv[]) {
  int *flist,idx, n, ugly = 0;
  int start, stop;
  if(argc != 3)exit(EXIT_FAILURE);
  start = atoi(argv[1]); /*enter a range */
  stop = atoi(argv[2]);
  for(n = start; n < stop +1; ++n) {
    idx = 0;
    flist = pf(n);
```

```
      while(flist[idx]) {
        if(flist[idx]==TWO ||flist[idx]==THREE
           ||flist[idx]==FIVE) {
          ugly = 1;
          ++idx;
        }
        else {
          ugly = 0;
          ++idx;
        }
      }
      if(ugly == 1)
      printf("%d\n", n);
      free(flist);
    }
    return 0;
  }

  /* find the prime factors of number */
  int *pf(int number) {
    int *flist, quotient=0, divisor=2, idx=0;
    flist = calloc(SIZE,  sizeof(int));
    while(divisor < number + 1) {
      if(divisor == number){
        flist[idx] = quotient;
    /* add last factor to list */
        break;
      }
      if(number % divisor == 0) {
        flist[idx++] = divisor;
        quotient = number/divisor;
        number = quotient;
      }
      else ++divisor;
    }
    return flist;
  }
```

From Roger Orr <rogero@howzatt.demon.co.uk>

The initial problem which the student reports is the false positives for some primes. For example:

```
C:>ugly 30 32
30
31
32
```

where '31' should be missing - but:

```
C:>ugly 70 72
72
```

correctly doesn't display '71'. How can we find the bug?

I can think of four main ways to resolve down the problem.

a) explore the problem further
b) review the code
c) use a debugger
d) change the algorithm

These options are not exclusive - and for more complicated problems several of these might be used together.
Let's look at how the problem might be resolved in each case:

a) Try more numbers and see if a pattern appears.

```
C:>ugly 10 40
10
11 **
12
13 **
15
16
17 **
18
19 **
20
24
25
27
30
31 **
32
36
37 **
40
```

The ones with asterisks ought not to be displayed. However 23 and 29 are correctly not shown.

We may eventually realise that a prime is printed if and only if the previous number printed was 'ugly'. It is then a small step to realise that the `ugly` variable – which maintains the state of being ugly – is not being reset between iterations of the main loop.

b) Review the code

It is possible to perform a walk through of the code and discover the fault. Even for only a few lines of code however it might take a long time to find the error in failing to reset `ugly`.

c) Use a debugger

If the program is compiled for debugging and the first example is single stepped through the debugger it should be fairly easy to spot that the variable `ugly` starts the loop with the value `1` for the second number (31).

Then the `while(flist[idx])` evaluates to `false` since the first element is zero, and so the test `if(ugly == 1)` is successful.

d) Change the algorithm

We could simply cover over the problem by changing the algorithm. Many bugs are caused by boundary conditions and special cases.

In this example, `pf` returns an array with the first number zero for prime numbers which is a special case. This is because, when the number is prime, no number divides it and so the initial value if `quotient` is returned.

The first approach is to add extra code for this special case – we could change the test

```
if (ugly == 1)
```

to

```
if (ugly == 1 && flist[0])
```

and if we try this out we'll find that the bug appears to have gone.

The danger with this approach – as pointed out by the student – is that this may not have fixed the bug but simply changed the symptoms. The other danger is that the new algorithm may not be correct!

A second approach, for example, is to change the initialisation of `quotient` from

```
quotient = 0
```

to

```
quotient = number
```

then `pf` will return the input number as the first item for primes and we remove the special case for primes.

If we try this we will find that both the examples above now work correctly. However we've introduced a different bug – 2, 3 and 5 are now displayed although they are not compound numbers.

What have we done? We've tried to remove a special case – prime numbers – but they are a special case in the problem statement and not just an implementation detail. Albert Einstein is credited with the statement: "Make things as simple as possible, but no simpler", and we've hit the second part of this!

How can we fix the bug? We need to ensure that `ugly` is reset at the beginning of the loop. The simple, naive approach just adds a line `ugly=0`, after the `for` statement.

While this is adequate a better approach moves the declaration of the variable `ugly` inside the `for` loop.

Remove:

```
, ugly = 0
```

from the line after `main`, and put the line:

```
int ugly = 0;
```

directly after the `for` statement.

It is in general good to declare variables with the smallest scope possible, and close to their use. The variable `ugly` is required only within the loop and it should not persist its value between iterations. We can make the change and now the program runs as expected.

What else is wrong with the code? The corrected code works for all numbers up to 1048576 when it misbehaves. The problem is the hard-wired limit of `SIZE`. This limits the program to 19 factors (the last entry can't be used since the `flist` array is always terminated with a zero.) One easy solution is to check `stop` is less than 1048576 and report an error if not. More complicated solutions are obviously possible to remove the fixed size limit – but only if necessary.

Using `#define` to define names for numbers seems strange, to put it mildly. What is going on? The numbers 2, 3 and 5 are special for this program, and I think the programmer wanted to make that explicit.

However the names are not helpful – they literally add nothing to the program (try reading it aloud to see what I mean!)

The principle is a good one though – beware magic numbers. I would like to keep the principle but not the implementation.

For example:

```
const int factors[] = { 2, 3, 5 };
```

This defines an array containing all the factors making 'ugly' numbers, and the rest of the code does not need to refer to them again directly, so the test

```
if (flist[idx] == TWO ||
    flist[idx] == THREE ||
    flist[idx] == FIVE) {
```

can be re-written as

```
if (flist[idx] == factors[0] ||
    flist[idx] == factors[1] ||
    flist[idx] == factors[2]) {
```

Of course, once the magic numbers are in an array it might be worth generalising the check so the code could be modified very easily if you decided that, for example, '7' was also an ugly factor.

The complete check could be re-written using a nested loop as

```
ugly = 0;
int j;
for(j = 0; j < sizeof(ugly_factors) /
    sizeof(ugly_factors[0]); ++j)
  if(flist[idx] == ugly_factors[j])
    ugly = 1;
++idx;
```

While on the subject of names, I don't find the function name `pf` very informative. It might mean many things, it might be 'pointer to function' for example! Perhaps calling it '`factorize`' would be better.

There is also a problem with memory allocation. The code allocates a fixed size buffer in '`pf`' which it is the caller's responsibility to free. There are two potential problems with this, firstly it would be easy to leak memory should the caller forget to free the returned pointer and secondly memory allocation can be relatively slow. If the caller passed in the buffer instead then no memory allocation would be required since a local variable in the caller could be used for the array.

And what about performance? The code works, but is not very efficient. The '`pf`' function checks each possible divisor of the number in turn, incrementing the divisor by one each time. Try running it over the range 524288 1048575 to see how long it takes!

Does this matter? We don't know - we do not have the information about the proposed use of the program. If performance matters then a different approach to the problem will improve the speed.

One easy way to do this with the current program structure is to exit the main loop in `pf` prematurely (with `flist[0]` set to zero) as soon as the divisor exceeds the largest ugly factor, since the number cannot be ugly in this case and further factoring is not necessary for this program. This of course makes '`pf`' no longer a general purpose function to obtain all the prime factors of a number.

However, when I tried it, the time taken to calculate the entire range above changed to 1.3 seconds - the original algorithm was still running after 30 minutes (when I got bored and killed it off).

From Nevin Liber `<nevin@eviloverlord.com>`

Logic Error:

As stated, when `number` is prime, `pf(number)` returns an array whose first element is 0.
Now, let us look at the code which tests this value:

```
for(n = start; n < stop+1; ++n) {
  idx = 0;
  flist = pf(n);
  while(flist[idx]) {
    /* ... */
    /* modify ugly */
    /* ... */
  }
  if(ugly == 1)
    printf("%d\n", n);
  /* ... */
}
```

So, when `n` is prime, `flist[0] == 0`, the entire `while` loop is skipped, and `ugly` erroneously retains its value from the previous iteration of the `for` loop. Q.E.D.

Note: it actually doesn't print all the prime numbers. For instance, 23 is skipped because 22 == 2 * 11 is not ugly.

A way to fix this particular bug is to clear `ugly` at the beginning of each iteration; i.e.,

```
for(n = start; n < stop+1; ++n) {
  ugly = 0;
  idx = 0;
  flist = pf(n);
  /* ... */
}
```

Code critique:

```
#define SIZE 20 /* max number of prime factors*/
```

This is a latent bug. Besides the comment being wrong (the maximum number of prime factors is 19), if a given number has 20 or more prime factors, the code doesn't work, which can manifest itself either by incorrect results or by a memory corruption.

```
#define TWO   2
#define THREE 3
#define FIVE  5
```

These are still "magic numbers". It is clearer just to use the numbers directly.

```
int *flist, idx, n, ugly = 0;
int start, stop;
if(argc != 3) exit(EXIT_FAILURE);
```

Always use braces; it makes things far clearer.

Declare one variable per line. It is far more readable and maintainable. Unless there is a good reason not to, initialize the variables as you declare them.

```
start = atoi(argv[1]);
stop = atoi(argv[2]);
```

What happens if `argv[1]` or `argv[2]` is negative? Does the algorithm still work? What happens if the parameters aren't numbers?

```
for(n = start; n < stop + 1; ++n)
```

`n <= stop` is more readable; the `+1` tends to throw people off. What happens when `stop` is the maximum value an `int` can hold?

```
flist = pf(n);
```

Need some comments that the ownership of the array that `pf()` returns needs to be managed by the caller. This technique is highly error prone in non-trivial projects.

```
while(flist[idx]) {
  if(flist[idx] == TWO ||
     flist[idx] == THREE ||
     flist[idx] == FIVE) {
     ugly = 1;
     ++idx;
  }
  else {
     ugly = 0;
     ++idx;
  }
}
```

Note: Kudos for using pre-increment over post-increment. This is a good habit to get into, especially if you ever move to C++, since pre-increment performs at least as well as post-increment.

Since `++idx` is in both clauses of the `if` statement, factor it out. I might write this as:

```
while(flist[idx]) {
  ugly = 2 == flist[idx] ||
         3 == flist[idx] ||
         5 == flist[idx];
  ++idx;
}
```

What this `while` loop does is test the last non-zero value of `flist` to see if it is 2, 3, or 5; if it isn't, then the number isn't ugly (since it has other prime factors). This works for the problem as stated because 2, 3, and 5 are the lowest sequential primes. If, for instance, 11 were added to the ugly prime set, it isn't obvious how to extend this algorithm, as just adding `|| 11 == flist[idx]` doesn't work.

```
if(ugly == 1)
printf("%d ", n);
free(flist);
```

Good: no memory leak.

Again, use braces. Since the indentation is wrong (and something that is not enforced by the language), to the casual observer it looks like `printf()` is always called.

```
/* find the prime factors of number */
int *pf(int number)
```

Need a better interface. What happens when `number` is prime? Unless the implementation is studied, it is unexpected that no prime factors are returned for prime numbers. While you can convey this in comments (as well as that the caller has to `free()` the returned array, don't pass in a number with `SIZE` or more prime factors, etc.), it is better to make an interface that does what is expected.

```
while(divisor < number + 1) {
  if (divisor == number) {
    flist[idx] = quotient;
    break;
  }
}
```

The implementation just isn't obvious. Sometimes `quotient` means the previous quotient (when the initial value of `number` isn't prime); sometimes it means `0` (when the initial value of `number` is prime). Make these kinds of things explicit. Also, `while (divisor <= number)` is more readable.

Approach Critique:

There is a lot of messy, error-prone bookkeeping involved with calculating the prime factors. Yet the problem isn't asking for the prime factors; it is just asking for compound numbers that aren't solely made up of ugly primes.

Plus, the code which checks for ugliness is tightly coupled with the return value from `pf()`; i.e., it assumes `pf()` returns prime factors in ascending order, returns `0` as the first element in the array when the input is prime, the caller is responsible for `free()`ing the array, etc. That behaviour should all be encapsulated in a function, not spread across `main()` and `pf()`.

It also isn't data driven; if the definition of ugly primes changes, more code needs to be added. Right now, the algorithm is dependent on the ugly primes being the lowest valued and sequential. If that constraint is violated, the entire algorithm has to be rewritten. We can be more resilient.

Alternate Implementation:

We don't need to know all the prime factors; we just need to check that:
a) The only prime factors are 2, 3, and 5
b) The number has at least 2 prime factors.
If both of those hold, the number is ugly.

If I were to implement it, I would do so as follows:

```
int IsUgly(unsigned long number) {
  unsigned long uglyFactorsCount = 0;
  if(number) {
    static const unsigned long uglyPrimes[]
              = { 2, 3, 5 };
    size_t p = 0;
    for(; sizeof(uglyPrimes) /
        sizeof(uglyPrimes[0]) != p; ++p) {
      unsigned long prime = uglyPrimes[p];
      while(0 == number % prime) {
        number /= prime;
        ++uglyFactorsCount;
      }
    }
  }
  return 1 == number &&
         1 < uglyFactorsCount;
}
```

The algorithm iterates over all the primes in `uglyPrimes` and factors them out of `number`, counting (in `uglyFactorsCount`) the number of factors that are divided out along the way. If the result of `number` is `1`, then all of its prime factors are in `uglyPrimes`, and all that is left is to make sure that there were at least two factors divided out.

Some details:

```
int IsUgly(unsigned long number)
```

I pass in an `unsigned long` so I don't have to deal with negative numbers.

```
if(number)
```

If I didn't check for `0 != number`, `while(0 == number % prime)` would always be true, and there would be an infinite loop bug. This check is needed for correctness.

At this point, I started to micro-optimize. Since I have to check `number` anyway, and I know that 1 cannot be ugly (no matter what numbers are in `uglyPrimes`), there is no reason to go through the factoring code, and I could say `if(1 < number)`. Since no prime can be ugly, I could have used `if(3 < number)`, but then the value of `uglyFactorsCount` isn't technically correct in those circumstances, and even though I only care that it has a value less than 2 (which would still hold) in order for the algorithm to give the correct answer, it started feeling messy. I remembered the old adage "measure, then optimize", and abandoned this path.

```
if(number) {
  static const unsigned long uglyPrimes[]
            = { 2, 3, 5 };
```

I declare variables as close to their use as possible, and try and limit their scope as much as possible. Also, I declared it `const`, as I want the compiler to enforce that I don't modify the values in the array.

```
size_t p = 0;
for(; sizeof(uglyPrimes) /
    sizeof(uglyPrimes[0]) != p; ++p)
```

By using `sizeof` in this fashion, the code is always right no matter how many elements are in `uglyPrimes`.

Note: this works because `uglyPrimes` is the whole array. If it were passed in to the function instead, it would decay into a pointer, and the result would not be what was expected.

Note: I prefer `sizeof(uglyPrimes[0])` to `sizeof(*uglyPrimes)`, as there can be unexpected results with the latter when programming in C++ (then again, there are better ways of doing this kind of thing in C++). I used `!= p` instead of `> p`, since the former is more general when using with iterators in C++. But I digress...

Note: I tend to put l-values on the right side of expressions. In case I accidentally write something like `if (0 = a)` instead of `if (0 == a)`, I get a compile time error.

For completeness, `main()` would look something like:

```
int main(int argc, char * argv[]) {
  if(3 == argc) {
    unsigned long number
              = strtoul(argv[1], NULL, 0);
    unsigned long stop
              = strtoul(argv[2], NULL, 0);
    for(; number < stop; ++number) {
      if(IsUgly(number)) {
        printf("%lu\n", number);
      }
    }
    if(number == stop && IsUgly(number)) {
      printf("%lu\n", number);
    }
    return 0;
  }
  else {
    return 1;
  }
}
```

Note: The reason there is an extra `if (number == stop /* ... */)` is to avoid the infinite loop bug when stop contains the largest possible `unsigned long` value.

From Duncan Booth

<duncan.booth@suttoncourtenay.org.uk>

The first thing that strikes me about this code is that the structure does not obviously relate to the question. The question asks for a list of ‘ugly numbers’, so I would expect to see at least one function with the word ‘ugly’ somewhere in its name. Instead there is a function which sort of produces a list of prime factors, and the rest of the code is all bundled into `main`.

The student wonders why some additional numbers are output. This is because for any prime number the `while` loop never executes, so the variable `ugly` is never reset. This is a direct result of putting too much logic into the main function – extracting the test for ugliness to a separate function would have prevented this particular error ever happening.

The function to extract prime factors has several problems of its own: it uses a fixed length array for the factors with no check for overflow and the length chosen is too short even for 32 bit integers; if it is given a large prime number to factorise it will test every possible divisor up to the number (which could take a while) when it only needs to test divisors up to the square root of the number to determine the prime factors, or up to five to determine whether the number is ugly. The function also doesn’t do what the comment at the top says: if it is given a prime number to factorise then instead of returning the number itself it returns an empty list as a special case. This is likely to surprise anyone trying to use the function, so it would be good if the behaviour were documented, but better still if the special behaviour for primes was removed altogether.

Rather than just showing a working program, I will show the steps I use to get to a solution to the problem. I am going to use Python rather than C since that lets me ignore issues like memory management and overflows and also lets me try a variety of algorithms more quickly than I could in C. When I have a working solution, then if it needs speeding up, or if there is another reason for requiring a C solution the code can be translated to C fairly easily.

Here is my first effort in Python. It follows the student’s logic fairly closely in so far as it counts through all the numbers in the range and tests each one for ugliness. The test is a bit different though, instead of extracting all the prime factors, we just look for the factors 2, 3 and 5 and then see if there is any unfactorised residue.

The main code converts its arguments to integers using the eval functions. This makes it easy for me to test comparatively large numbers as I can type an expression as the argument (say `2**29` instead of having to enter `536870912`).

```
# File ugly.py

import sys

BADFACTORS = 2, 3, 5

def isUgly(n):
  if n in BADFACTORS or n in (0, 1):
    return False
  for factor in BADFACTORS:
    while n%factor == 0:
      n = n/factor
  return n == 1

def printUglyRange(start, end):
  print "Ugly numbers from",start,"to",end
  for i in xrange(start, end):
    if isUgly(i):
      print i
  if name__==’__main__’:
    if len(sys.argv) != 3:
      sys.exit("Usage: %s start end" %
                                sys.argv[0])
    start, end = eval(sys.argv[1]),
                           eval(sys.argv[2])
    printUglyRange(start, end+1)
```

I don’t feel this code quite expresses my intent yet, so the next step is to refactor `printUglyRange`:

```
  def uglyRange(start, end):
    for i in xrange(start, end):
      if isUgly(i):
        yield i

  def printUglyRange(start, end):
    print "Ugly numbers from",start,"to",end
    for i in uglyRange(start, end):
      print I
```

For consistency with how Python works generally, the `uglyRange` function includes its start value but excludes the end value, so 1 is added to the end value to get the same output as the student expected. The `yield` statement may be unfamiliar to C programmers: when executed it suspends execution of the generator function it is in and returns its argument to the `for` loop that invoked it; each time the `for` loop needs another value the generator is resumed exactly where it was suspended and another value calculated.

This separates the printing of the ugly numbers from the logic, and the program seems to run, but when we get up to large numbers (like `2**29`) it takes a long time to output each number. The problem is that we are searching an increasingly sparse space for each ugly number instead of going directly to the next value. A better solution would be to generate only the ugly numbers and completely ignore other numbers. First though, I suggest writing some tests in another file `testugly.py`:

```
import unittest
class TestUgly(unittest.TestCase):
UGLYTO11 = [ 4, 6, 8, 9, 10, ]
UGLY100TO130 = [ 100, 108, 120, 125, 128, ]
```

```
    def setUp(self):
      import ugly
      self.uglyRange = ugly.uglyRange

    def test0to11(self):
      expected = self.UGLYTO11
      actual = list(self.uglyRange(0, 11))
      self.assertEquals(expected, actual)

    def test100to130(self):
      expected = self.UGLY100TO130
      actual = list(self.uglyRange(100, 130))
      self.assertEquals(expected, actual)


  if name__==’__main__’:
    unittest.main()
```

The tests allow us to modify the code with confidence that it will continue working, so we can move from an implementation which is easy to understand to one that is much more complex. I tried a few different algorithms, but the version below is my favourite. It generates ugly numbers in ascending order with the numbers partitioned into three lists according to the smallest factor present in that number:

```
  import sys

  BADFACTORS = 2, 3, 5

  class UglyFifo:
    '''Stores a list of ugly numbers,
       first in first out. Fifos are
       chained so that pushes ripple
       down the chain.'''

    def __init__(self, factor, nextFifo):
      self.head, self.tail = [], []
      self.factor = factor
      if nextFifo:
        self.callNext = nextFifo.push
      else:
        self.callNext = None
      self.push(factor)

    def push(self, n):
      '''Append n multiplied by the
         factor for this fifo, then
         propagate n to the next
         queue.'''
      self.tail.append(n * self.factor)
      if self.callNext:
        self.callNext(n)

    def pop(self):
      '''Returns the oldest (i.e.
         smallest) ugly number from
         the fifo'''
      if not self.head:
        self.head, self.tail = self.tail,self.head
        self.head.reverse()
      return self.head.pop()

    def pushThenPop(self, n):
      self.push(n)
      return self.pop()


  def generateUgly():
    '''Generate a potentially infinite
       sequence of ugly numbers, in
       increasing order.'''
    values = []
    fifo = None
    def processLowest(lowest):
      '''Push a new value into a fifo,
         update the working values and
         return an ugly number’’’
      ugly, fifo = lowest
      lowest[0] = fifo.pushThenPop(ugly)
      return ugly

    for factor in BADFACTORS:
      while values:
        lowest = min(values)
        if lowest[0] > factor:
          break
        yield processLowest(lowest)
      fifo = UglyFifo(factor, fifo)
      values.append([fifo.pop(), fifo])

    while 1:
      yield processLowest(min(values))

  def uglyRange(lo, hi):
    '''Generate all ugly numbers from lo
       (inclusive) to hi exclusive.’’’
    for i in generateUgly():
      if i >= hi:
        break
      if i >= lo:
        yield i


  def main(start, end):
    print "Ugly numbers from",start,"to",end
    for i in uglyRange(start, end+1):
      print i

  if __name__=='__main__':
    if len(sys.argv) != 3:
      sys.exit("Usage: %s start end" %
                                  sys.argv[0])
    main(eval(sys.argv[1]), eval(sys.argv[2]))
```

Perhaps this seems like overkill for a simple problem, but this version of the program works up to quite large ugly numbers, albeit with a delay if the starting value is large. On my system it takes just over 10 seconds to calculate the 1.7 million values up to 10**100 which is probably fast enough for most purposes and therefore doesn’t really need converting back to C.

If the student wants to convert this to C (or is compelled to do so by the teacher), there are several ways to do it, but the one I would recommend would be to use the Python program to write all ugly numbers up to some arbitrary limit into a file (there are 1844 up to 2**32, or 13278 up to 2**64) then write a program to print ugly numbers by reading them from the file and printing those in the required range. The code could handle the numbers as strings which would allow even a simple C program to work with arbitrarily large values, but I’ll leave this as an exercise for the student.

From Margaret Wood

<margaretwood@pocketmail.com.au>

The student is quite correct in stating that any prime number should fail the `while()` test. The problem is that the value of `ugly` is only set within the `while` loop. When `n` is a prime number, the value of `ugly` remains at what it was set to on the previous trip through the `while` loop. Thus any prime number which immediately follows an ugly number will also be printed. To fix the problem it is just necessary to set `ugly` to `0` at the start of each iteration through the `for` loop.

Now to look at improvements to the code in general. I’ll start with the function to calculate prime factors. This could be a useful general purpose function if it did what the comment claims it does, namely to find the prime factors of a number. As it stands, however, it only finds the prime factors of compound numbers, returning 0 for prime numbers. The student has introduced a special behaviour for primes. I would prefer to modify the function to:

```
/* find the prime factors of number */
int *pf(int number) {
  int *flist, quotient=0, divisor=2, idx=0;
  flist = calloc(SIZE, sizeof(int));
  while(divisor < number + 1) {
    if(divisor == number) {
      flist[idx] = divisor;
      /* add last factor to list */
      break;
    }
    if(number % divisor == 0) {
      flist[idx++] = divisor;
      quotient = number/divisor;
      number = quotient;
    }
    else ++divisor;
  }
  return flist;
}
```

then add the special treatment in the main function

```
if(flist[1] == 0) {
  ugly = 0;
else {
  while flist[idx]) {
    ...
  }
}
```

Ideally the `pf()` function should also ensure that the factorisation does not go beyond the end of the array, either by checking that `idx` remains less than `SIZE`, or by choosing `SIZE` so that the array is large enough to hold the maximum possible number of factors of any number that will fit in an int. (Since `MAXINT` is always `2n-1`, where `n` will depend on how many bytes make up an `int`, any number that can be stored in an `int` can have at most (`n-1`) prime factors).

However, we don't actually need to know all the prime factors of a compound number to determine whether it is ugly or not. We just need to know whether it has at least one prime factor that is different from 2, 3 and 5. For all numbers > 5 we can determine this without storing the factors as follows.

```
while(n % 2 == 0) {
  n /= 2;
}

while(n % 3 == 0) {
  n /= 3;
}

while(n % 5 == 0) {
  n /= 5;
}

if(n == 1) {
  ugly = 1;
}
```

The numbers < 6 can be treated as special cases, 4 is the only ugly one:

```
if(n == 4) {
  ugly = 1;
}
```

(Remember that `ugly` is initialised to 0 at the start of the for loop, so 1, 2, 3, 5 will not become ugly)

Thus all that is required (apart from checking that `start` and `stop` are valid numbers, with `stop >= start`, and printing some helpful message if the user has not supplied 2 correct arguments) is:

```
/* find "ugly numbers": their prime factors
   are all 2, 3, or 5 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int n, ugly, number;
  int start, stop;
  if(argc != 3) {
    /* print helpful message */
    exit(EXIT_FAILURE);
  }

  start = atoi(argv[1]); /*enter a range */
  stop = atoi(argv[2]);
  if(start > stop) {
    n = stop;
    stop = start;
    start = n;
  }

  for(number = start; number < stop +1;
      ++number) {
    n = number;
    ugly = 0;
    if(n == 4) {
      ugly = 1;
    }
    else if(n > 5) {
      while(n % 2 == 0) {
        n /= 2;
      }
      while(n % 3 == 0) {
        n /= 3;
      }
      while(n % 5 == 0) {
        n /= 5;
      }
      if(n == 1) {
       ugly = 1;
      }
    }

    if(ugly == 1) {
      printf("%d\n", number);
    }
  }
  exit(0);
}
```

From Simon Sebright `<simonsebright@hotmail.com>`

Prologue

I always feel slightly awkward when writing these things. Ideally the student would be with the critiquer and the result would be a dialogue. Instead, they have to be monologues and as such are not tempered by little indicators of reasons why things were done or not done in a certain way.

Introduction

Regardless of what comes next (which I have at this stage already written), I have to ask why this student did not find the answer to his or her own problem. A simple mental run through the code, or use of a debugger should have left the matter high and dry. Again this is an issue about which I can only write from my immediate perspective, with no feedback. My four year old daughter has just spent the evening asking me why, why, why? I wouldn't expect a student to have the same naivety.

Part I

The immediate logic error. I haven't run the code, but an inspection reveals that the variable `ugly` will not be set for the current number if the statement `while(flist[idx])` never lets flow into the `while` loop. This is the case if the number is prime, so prime numbers have their ugliness determined by the last number to be processed. I won't proceed any further with this section because I feel that corrections to the logic are pointless without some design changes.

Part II

It's true there are some aspects of the code I could criticise on a low level, but I fear that would negate what I think are more important high level concerns. For example, I don't think `#define THREE 3` is adding anything, but my suggestions below don't use a direct replacement. I'd prefer a `static const int`, but as in the classic town's person giving directions "I wouldn't start from here". The code appears to be C, although the question did not stipulate. C++ would give us better idioms for things, as I've mentioned below, but I'll stick to a functional approach for the bulk of the critique.

I think the main reason for the occurrence of the logic error is the lack of clarity of the function `pf()`. First of all, let's not be lazy and call it `GetPrimeFactors()`. Well, what are prime factors? We have to decide whether or not they should include 1 and the number itself (if prime). I'm not familiar with the mathematical definition, so we'll err on the side of the student and say that these are not prime factors. What is `pf()` returning? Well, it's a list of prime factors. There's a lack of symmetry here. A C++ solution would have the function return a `list` containing the prime factors, here `main()` has to know how the list is allocated (if at all). Likewise, it has to know that the signal for the end of list is a zero (which can not be prime factor either way). `list<>` wins hands down.

Now, let's take a step back. What are we trying to achieve? The user wants to supply a numeric range. Within that range, we want to output all ugly numbers. Straight away, we could write some pseudo code here, not getting excited about the algorithm for prime factors, after all, I haven't needed to mention them yet:

```
main
get range to test with validation
for(each number in range)
if current number is ugly
output current number
end if
end for
end main
```

Well, now I think we need a function to see if a number is ugly:

```
bool IsUgly(int n);
```

Here, we get a chance to condense the ugliness of numbers into one place and to get rid of those silly `#defines`. A number is ugly if its prime factors are only 2, 3 or 5. Hmm, let's abstract that. A number is ugly if all its prime factors (here is where we find it handy that 1 and self are not prime factors) are to be found in the set 2, 3, 5. So, the `IsUgly()` function needs two things – a list of prime factors and a set of ugly prime factors to test against. Now, we could keep abstracting and have the `IsUgly()` function defer to a generalised list intersection algorithm passing in the ugly prime factors. For this exercise, that's probably going too far, but after all, we might change the definition of ugly (or generalise later to prettiness, etc.) How about this:

```
bool IsUgly(int n) {
  static const int UglyPrimeFactors[]
            = { 2, 3, 5, 0 };
  // again, note the need for a
  // termination condition, something not
  // necessary in a list<>

  // Get prime factors
  // See if all prime factors are one
  // of ugly prime factors. In C++,
  // this could be some sort of
  // intersection call on the list of
  // prime factors and the list of ugly
  // prime factors
}
```

Once and once only, and preferably without the need for comments, that's my motto.

I have to say that in my professional life, I rarely come across `main()` or command line arguments, so I won't go into too much detail here. There does seem to be a rather user-unfriendly approach with neither help text available nor error messages in response to "invalid" input. I think I'd like to have `main()` process input, help text and the like and if all OK, call a function to do the meat of the program, that way we have a function which can be called from `main()` here, or some UI dialog class, etc. And, test harnesses. So, we can write a test harness which tests a large range of numbers against known ugliness. Ditto for the `GetPrimeFactors()` function, which we can sort of test through the testing of `IsUgly()`, but to be sure, we'd again test it with known input/output combinations.

I've now lost the code as my wife has tidied up (put her things on the desk), so nitty gritty comments are out of the window. From what I remember, I'd like to see the word ugly mentioned in code, not comments. My pseudo code above never had an issue with uninitialised variables, I like to declare them where needed. I know in C this was not always possible, but my pseudo code was not in C and didn't have the given logic problem. I still feel like the student ought to be here and contribute or sob. By now, I've totally lost the plot, beer goggles fogging the monitor, but I'll tender one last point – don't get too excited about implementation details. For me the exciting thing is planning what details are going to be needed. It seems as if the student was hell bent on writing a wicked prime factoring function. When you are a professional programmer, that's the last of your worries, or you really are locked away in a darkened room. Rather, you need to step back, look at the view and take it in (understand it).

The Winner of SCC 27

The editor's choice is:

Nevin Liber

Please email `francis@robinton.demon.co.uk` to arrange for your prize.

Francis' Commentary

I chose the student code for this critique because there was a point that I wanted to make. The student is more competent than most in so far as coding in C is concerned. Magic numbers are avoided, all uppercase is used for pre-processor constants and uses lower case for other variables. `EXIT_FAILURE` is used after checking for the correct number of command line arguments fails. However there are a couple of points about the code:

- I do not think naming 2, 3, 5 actually adds anything in this case.
- replacing `return 0` with `return EXIT_SUCCESS` would add even more polish.
- the code does not validate that the command line arguments are actually suitable ones.

Once the logic error has been dealt with the program illustrates reasonable competence with C and from that respect I would give a good grade. However it still, in my opinion, is poor programming because it is a very heavy handed solution to the problem. That is the main point I wanted to make, there is a great deal more to programming than writing syntactically correct source code that produces the correct answer. So let me tackle the problem from the start.

The definition of an ugly number is one that has no prime factors other than 2, 3 and 5 [that is not how I would have defined an ugly number but that is irrelevant]. What the student has done is to reduce each number to its prime factors (not very efficiently as it happens, but that is another artefact of lack of clarity of thought). Then the code checks to see if any of the factors are other than the specified 2, 3 and 5.

Let us step back and think a little. What the problem requires is that we ignore all factors that are 2, 3 or 5 and see if there are any others. We can simply discard the factors in which we are not interested. That idea leads me to write the following little function:

```
int remove_factor(int number, int factor) {
  while((number % factor) == 0)
    number /= factor;
  return number;
}
```

In other words as long as the number is exactly divisible by `factor` (not a perfect name, can you suggest a better one?) reduce `number` by dividing by `factor`.

Now look at that little bit of code and decide if there are any ways in which it could fail. OK, it assumes that neither `number` nor `factor` is zero. The reason for failure will be different in the two cases. If `number` is zero the function is well defined, it simply goes into an infinite loop. If `factor` is zero we have a divide by zero error. Should `remove_factor()` handle either of those problems? Doing so is not cost free. It is a design decision, but one that can now be clearly identified and handled as the programmer chooses.

We can use this little function to determine if a given number is an ugly one:

```
bool is_ugly(int number) {
  number = remove_factor(number, 2);
  number = remove_factor(number, 3);
  number = remove_factor(number, 5);
  return (number == 1);
}
```

You might not want to use `bool` as the return type but that is your choice. Again there is the issue of whether `number` is a suitable candidate. We have two issues here: it must not be negative and it must be a compound number (i.e. it must have more than one prime factor). As the second is a defining property of being 'ugly' I think it should be dealt with here. So our function gets modified to:

```
bool is_ugly(int number) {
  if(number == 4) return TRUE;
  // direct test for smallest composite
  if(number < 6) return FALSE;
  number = remove_factor(number, 2);
  number = remove_factor(number, 3);
  number = remove_factor(number, 5);
  return (number == 1);
  // only 2, 3 and 5 were factors
}
```

Note that the second test deals with all the other cases (`number` being zero or negative) and also deals with one of the possible problems with `remove_factor()`. The other problem with `remove_factor()` is eliminated because we haven't actually tried to remove zero as a factor. Deal with problems at an appropriate level and assume that the caller of a function knows the pre-conditions (because they have been documented in the manual.)

Now I am ready to write my program:

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

int main(int argc, char *argv[]){
  if(argc != 3) {
    bad_command_line_args("Wrong number");
    return EXIT_FAILURE;
  }
  int first = atoi(argv[1]);
  int last = atoi(argv[2]);
  if((first < 1) || (first > last)) {
    bad_command_line_args("Bad range");
    return EXIT_FAILURE;
  }

  for(int i = first; i <= last; ++i) {
    if(is_ugly(i)) printf("%d\n", i);
  }
  return EXIT_SUCCESS
}
```

I haven't supplied the code for `bad_command_line_args()` but writing it should be trivial, certainly for a student who wrote the code we are critiquing. Note that this program is considerably shorter than the original even though it does a great deal to trap errors. All the code conforms to the C99 Standard though some, such as late declarations does not conform to the older standard.

The interesting feature of the above code is that it uses less C technology than the original yet, I hope you will agree, it is a much better program. One of the arts of programming is to use simple things well.

Student Code Critique 28

(Submissions to `caabeiro@vodafone.es` by July 10th)

Program 1

I'm newbie to C++ and I would like to know which would be the best (elegant and correct) solution for the following small (string) read problem with `gets`.

```
#include <iostream>
using namespace std;
#include <cstdio>

main() {
  int n;
  int waste; // needs this to work
             // (read) properly!!
  char name[51];

  cout << "Enter any integer number...\n";
  cin  >> n;
  cout << "Enter your name...\n";
  cin  >> waste; // 'gets' does not read the
                 // name without this line!!
  gets(name);
}
```

Mixing C and C++ functions doesn't seem the best way to write this program. Please provide alternatives, taking also into account its security implications.

Program 2

*If I enter 23 and 5 the answer should be 23 * 5 = 115 my answer is off by five or whatever number 2 is. I was told I am not including the first two numbers in the loop. I thought when I* `cin` *the numbers it is including them. Does anyone have any suggestions on how I can fix this?*

```
#include<iostream>
#include<iomanip>
#include<string>

using namespace std;

int main()
{
   while (1) {
int   num1, num2, total = 0;
cout<<"Enter two numbers: ";
cin>>num1>>num2;
while (num1 != 1)
cout<<setw(5)<<num1<<setw(5)<<num2<<endl;
   num1 /= 2;
   num2 *= 2;
   if (num1 % 2 != 0)
     total += num2;
}

cout<<"the total is  "<<total <<endl;
}  //End While 1

return 0;
}
```

There are numerous errors in both the code and the student's understanding. Please address these comprehensively.

Introduction To C#

Mike Bergin <author@mijobee.com>

The New, New C++

C# (pronounced See Sharp), is Microsoft's latest addition to the plethora of OO (Object Oriented) programming languages available and the premiere language for developing .NET software. In this article I will focus on the basic features of the language, future articles will cover more advanced topics such as inheritance, interfaces and attributes. After reading this series of articles you will have a basis for determining if C# is for you.

The C# programming language was developed by a team of four language designers led by Anders Hejlsberg, Microsoft's Chief C# Architect. In August of 2000 Microsoft submitted C# to ECMA International where an official standards specification was developed. In addition to Microsoft's .NET implementation of the .NET platform there are several free open source alternatives such as the Mono project, `www.go-mono.com`.

The Example Application

In this article I will examine a fictitious address book application that stores the first and last names of contacts. The code below is the example I will review.

```
 1   using System;
 2
 3   namespace Example.AddressBook
 4   {
 5     class Contact
 6     {
 7       // Internal member variables
 8       string firstName;
 9       string lastName;
10
11
12       Contact( string firstName,
                             string lastName )
13       {
14         FirstName = firstName;
15         LastName = lastName;
16       }
17
18
19       // Properties
20       string FirstName
21       {
22         get { return firstName; }
23         set { firstName = value; }
24       }
25
26       string LastName
27       {
28         get { return lastName; }
29         set { lastName = value; }
30       }
31
32
33       string GetFullName()
34       {
35         return FirstName + ‘ ‘ + LastName;
36       }
37
38
39       static void Main()
40       {
41         Contact me = new Contact(“Mike”,
                                   “Bergin”);
42         Console.WriteLine(me.GetFullName());
43         me.FirstName = “Michael”;
44         Console.WriteLine(me.GetFullName());
45         Console.WriteLine(
46            new Contact(“Dummy”,
                        “Value”).GetFullName());
47       }
48     }
49   }
```

This small piece of code provides facilities for storing and retrieving a contact's first and last name and a method of determining the contact's full name.

Variables and Literals

A variable in C# is the same thing as a variable in a mathematical equation such as `x = 1 + 2`. The value of `x` may change as the result of evaluating further lines in the equation so it is called a variable. The values `1` and `2` never change so they are called literals because they are to be taken literally. The data used by a C# application, such as a contact's first name, are stored in variables or represented by literals. A variable is analogous to a shoebox that an item may be placed in and then later retrieved from.

There are two fundamental types of data in C#, numeric and text. Numeric data represent numbers, such as 1, 2 or 84, while text data represent words such as a person's first name. The C# programming language defines 11 numeric data types, two text data types and a Boolean type. The Boolean data type is used to hold either the value true or false. The various numeric data types are listed below.

```
byte    decimal   double    float
int     long      sbyte     short
uint    ulong      ushort    bool
```

These different data types all represent numeric values however, each type implies different limitations on the values it can hold. For example the `int` data type can contain whole numbers in the range of -2,147,483,648 to 2,147,483,647. The `float` data type uses the same amount of resources as an `int` but can hold fractional values. For example, an `int` would be used to hold the number of people on a bus whereas a `float` would be used to represent a monetary value such as $10.50.

The two text data types defined by the C# language are `char` and `string`. The `char` data type represents a single character, such as the letter A. The `string` data type contains combinations of characters such as a person's first name. The code on lines 8 and 9 of the example declare variables of type `string`:

```
8       string firstName;
9       string lastName;
```

The first variable declared on line 8 is used to hold the contact's first name so it is given the identifier `firstName`. In the previous analogy this would be a shoebox used to hold the contact's first name. On line 9 a second variable of type `string` used to hold the contact's last name is declared and given the identifier `lastName`, same type of data but different meaning. These lines of code are commonly referred to as variable declarations. A variable declaration's basic anatomy consists of two parts, the type of data it will hold and the variable's name. In the shoebox analogy a variable declaration would be a label on the shoebox indicating what type of item it contains and what the item represents, i.e. “Tapes - Dance Music”.

Methods

In C# the service a piece of code provides is referred to as a method, for example printing photographs. In the example application a method named `GetFullName` starting on lines 33 through 36 determines the contact's full name by combining the first and last names:

```
33      string GetFullName()
34      {
35        return FirstName + ‘ ‘ + LastName;
36      }
```

The code on line 33 is referred to as a method signature. There are two parts to this method signature, `string` and `GetFullName`. The first part of the signature indicates if the method will return any data and if so what type of data. In the case of the `GetFullName` method, the signature declares that it will return data of type `string`. Returning data means that the variable's contents will be made available to the requestor when the method completes. In this example a `string` containing the contact's full name will be made available to the code that called the method. The last part of the signature is the method's name, `GetFullName`, and is used when requesting that the method be executed.

The body of a method contains the instructions that are executed when the method is called. The `GetFullName` method's body consists of the lines of code between the opening and closing curly braces `{}`, lines 34 through 36. The body of the `GetFullName` method performs a very simple operation and then returns the result to the caller. The actions the method performs are outlined below.

1. The contact's first name is retrieved.
2. The `+` operator is applied to the value of the first name variable and the literal `' '` which is a single empty space character. This results in a string containing the contact's first name and a trailing space.
3. The contact's last name is retrieved and the `+` operator is applied to the value resulting from the first `+` operation. The resulting value is a `string` containing the contact's first and last name separated by a single space.
4. The value resulting from the previous three steps is then returned to the code that called the method.

The `+` operator, when applied to text data, performs concatenation, the combining of two strings into one. For example if the strings "First" and "Last" are concatenated the string "FirstLast" would be created. The return keyword specifies that the value following it be made available to the code that requested this method be executed.

Properties

The C# language defines a construct called a property. A property can be thought of as a method that returns and/or sets the value of a variable. The property `FirstName` declared on lines 20 through 24 of the example provides mechanisms for getting and setting the value of the `firstName` variable:

```
20      string FirstName
21      {
22        get { return firstName; }
23        set { firstName = value; }
24      }
```

The code on line 20 is a property declaration and consists of two parts, `string` and `FirstName`. The first part of this declaration is the type of data the property manages, in this case `string` because this property manages the `firstName` variable. The last part of the property declaration is the property's identifier, `FirstName`, the name used by code when calling this property.

The property's body is the code between the opening and closing curly braces following the property declaration, lines 21 through 24. This is very similar to methods however properties have one more level of curly braces that contain the instructions. Inside the body of a property a `get` and/or `set` code block may be defined. A code block is a generic term used to refer to code between opening and closing curly braces. If the value of the property is to be retrieved a `get` code block must be defined that returns the property's value. In the case of the `FirstName` property a `get` code block is defined that returns the value assigned to the `firstName` variable. If value of the property is to be modifiable then a `set` code block must be defined. In the case of the `FirstName` property the `set` code block assigns the value provided by the calling code to the `firstName` variable.

The code blocks defined in the body of the `FirstName` property are each on a single line which illustrates the important point that a line of code in C# is terminated by a semi-colon, not a return. It is therefore possible to have multiple statements on a single line as long as there are terminating semi-colon characters to divide them. A single statement may also span multiple lines but a single word may not be split between two lines. To illustrate this point the `FirstName` property defined on lines 20 through 24 could be written:

```
public string FirstName
{
  get
  {
    return _firstName;
  }
  set
  {
    _firstName = value;
  }
}
```

The `return` keyword specifies that the value following it be made available to the calling code, just as in the `GetFullName` method. The `set` code block introduces something that we haven't covered yet but before going into detail I will examine how a property is used in code. On line 43 the `set` code block of the `FirstName` property is called. Whenever a property name is followed by the assignment operator, `=`, the `set` code block is executed. The value that follows the assignment operator, in this case "Michael", is made available to the `set` code block as a variable named `value`. Just as the `return` keyword makes a variable available to the calling code, assigning a value to a property makes the variable following the assignment operator available to code in the `set` code block.

Classes and Objects

A class is a template for creating what are called objects, it defines what variables, methods, and properties objects created from the class will have. Although many objects may be created from a single class and exist at the same time, each object is unique. If a variable belonging to one object is changed the change is not reflected in other objects created from the same class.

On line 5, a class named `Contact` is declared:

```
5     class Contact
```

This line of code is referred to as a class declaration and is composed of two parts, `class` and `Contact`. The first part indicates that a class is being defined. The following part is the class' identifier, just like variables, methods and properties each class must have a name. The code contained between the curly braces following the class declaration is called the class definition, lines 6 through 47. The class definition contains variable, method and property declarations that define the composition of the objects created from it.

There are two main steps in the object creation process, creating a copy of the object's template, the class, and initializing the new instances. Copying the object's template is taken care of behind the scenes however, each type of object may need to be initialized differently so this logic must be provided. The initialization phase of object creation requires a specialized method called a constructor. Constructors have the same name as the class that contains them and no return type. Lines 12 through 16 contain an example of a constructor:

```
12      Contact( string firstName,
                      string lastName )
13      {
14        FirstName = firstName;
15        LastName = lastName;
16      }
```

This constructor has what looks like two variable declarations between the parentheses following the constructor's name, line 12. These two variable declarations are called parameters. Parameters provide a mechanism for making variables available to a method's body, in this case the constructor's body. The values passed to the constructor can be accessed from within the constructor's body, the code block on lines 13 through 16. This is similar to both the `return` keyword and how data is made available to the `set` code blocks of properties.

Classes are multifaceted; they act as templates that define the variables, methods and properties of objects created from them as well as having variables, methods and properties of their own.

Namespaces

A namespace is a scope defined by the programmer that limits the visibility of elements, such as classes, declared within it. To simplify this idea let's look at how we as humans identify ourselves. In most cultures people are given at least two names, a surname and a given name. Our surnames provide a scope, and imply a relationship between the people within that scope, it is the namespace. My given name provides a unique identifier for me within the scope of my family, or namespace. If I were assigned only one name, let's say my given name Michael, how would you be able to tell me apart from other Michaels? Due to the sheer number of names and that we have at least two of them there are a huge number of possible combinations, cutting down on the possibility that you will run into someone with the same surname and given name. In programming this is used so that you and another programmer don't accidentally use the same name because then how would the computer know which item you are referring to.

In our example application the `Contact` class is defined in the namespace `Example`, making its full name, what is referred to as a FQN (Fully Qualified Name), `Example.Contact`. The namespace declaration is on line 3 and everything contained within the curly braces following the declaration is contained within the `Example` namespace:

```
3   namespace Example.AddressBook
```

If code in another namespace wanted to reference the `Contact` class the program would have to provide the FQN `Example.Contact`. Namespace identifiers can become very long so to save keystrokes C# provides a mechanism called importing. Importing makes all of the elements defined within a specific namespace available without having to provide their FQNs. This is illustrated on line 1:

```
1   using System;
```

On line 1 we declare that the code in this file would like to import the `System` namespace, which contains many fundamental .NET classes. All of the code contained within the `Contact.cs` file can now reference all of the classes contained in the `System` namespace without having to use their FQNs. One drawback of importing a namespace is that if you declare a class with the same name as one of the classes in the imported namespace you must still use their respective FQNs to uniquely identify them. So if there was a class named `Contact` in the `System` namespace we would need to refer to it as `System.Contact` and the class in the example as `Example.Contact` even though the `System` namespace is imported.

Execution

When a C# application is loaded the computer needs to know what instruction it should execute first. Execution of all C# application begin with a call to a `static` method named `Main`. When all of the instructions in the `Main` method have been executed the application exits. The `Main` method is referred to as the executable's entry point. The example application contains an entry point defined on lines 39 through 46. As you'll notice the method is named `Main` and is declared to be `static`.

```
39      static void Main()
40      {
41        Contact me = new Contact("Mike",
                                  "Bergin");
42        Console.WriteLine(me.GetFullName());
43        me.FirstName = "Michael";
44        Console.WriteLine(me.GetFullName());
45        Console.WriteLine(
46           new Contact("Dummy",
                      "Value").GetFullName());
47      }
```

The first line of the `Main` method defines a variable named `me` of type `Contact`. In the previous section we discussed the concept how objects are created from classes, this is how it is done in code. The constructor being called, defined on lines 12 through 16, declares that it accepts two parameter of type `string`. In this particular call the parameters to the `Contact` constructor are set to `Mike` and `Bergin`. The constructor's body assigns these two values to the `FirstName` and `LastName` properties.

The next line calls a static method named `WriteLine` on a class named `Console`. The `Console` class is contained in the `System` namespace that was imported so there is no need to use its FQN, `System.Console`. If we hadn't imported the `System` namespace this statement would be:

```
System.Console.WriteLine(me.GetFullName());
```

The `WriteLine` method of the `Console` class accepts a single parameter of type `string`. This method prints whatever `string` value it is passed to the console. In this case we pass the `string` value returned by calling the `GetFullName` method on the object referenced by the `me` variable. The result of this method call will be the string `Mike Bergin` because these are the values passed to the constructor on the previous line.

In the next statement we change the value contained by the `FirstName` property of the object referenced by the `me` variable. The value `Michael` is assigned to `FirstName` causing the property's `set` code block to be executed which storing the value following the assignment operator in the `firstName` variable.

Now that the `FirstName` property has been modified a different `string` will be printed to the console. The `FirstName` property was set to `Michael` but the `LastName` property was left unmodified. Calling the `GetFullName` method on the `me` variable again now returns the value `Michael Bergin`, reflecting the change to the `FirstName` property.

The last two lines of the `Main` method illustrate two features of C#, the ability to chain method calls and to split a single statement between two lines. The first portion of this statement calls the `WriteLine` method on the `Console` class. On this next line another object of type `Contact` is created, this time passing the values `Dummy` and `Value` to the constructor. Immediately following the call to the constructor is a call to the `GetFullName` method, illustrating how methods may be chained together alleviating the need to explicitly define a variable. The C# language is interpreted from left to write just as the English language is read so in this case the call to the `Contact` constructor returns a new instance of the `Contact` class, then `GetFullName` is called on the new instance. The value returned by this series of calls is the `string` `Dummy Value`. Since this is the last statement in the `Main` method the application exits.

Running the Example

In order to run the sample application the source code file `Contact.cs` must be created and then compiled. C# is a case sensitive language so the exact case used in the example must be preserved. The line numbers are to aid in the explanation of the code and should not be included in the source code file. The file can be created using a standard text editor such as `vi` or `emacs` on Unix and `notepad` on Windows. Once the source code file has been created it needs to be compiled.

In order to compile the software a .NET development framework must be installed. The Mono project, `www.go-mono.com`, provides a free open source .NET development framework with support readily available via mailing lists and IRC chats. Mono is available for Unix like systems and Microsoft Windows allowing developers to work with the same tool chain regardless of operating system. If you're an experienced Visual Basic programmer the Mono project has a VB.NET compiler as well.

The C# compiler distributed with the Mono .NET development framework is named `mcs`. To compile the example application execute the compiler passing the name of the source code file as a command line argument:

```
mcs Contact.cs
```

When the compiler has finished a message such as "Compilation succeeded" will be printed to the console and the compiler will exit. Now that the source code is compiled you can run the example application by running the `mono` executable passing the name of the compiled executable, `Contact.exe`, as a command line argument:

```
mono Contact.exe
```

Summary

This article introduced the basics of the C# programming language. In the next article I will cover classes and objects in greater detail illustrating how different portions of code cooperate to perform a use service. For errata and other information visit `www.mijobee.com`. Please email me with any comments or suggestions at `author@mijobee.com`.

Mike Bergin

Acknowledgments

Special thanks to Paul Grenyer for his invaluable feedback.

Professionalism in Programming #26

The need for speed (part three)

by Pete Goodliffe `<pete@cthree.org>`

Technological progress has merely provided us with more efficient means for going backwards.

Aldous Huxley

In my previous article we learnt the important optimisation process, the steps that ensure any optimisation really is worthwhile. Being eager code monkeys, I can see that you're all desperate for practical code optimisation techniques.

So here they are – in this final part we'll investigate specific code techniques for optimisation. Just don't tell anyone that I showed you. To redress the theological balance, we'll also see how to *avoid* optimisation in the first place.

Optimisation techniques

OK, we've avoided it for long enough – now it's time to look at some of the really gory optimisation details. You've proved that your program performs badly, and have found the worst code culprit. Now you need to whip it into shape. What can you do?

There's a palette of optimisations to choose from. Which is the most appropriate will depend on: the exact cause of the problem, what you're trying to achieve (e.g. increase execution speed, or reduce code size), and how much of a performance improvement is required. We'll look at some examples of specific optimisations, but in any given situation you probably won't have much of a choice – the best modification will be obvious.

I've grouped these optimisations into two broad categories: *design* changes and *code* changes. Generally, a change at the design level will have a far more profound effect on performance than a code level tweak. An inefficient design can strangle quality more than a few dodgy lines of source code, so a design fix – whilst harder – will have a bigger payoff.

Most often, our goal is to increase execution speed. The speed-based optimisation strategies are to:

- speed up slow things,
- do slow things less often, or
- defer slow things until you really need them.

The other common optimisation goals are to reduce memory consumption (mainly by changing the data representation, or by reducing the amount of data accessed at once), or to reduce executable size (by removing functionality, or by exploiting commonality). As we'll see, these goals often conflict – most speed increases come at the expense of memory consumption, and vice versa.

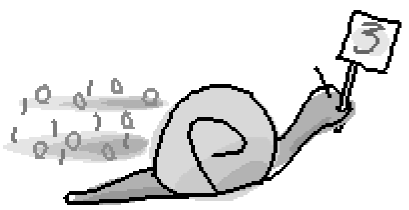
Design changes

These are the *macro* optimisations, the fixes on a large scale. Here we're looking at the internal design of a module. This doesn't address the highest level of performance degradation – *architectural* deficiencies will have the most serious impact on the performance of a system. Unfortunately these are unlikely to be fixable, since so much code depends on them. Hardware bottlenecks are probably the worst – I worked on an embedded product that was eventually abandoned because the memory subsystem was fundamentally incapable of delivering acceptable performance.

Performance-sapping bad design is also hard (or uneconomical) to fix for similar reasons. We end up papering over the cracks, employing small code level fixes instead. The nearer a project is to a release deadline, the less likely you are to perform design changes; the risk is too great[1]. When brave enough, the kinds of design optimisation we can perform include:

- **Remove functionality** – the quickest code is code that doesn't run at all. A function will be slow if it is doing too many things, some of which are unnecessary. Cut out the superfluous stuff. Move it elsewhere in the program. Defer all work until it's really necessary.
- **Exploit parallelisation**. Use threading to prevent one action being serialised after another.
- **Avoid or remove excessive locking**. It inhibits concurrency, generates overhead and often leads to deadlock. Employ static checking to prove which locks are necessary and which aren't.

1 Sadly, it's often only near project deadlines that anyone notices performance isn't good enough.



- **Compromise design quality to gain speed**. For example: reduce indirection and increasing coupling. You can do this by breaking encapsulation, leaking a class' private data through it's public interface. Knocking down module barriers will cause irreparable damage to the design. If possible, try a less disruptive optimisation mechanism first.
- **Change the data storage format**, or its on-disk representation to something more suited to high speed operation. For example: speed up file parsing by using a binary format. Transmit or store compressed files to reduce network bandwidth.
- **Add layers of caching or buffering**, to enhance slow data access or prevent lengthy recalculations. Precompute values that you know will be needed, and store them for immediate access.
- **Create a pool of resource** to reduce the overhead of allocating objects. For example: preallocate memory, or hold a selection of files open rather than repeatedly opening then closing them. This technique is particularly used to speed up memory allocation; older OS memory allocation routines were designed for simple non-threaded use. Their locks stall multithreaded applications, leading to abysmal performance.
- **Sacrifice accuracy for speed** if you can get away with it. Dropping floating point precision is the obvious example. Many devices have no FPU (Floating Point Unit hardware), and instead employ slower software FPU emulation. You can switch to fixed point arithmetic libraries to bypass a slow emulator, at the expense of numeric resolution. This is particularly easy in C++, by taking advantage of its abstract data type facilities.
  Accuracy is not solely due to your choice of data types; this tactic can run far deeper, to your use of algorithms or the quality of your output. Perhaps you can let the user make this decision – allow them to select *slow but accurate* or *fast but approximate* operation modes.
- **Avoid overuse of exceptions**. They can inhibit compiler optimisations[2], and when used too frequently will hamper timely operation.
- **Forgo certain language facilities** if it will save code space. Some C++ compilers allow you to disable RTTI and exceptions, with a consequent saving in executable size.

The major design level optimisations involve improvements in *algorithms* or *data structures*. Most speed degradation or memory consumption is down to a bad choice of one or both, and a subsequent change will rectify this.

Algorithms have a profound impact on the speed of execution. A function that works acceptably in a small local test may not scale up when Real World data gets thrown at it. If profiling shows that your code spends most of its time running a certain algorithm, you must make it run faster. One approach is at the code level, chipping small improvements from each instruction. A better approach is to replace the entire algorithm with a more efficient version.

Consider this realistic example. A particular algorithm runs a loop 1000 times. Each iteration takes 5 milliseconds to execute. The operation therefore completes in around 5 seconds. By tweaking the code inside the loop, you can shave 1 millisecond from each iteration – that's a saving of 1 second. But instead, you can plug in a different algorithm, where an iteration takes 7 milliseconds, although it only iterates 100 times. That's a saving of almost 4.5 seconds – significantly better.

For this reason, prefer to look at optimisations that change fundamental algorithms, not that tweak specific lines of code. There are many algorithms to chose from in the computer science world, and unless your code is particularly dire you'll always gain the most significant performance improvements by selecting a better algorithm.

Data structures are intimately related to your choice of algorithms; some algorithms require certain data structures, and vice versa. If your program is consuming far too much memory then changing the data storage format may improve matters, although often at the expense of execution speed. If you need to search a list of 1000 items quickly, don't store them in a linear array with *O(n)* search time; use a binary tree with *O(log n)* performance.

2 `try/catch` blocks act, like functions, as a barrier to an optimiser. It's not possible to look through the barrier to perform optimisation, so some potential speed ups will be lost.

Complexity Notation

Algorithmic complexity is a measure of how well an algorithm scales – how long it takes in proportion to the size of input. It's a *qualitative* mathematical model, allowing you to quickly compare the performance characteristics of different implementation approaches. It doesn't measure exact execution time (this is highly dependent on CPU speed, OS configuration, etc).

Complexity is determined by the amount of work an algorithm must perform: the number of basic operations it executes. A 'basic operation' is something like: an arithmetic operation, an assignment, a test, or a data read/write. Algorithmic complexity doesn't count the exact number of operations performed, just how this value relates to the problem size. We are usually interested in the *worst case* performance of an algorithm, the most work that will ever need to be done. A good comparison looks at the *best case* and *average* time complexity as well.

Algorithmic complexity is expressed using *Big O* notation, invented by the German number theorist Edmund Landau. For a problem with input size *n*, it might have a complexity of:

O(1): *Order 1* This is a *constant time* algorithm. No matter how large the input set, it always takes the same amount of time to complete the task. This is the best performance characteristic possible.

O(n): *Order n* A *linear time* algorithm's complexity rises in line with the input size. Searching a linked list will involve visiting more nodes as the list size grows; the number of operations is directly related to the size of the list.

$O(n^2)$: *Order n squared* This is where performance really begins to get bad: complexity is increasing faster than the rate of input growth. A *quadratic time* algorithm may seem fine when you give it a small set of data, but large data sets take a seriously long time. The bubblesort algorithm is $O(n^2)$.

Of course, complexity may be of any order; the quicksort algorithm averages *O(n log n)*. This is worse than *O(n)*, but far better than $O(n^2)$. A simple optimisation route for a slow bubblesort algorithm is to replace it with a quicksort algorithm, especially since there are plenty of freely available quicksort implementations.

These big O expressions don't include constants or low-order terms. You'll never see any talk about a complexity of *O(2n+6)*. When *n* gets large enough, these constants and low-order terms dwarf into insignificance.

Selecting a different data structure seldom requires you to implement the new representation yourself. Most languages come with library support for all common data structures.

Code changes

And so now we creep anxiously on to the really disgusting stuff: the *micro* level, small scale, short sighted, code tweaking optimisations. There are many ways to molest source code for the sake of performance. You must experiment to see what works best in each situation – some changes will work well, others will have little, or even negative effect. Some may prevent the compiler's optimiser from performing it's task, producing startlingly worse results.

The first task is easy: turn on compiler optimisation, or increase the optimisation level. It often gets disabled for development builds since the optimiser can take a very long time to run[3], increasing the build time of large projects by an order of magnitude. Try configuring the optimiser, and test what affect this has. Many compilers allow you to bias optimisation towards extra speed or reduced code size.

There are a collection of very low level optimisations that you should know about, but generally avoid. These are the kind of changes that a compiler is able to perform for you; if you've switched the optimiser on, it'll be looking in these areas. When applied by hand, they tend to butcher your code's readability the most, since they warp the fundamental logic out of shape. Only consider using one of these optimisations if you can *prove* it's really required, and that there are no alternatives:

Loop unrolling

For loops with very short bodies, the loop scaffolding may be more expensive than the looped operation itself. Remove this overhead by flattening it out – turn your 10 iteration loop into 10 consecutive individual statements.

Loop unrolling can be done partially; this makes more sense for large loops. You can insert four operations per iteration, and increment the loop counter by four each time. This tactic gets nasty if the loop doesn't always iterate over a whole number of unrolls.

Code inlining

For small operations, the overhead of calling a function might be prohibitive. Splitting code into functions brings significant benefits: clearer code, consistency through reuse, and the ability to isolate areas of change, yet it can be removed to increase performance. This process merges the caller(s) and the callee.

There are a number of ways to do this. With language support, you can request it in the source code (in C/C++ using the `inline` keyword); this method preserves a lot of the code's readability. Otherwise you have to merge the code yourself, either by duplicating the function over and over again, or using a preprocessor macro to do the work for you.

It's hard to inline recursive function calls – how would you know when to stop inlining? Try to find alternative algorithms to replace recursion.

Often inlining opens the way for more code level optimisations to be performed, that were not previously possible across a function boundary.

Constant folding

Calculations involving constant values can be computed at compile time, to reduce the amount of work done at run time. The simple expression `return 6+4;` can be reduced to `return 10;`. Carefully ordering the terms of a large calculation might bring two constants together enabling them to be reduced into a simpler subexpression.

It's unusual for a programmer to write something as obvious as `return 6+4;` However these sorts of expression are common after macro expansion.

Move to compile time

There is more you can do at compile time than just constant folding. Many conditional tests can be proved statically, and removed from the code. Some kinds of test can be avoided altogether; for example, remove tests for negative numbers by using unsigned data types.

Strength reduction

This is the act of replacing one operation with an equivalent that executes faster. This is most important on CPUs with poor arithmetic support. For example: replace integer multiplication/division with constant shifts or adds; `x/4` can be converted to `x<<2` if it's faster on your processor.

Subexpressions

Common subexpression elimination avoids the recalculation of expressions whose values have not changed. In code like this:

```
int first  = (a * b) + 10;
int second = (a * b) / c;
```

The expression `(a * b)` is evaluated twice. Once is enough. You can factor out the common subexpression, and replace it with:

```
int temp   = a * b;
int first  = temp + 10;
int second = temp / c;
```

Dead code elimination

Don't write needless code; prune anything that's not strictly necessary to the program. Static analysis will show you the functions that are never used, or the sections of code that will never execute. Remove them.

Whilst the above are particularly distasteful code optimisations, the following ones are slightly more socially acceptable. They focus on increasing program execution speed.

- If you find that you're repeatedly calling a slow function, don't call it so often. Cache its result and reuse this value. This might lead to less clear code, but it will run faster.
- Reimplement the function in another language. For example: rewrite a critical Java function in C using the JNI (*Java Native Interface*) facility. Conventional compilers still beat JIT code interpreters for execution speed.

3 It has to do complex inspection of the parsed code to determine the set of possible speed ups, and select the most appropriate ones.

Don't naively assume that one language is 'faster' than another – many programmers have been surprised how little difference using JNI makes. It's been commonly claimed that OO languages are far slower than their procedural counterparts. This is a lie. Bad OO code can be slow, but so can bad procedural code. If you write 'OO' style code in C it's likely to be slower than good C++; the C++ compiler will generate better tuned method dispatch code than your attempts.

- Reorder the code for improved performance.
- Defer work until it's absolutely necessary. Don't open a file until you're about to use it. Don't calculate a value if you might not need it, wait until it's wanted. Don't call a function yet if the code will work without it.
- Hoist checking further up the function. After having done a lot of work, it's a waste to perform a test that might lead to early return. Make the check sooner to avoid all the previous work.
- Move invariant calculations out of a loop. The most subtle source of this problem is a loop condition. If you write: `for(int n = 0; n < tree.numApples(); ++n)`, yet `numApples()` manually counts one thousand items on every call, you'll have a very slow loop. Move the count operation before the loop:

```
int numApples = tree.numApples();
for (int n = 0; n < numApples; ++n) {
  ... something ...
}
```

- Use *lookup tables* for complex calculations; trading time for space. For example: rather than write a set of trigonometric functions that individually calculate their values, precalculate the return values and store them in an array. Map input values to the closest index into this array.
- Exploit *short circuit evaluation*. Make sure that the tests likely to fail are placed first to save time. If you write a conditional expression: `if (condition_one && condition_two)` make sure that `condition_one` is statistically more likely to fail than `condition_two`.
- Don't reinvent the wheel – reuse standard routines that have already been performance tuned. Library writers will have already honed their code carefully. But be aware that a library may have been optimised for different goals than yours; perhaps an embedded product was profiled for memory consumption, not for speed.

Size-focused code level optimisations include:

- Produce compressed executables, that unpack their code before running. This doesn't necessarily affect the size of the running program, but it reduces the storage space required[4]. This might be important if your program is stored in limited flash memory.
- Factor common code into a shared function.
- Move seldom used functions out of the way. Put them into a dynamically loaded library, or into a separate program.

Writing efficient code

If the best approach is to not optimise, how can we avoid any need to improve code performance? The answer is to *design for performance*, planning to provide adequate quality of service from the outset, rather than trying to whittle it out at the last minute.

Some will argue that this is a dangerous road to follow. Indeed, there are potential hazards for the unwary. If you try to optimise as you go along, you'll write at a lower level than needed; you'll end up with nasty, hacky code full of low-level 'performance' enhancements and special 'back door' exploits.

How do we reconcile these seemingly opposing views? It isn't hard, because they're not actually at odds. They are two complementary strategies:

- write efficient code, and
- optimise code later.

If you make a point of writing clear, good, efficient code *now* you will not need to perform heavy optimisations later. Some claim that you don't know whether any optimisation is necessary at first, so you should write everything *as simply as possible*, and only optimise when profiling proves that there is a bottleneck.

4 It may have the pleasant side-effect of decreasing program start-up time; a compressed executable will load from disk much faster.

Pessimisations

Without careful measurement you can easily end up writing 'optimisations' that are not at all optimal. A perfectly good optimisation for one situation might turn out to be a performance disaster in another. Here's a little case study. Exhibit A: The *copy on write* string optimisation.

This was a common optimisation applied to C++ standard library implementations. Programs which performed intensive string manipulation experienced a massive overhead when copying long strings, both in terms of execution speed and memory consumption. Copying large strings means duplicating and shoveling around large quantities of data. Many string copies are automatically generated, temporary objects that are created and then thrown away shortly after – never actually modified. The expensive copy operation is an unnecessary cost.

The copy on write (COW) optimisation turns the string data type into a form of *smart pointer*; the actual string data is held in a (hidden) shared representation. The string copy operation now only has to perform an inexpensive 'smart pointer' copy (attaching a new smart pointer to the shared representation), rather than duplicate the entire string contents. Only when you make a modification to a shared string is the internal representation copied, and the smart pointer remapped. This optimisation avoids a large number of unnecessary copy operations.

COW worked well in single threaded programs; it was shown to speed up performance greatly. However, a problem became apparent when multithreaded programs used COW strings. (Indeed, this problem also manifests in single threaded programs, if the COW string class is built with multithreading support). Many implementations performed very conservative thread locking around the copy operations – these locks become a *major* bottleneck. Suddenly a lightning fast program slowed down to a crawl. The COW optimisation proved to be a serious pessimisation.

Far better multithreaded performance was achieved by reverting to 'classic' string implementations, and writing more careful code that reduced automatic string copying. Thankfully, library implementors now provide more intelligent versions of the string class, which are both thread safe and fast.

This approach has obvious flaws. If you know that you need a list type with good search performance (because your program must perform fast searches) pick a binary tree over an array. If you're not aware of any such requirement, *then* go for the most appropriate thing that will work. This still might not be the simplest – an array is a hard data structure to manage.

As you design each module, don't chase performance blindly – only spend effort on it when necessary. Understand the mandated performance requirements and at each stage justify how your choices will meet these requirements. When you know what level of performance is required, it's easier to design in appropriate efficiency. It also helps you to write explicit tests to prove that you do achieve these performance goals.

Some simple design choices that will increase efficiency and aid later optimisation are:

- Minimise your reliance on functions that might be implemented on remote machines, or that will access the network or a slow data storage system.
- Understand the target deployment and how the program is expected to be run, so you can design it to work well in these situations.
- Write *modular* code, so it's easy to speed up one section without having to rewrite other sections too.

Conclusion

High performance code is not as important as some people think. Although you sometimes do have to roll your sleeves up and tinker with code, optimisation is a task you should actively avoid. To do this, make sure you know the system's performance requirements before you start working on it. At each level of design, ensure you provide this quality of service. That way, optimisation will be unnecessary.

When you optimise, be very methodical and measured in your approach. Have a clear goal, and prove that each step is getting you closer to it. Be guided by solid data, not your hunches. As you write code, ensure that your designs are efficient, but don't compromise on quality. Worry about code-level performance only when it proves to be a problem.

That concludes this three part miniseries on optimisation. I hope you've found it useful.

Pete Goodliffe

The ACCU Conference 2004

A Report

by Pete Goodliffe *et al* <pete@cthree.org>

The lion and the calf shall lie down together but the calf won't get much sleep. *Woody Allen*

Another year, another ACCU conference. And this year's conference, like its predecessors, was an excellent event. In this write-up a number of attendees will tantalise you with brief reports of what went on. Let it serve as a memory-jog and piece of nostalgia for those who attended, and as an encouragement to come next year.

The Small Print

The 2004 ACCU conference took place between 14th-17th April, and this year returned to the centre of Oxford. A sense of grandeur was lent by the venue, the Randolph Hotel. Although it's comparatively hard to get into deepest darkest Oxford (the Park and Ride serves admirably), the central location made it much easier to fall out of the conference into the local night life – an essential part of conference attendance.

A Taste of the Atmosphere

All four days were packed full of high quality seminars, presentations, tutorials, workshops, "bird of a feather" sessions, and socialising. The ACCU conference is always a great place to meet like-minded individuals; people who like to program, love to learn, and want to sharpen their skills. Many ACCU members relished the opportunity to put faces to the names they see in periodicals and on the mailing lists. It's always surprising when your mental image of an intellectual giant is dashed against the harsh rocks of reality!

The geeks (let's face it, that's all of us) were particularly pleased by the Randolph's freshly installed wireless network, and many a happy hour was spent pulling out hair whilst trying to configure the darned thing to work.

Friday night's speakers banquet was a great success, if not an amusing occasion as delegates struggled to find enough spaces to eat! It was a good quality meal with an added twist – between each course delegates had to switch tables; a chance to meet several speakers, and the many other delegates. Who said technical conferences are dull?

A Taste of the Sessions

At the core of any ACCU conference are the keynotes and seminars. On each day there were five concurrent themed tracks, ranging from *Process* to *Python*, from *Open Source Software* to hardcore *C++*. The conference boasted more than 70 speakers, with many new speakers rising from within the ranks of ACCU, a refreshing trend – especially considering the high quality of their talks.

To provide a flavour of the content here are a selection of brief overviews. Unless otherwise specified, these have been supplied by `accu-general` regular Ric Parkin (`ric.parkin@ntlworld.com`):

Keynote: World Domination – Eric S Raymond

A talk about Open Source Software, from the writer of the highly influential "The Cathedral And The Bazaar". While evangelical, he made some interesting points about *domination relationships*, which is where your business depends on someone else, out of your control. In particular, software vendors often dominate their customers' future, and not always in a good way. In this light, OSS can be seen as a way of managing your risk – you have more control over your software. For software vendors, this implies a change from making software as a manufacturing company, to supporting software as a service company.

Paul Grenyer adds: The conference fun started in the first keynote, when Eric Raymond mistook Francis Glassborow for nothing more than "The Camera Guy", due to the fact he was filming the session, when he raised his hand to ask a question.

Keynote: C++/CLI – Herb Sutter

Herb is always an excellent speaker, and here he was going over some aspects of the proposed new C++ binding to the .Net CLI. There are two major design decisions that look to me to be vital, and to be done well:

- The first is the realisation that garbage collection and lifetimes are orthogonal – GC helps to reuse heap memory when needed, destruction is for tearing down objects when their lifetime is over. In C++/CLI they're changing the destructor to be equivalent to calling `IDisposable::Dispose`, and the compiler will Do The Right Thing with members. This is a major step forward, and will make C++ a better language to use than C# etc. where you have to call this stuff by hand.
- The second is to realise that GC references and C++ pointers are different, and no amount of fancy tricks will let you pretend otherwise. So a different syntax is used.

Herb also did a whirlwind tour of the context sensitive keywords used to declare managed types, and some differences between .Net Generics and templates. To finish, he announced that not only are Microsoft going to give away the VC7.1 command line tools free from today with no restrictions, but they're also going to do the same for the new version as soon as it comes out...

Paul Grenyer adds: Another highlight was when Herb Sutter admitted something that I am sure many of us feel deep down inside: "I Love C++". This, along with his announcement about the now freely downloadable Microsoft Visual C++ compiler (yes free!!!) and the marrying off of two couples of ACCU members as an example of the pairing between C++ and CLI, made the session very memorable.

[The compilers can be found at `http://msdn.microsoft.com/visualc/vctoolkit2003/` *– it does not include the GUI, but when I spoke to the chap from Microsoft, he did say you can drop the free versions over the previous binaries and everything should still work – Ed]*

Code Craft – Pete Goodliffe

Tony Barrett-Powell writes: Pete's session was about how to write better software and used the framework of a road sign to structure the session. His first assertion was that the main differentiator between good and bad programmers was not technical excellence but attitude. To illustrate this assertion the session quickly proceeded into a 5 minute practical to work in teams followed by Pete discussing some archetypal programmer attributes and finally to think about your attitude during the practical. He went on to note that software is generally developed in teams and that the individual attitudes effect the team. Finally he presented the ideal programmer in terms of attitudes and his suggestion that adoption of these attitudes would improve a programmer was well received by the audience.

Design Experiences in C++ – Mark Radford

A wide ranging talk about various designs found to be useful, and some thoughts on costs and risk. Many ideas came from managing the risk inherent in writing software – by catching problems early you reduce the risk and costs of finding them later.

An example was making value types rather than built ins. The idea is that you write a new type for every type your domain problem uses. For the small upfront cost (small if you write a decent templated type generator) you write a domain level language that gets the compiler to catch mistakes, and makes the code more expressive in its intent.

Other subjects included tying the units of a quantity into the type system, the different trade offs when choosing how to iterate over a collection, looking at genericity and virtual calls to express static versus dynamic variability.

Mark finished off by looking at using design to capture and manage complexity. Despite some recent languages, OO is not the only, or best paradigm to use. C++ is unusual in being multi-paradigm which allows it to model complexity well. It is:

- *Object Based* – allows working with "values", not runtime polymorphism. Offers abstraction and encapsulation
- *Object Oriented* – it provides runtime polymorphism as a first class feature.
- *Procedural* – it deals with control flow and decisions
- *Generic* – you can parameterise on types

These different paradigms are tools, to be used when designing.

Beyond the Gang of Four – Henney

Tony Barrett-Powell writes: Kevlin's session considered design patterns and how the Gang of Four's book design patterns has influenced, or impeded, the progress of design patterns. Using the framework of the 23 design patterns from the Gang of Four, Kevlin presented some inconsistencies and flaws in the design patterns. For some of the patterns Kevlin reconstructed them to show that in fact they hid further, more fundamental, patterns. The best example of this was his reworking of the Singleton, a much abused pattern and the source of a substantial secondary industry. The result of the reworking was a community of patterns with a genuine creation constraint and much clearer consequences. In summary Kevlin reiterated the what patterns were and that the Gang of Four is a historical subset of the patterns we need for effective design.

Remembering Code Experiment – Derek Jones

A quick session over lunch to get some real world statistics on how well programmers can remember, by performing a series of tasks.

Participants were shown three variables and their values. When you thought you had memorized them, you turned the page. There was a nested if statement of the form `if((a < e) && (u < a)) then if(e < u) then A else B;` Assuming the first if was true, you ticked which statement got executed, A or B. Now that you've been distracted, there was then a list of the previous variable names, plus a fourth one. You had to write down the values, or tick if you would have to look again, or if the variable was the extra one.

This was surprisingly difficult, especially as the same names and values came up regularly but in different permutations, so you couldn't mentally build a system model, as it would mislead you. The results are to be analysed and written up for C Vu.

Correcting STL Problems – Dietmar Kühl

`http://www.dietmar-kuehl.de/cxxrt/`

The STL is great, but there are problems. People write loops by hand instead of writing an algorithm, because functors are a pain. Libraries like `Boost.Lamda` can help but are not perfect. Also, passing pairs of iterators is usually too difficult and unnecessary – most of the time whole containers are used. So we add lots of algorithm overloads and use meta programming techniques to sort out possible ambiguities in the signatures. This gets you code like:

```
std::for_each( intcontainer, std::cout << _1
      << " squared is " << _1 * _1 << "\n" );
```

But there are worse problems. The STL concepts are too restrictive:

- proxy containers are not allowed, leading to the farce that `std::vector< bool >` isn't a Container!
- input iterator is limited – writing `copy_until` is impossible.
- limited algorithms.
- it's difficult to chain algorithms together – e.g. want to pass result of `equal_range` to `for_each`.

The solution is to refine some of the iterator concepts – it has a position cursor, and some properties that may be able to be read, written, or treated as an l-value (i.e. a reference taken to the underlying value):

```
T val = get(propertymap, positionkey);
put(propertymap, positionkey, value);
T& ref = at(propertymap, positionkey);
```

Beyond Methodology – Allan Kelly

A wide ranging conversation with the audience, ripping apart the idea of having a Methodology as a panacea. It tended too far into the evangelical; deriding what's wrong without proposing enough to replace it. Still, there were some thought provoking ideas, even if they weren't always well received. For example, Alan suggested that bug tracking systems are often used to *hide* bugs – if it's 'in the system' it's easy for no one to take responsibility for a bug. This is true, but the solution isn't always to not have a bug tracking system!

Design And Implementation Of Templates For Input – Francis Glassborow

A case study of how Francis developed some wrappers for reading and validating user input for a book on teaching C++. A few interesting gotchas, but mainly useful as a reminder that things shouldn't be difficult to use and we would all benefit from better wrappers at the right level.

What is C++ Missing – Alan Griffiths

A lunchtime Birds Of A Feather informal meeting, which wasn't so much about the language, but what tool support was missing, and why this was so. A lot came down to C++ being a hard language to parse, and people not being used to going and getting a library from elsewhere (the number of people who had independently written their own unit testing tools was amazing!)

Manipulating Streams – Dietmar Kühl

Dietmar is one of the experts on the C++ IOStreams library, especially the fiendish formatting framework. This was a very useful run through on what you can do and where you should do it. It covered writing your own facets for numerical formatting, and custom stream buffers.

Boost Iterator Library – Thomas Witt

`http://www.boost.org/libs/iterator/doc/index.html`

Another solution to some STL problems, this is how boost has written an iterator framework. Similar to Dietmar, they presented a set of new iterator concepts:

- *Traversal*, one of: Incrementable, Single Pass, Forward, Bidirectional, Random, where each builds on the preceding
- *Access*, one or more of the orthogonal: Readable, Writeable, Swappable, L-Value (Referenceable),
- *Interoperability*, e.g. convert and compare `iterator` to `const_iterator`

Thomas then presented two class generators: Iterator Facade that implements an iterator in terms of a small core interface, and an Iterator Adapter that takes another iterator and makes a new iterator that uses the other as part of its implementation. There were lots of useful examples.

C++ Class Design, a Check-List Approach – Alan Bellingham

To avoid forgetting things in the rush to write code, Alan presented a classification of types, and sets of rules for each type to avoid problems. The types are:

- *Object Types* – they have identity; their address 'defines' them; not much copying, which implies they don't go directly into containers; often polymorphic.
- *Value Types* – no identity, just value; address is irrelevant – all copies are equivalent; gets copied a lot; not polymorphic.
- *Functor Type* – no data, just functionality; `operator()` or single method; very simple
- *Pure Type* – no data, no functionality; role is to be different from other types; e.g. tags, `std::no_throw_t`; sometimes a hierarchy; exceptions sometimes are pure, as all that is important is the type.

More C++ Threading – Kevlin Henney

`http://www.two-sdg.demon.co.uk/curbralan/papers/accu/MoreC++Threading.pdf`

Kevlin is another excellent speaker, and here he raced through the two main models for writing multithreaded code – a C-style procedural API, and a classic OO inheritance model. He then ripped them apart to show how one was error prone, and the other was the wrong abstraction, and then introduced his own novel abstraction using generic techniques to separate concerns and reduce coupling, which ended up very, very neat.

Honey I Shrunk The Threads – Andrei Alexandrescu

Andrei was in a forgiving mood this conference – his talks were comparatively light on meta programming, so didn't make your ears bleed as much as usual. Here, he looked at using generic techniques to help with the difficulties of avoiding races and deadlocks. An overview of locking strategies brought forth several problems, and he presented some techniques for getting the compiler to help you out.

Paul Grenyer adds: As expected both Kevlin Henney and Andrei Alexandrescu gave two high quality and well attended sessions, on threading, exceptions and patterns.

Speakers Dinner

On Friday evening we had a semi-formal dinner which gave everyone a chance to talk to the various speakers for a while. Much technical and not so technical discussion ensued, which I found great fun and very informative. They continued in the bar, late into the night.

Paul Grenyer adds: The banquet gave further amusement, first with there being too many attendees and not enough tables and then with the fire alarm going off at about 1.30am. Those who had already gone to bed were forced to resurrect themselves to come and stand in the rain. I think this was a cunning plan by the hotel to raise revenue as most people opted for the bar rather than returning to bed.

[concluded at foot of next page]

Functional Programming in Python: An Introduction by Example

Thomas Guest <thomas.guest@ntlworld.com>

In a previous article [1] I described the development of a simple Python script to relocate source files, demonstrating how Python's built in objects and modules can make light work of such tasks. The final script bore a superficial resemblence to a C program: the code was structured into blocks, with a main routine calling subroutines to get the job done; objects were scoped; and control flow was handled by familiar constructs – loops, if statements and return statements.

Python also allows the creation of more C++ like scripts, complete with classes, encapsulation, inheritance, polymorphism, exceptions etc. In this article, however, I want to show how Python copes cleanly in a situation where C++ struggles, namely functional programming.

This article, then, builds on the previous article by introducing some of Python's functional programming capabilities by means of example. It also shows some more advanced uses of Python's regular expression module.

Statement of the Problem

Let's suppose we want to convert some articles from plain text format into web pages. Since these articles are about programming, they embed fragments of C/C++ source code, and since the presentation of source code matters, we want these fragments to have their syntax highlighted according to the following rules:

```
builtin types   -> blue
keywords        -> bold, blue
comments        -> italics, red
string literals -> green
```

I should confess at this point – as will soon become obvious – that I'm an html novice, and am aiming for the very simplest of markup to achieve the scheme presented above: this article is about Python, not web page development. Nonetheless, the finished script could be adapted to produce output suitable for more sophisticated web pages.

Towards a Solution

Full tokenization of C/C++ is beyond the scope of this script. Even hooking in to some third-party parser would be overkill. Fortunately, we can get the job done using a simple pattern matching algorithm.

In outline this algorithm:

- creates a pattern to match regions of interest in the code,
- processes the code, marking up these matches.

To get us started, the following script picks out some C++ keywords associated with control flow and emphasises them using bold text:

```
<python>
import re
html_data = re.sub(r'(if|else|for|do|while)',
                   r'\1', src_data)
</python>
```

Notice here the `\1`, which backreferences the first (and in this case, only) group matched by the input pattern. The `r` before the string literal often appears in Python regular expressions. It stands for `rawstring` and ensures that backslashes are not handled in any special way by Python – the string literals are passed directly on to the regular expression module.

Patching up the Problems

The simple script above is broken. The code fragment:

```
<cpp>
  cutlery = fork + knife; // do we need spoons?
</cpp>
```

would be marked up:

```
<html>
  cutlery = <b>for</b>k + kn<b>if</b>e;
                 // <b>do</b> we need spoons?
</html>
```

There are three unwanted substitutions! Clearly, we should only match keywords which are complete words, and keywords cease to be keywords when they're in a comment. Both problems can be fixed by using a more complex pattern:

```
<python>
def markup(match):
  """Return an html-marked-up version of the
     input match object"""
  html = ''
  if (match.group(1)):
    html = bold(match.group(1))
  elif (match.group(2)):
    html = italics(match.group(2))
  return html

cpp_re = re.compile(
   r'(\bif\b|\belse\b|\bfor\b|\bdo\b|\bwhile\b)'
   r'|’
   r'(//.*?$|/\*.*?\*/)',
   re.DOTALL |
   re.MULTILINE)

html_data = cpp_re.sub(markup, src_data)
</python>
```

[continued from previous page]

Writing Exception Safe Code – Andrei Alexandrescu

Andrei zooms into the difficulties in writing exception safe code, working on the problems with manual commit/roll-back schemes, an improvement using manual RAII handles, and finally diving into implementing his ScopeGuard helper.

Interviewing Skills Panel

A very interesting talk about how to approach hiring people, mainly focussing on taking an interview and how to find out if the candidate is suitable. Some good advice on preparation, and some reinforcement of some of the things I try and do.

All Heap, No Leaks – Paul Grenyer

A quick talk (well, it was the last session and we all wanted to go home!) on how Paul wrote a simple smart pointer, and enforced it's use to avoid mistakes.

A Personal View: "Revenge of the Camera Guy"

by Paul Grenyer <paul@paulgrenyer.co.uk>

I have been attending ACCU conferences since 2001 and this year has certainly been the best, both in terms of the breadth and quality of subjects and speakers, and in terms of the inevitably excessive socialising. Any fears (if there were any) that the conference would falter under its new organisational committee have been well and truly silenced. A superb job has been done by all. The hotel was of the highest standard, my only criticisms were that the WiFi network didn't work in my room and that shoes were a necessity for breakfast.

My favourite session was Jon Jagger's look at the Design and Evolution of C#. I found it very interesting to see how the language was shaped from the successes and failures of C, C++ and Java.

It's difficult to see how this year's conference could be topped, but I am sure that it will and I am very much looking forward to next year.

Acknowledgements

The conference wouldn't have been possible without the hard work of a dedicated team of people. Attendees owe a debt of gratitude to the conference committee, who organised an excellent programme, and to the conference organisers who ensured the whole thing ran smoothly.

Next Year?

Plans are already under way for next year's conference, and even now it's looking to be unmissable event. Stay tuned... For its low cost the ACCU conference continues to represent staggering value. If nothing else, it's a great chance to pretend to be deeply interested in technical issues when you really just want to meet ACCU regulars and insult them face-to-face.

Pete Goodliffe

Thanks to Chris Wakefield of CTA Direct for the logo at the top of this article.

Some notes on the regex pattern:

- `\b` matches (the empty string at) the beginning or end of a word,
- the flag parameter to `re.compile`, `re.DOTALL | re.MULTILINE`, enables us to match C-style multi-line comments (refer to the Python documentation [2] for details),
- asterisks – such as the ones which appear in C-style comments – are regex special characters, so they need to be escaped for a literal match,
- `.*?` is a non-greedy match of any character. Non-greedy means that the regex parser doesn't consume all the text it can to match a part of a pattern before backtracking to match the next part of the pattern. So, for example, `//.*?$` matches from the start of a C++ style-comment up to the next end-of-line character, but `//.*$` would match from this start point up to the final end-of-line in the data.

And some notes our more advanced use of the `re` module:

- parentheses have been used to capture matching text into groups,
- the first argument to `<code>re.sub</code>` is now a function which accepts a `MatchObject`. This function applies markup according to the group which has been matched.

Finally, it's probably worth saying something about the triple-quoted string literal which appears as the first line of the function. This is the function's documentation string, or `docstring`. These strings are, by convention, triple-quoted, since they often span more than one line. Refer to the Python documentation, and in particular "PEP: 257 Docstring Conventions", for details.

Creating Patterns Using List Comprehensions

I'm not entirely happy with the revised script above. I don't like the repetition of the regex special character, `\b`, and I don't like the way the C++ keywords have been obscured by the presence of these characters. This situation is only going to get worse when we extend the pattern to include all the other C++ keywords.

We could build the pattern up using an explicit loop and the built in `join` function.

```
<python>
keywords = ('if', 'else', 'for', 'do', 'while')
kw_patterns = []
for kw in keywords:
  kw_patterns.append(r'\b%s\b' % kw)
kw_pattern = '|'.join(kw_patterns)
</python>
```

Alternatively, we can use a list comprehension instead of the loop:

```
<python>
kw_pattern = '|'.join([r'\b%s\b' % kw
                       for kw in keywords])
</python>
```

This alternative is the Python idiom for building lists from sequences.

There are downsides to using this idiom. What if a C/C++ programmer tries to understand/modify this script? What if I don't use Python for a while (day to day, I use C++) – will I remember what the syntax means? (Think of all those ultra-succinct Perl programs, which default pretty much everything to apparently conjure their results from thin air.)

In this case, the idiom is well worth adopting. List comprehensions are too useful to be ignored.

Lambda Functions

There's another thing I'm not happy about: the separation of the patterns being matched and the markup applied to the matches. Recalling our original statement of the problem:

```
builtin types   -> blue
keywords        -> bold, blue
comments        -> italics, red
string literals -> green
```

I'd like the script to somehow embody this mapping.

The right arrows suggest using a dictionary to map from lexical elements to markup functions. Rather than use a dictionary, we'll choose a sequence of 2-tuples. This is because the elements of a dictionary are not ordered (perhaps dictionary was not the best term for Python to adopt for such a collection!) and we want to control the order in which pattern matches are tested. For example, `double` will appear as both a builtin type and as a keyword, and we can indicate we want it to match as a builtin by placing builtins before keywords in our markup rules.

So, we want something like:

```
<python>
def italics(str):
  return '<i>' + str + '</i>'
def bold(str):
  return '<b>' + str + '</b>'
def colour(str, col):
  return ('<font color="' + col + '">'
          + str + '</font>')
def boldBlue(str):
  return bold(colour(str, 'blue'))
# ...similarly define blue, italicsRed etc ...
markup_rules = (
  (builtin_pattern, blue),
  (kw_pattern, boldBlue),
  (comment_pattern, italicsRed),
  (string_pattern, green)
)
</python>
```

Here, the html markup helper functions `italics`, `colour`, `bold`, are good general purpose utility functions. The markup utilities – `blue`, `boldBlue`, `italicsRed`, `green` – do not merit existence as named functions: they are simply the result of composing our utilities and binding arguments to values. This is where Python's lambda functions can help:

```
<python>
markup_rules = (
  (comment_pattern,
   lambda s: colour(italics(s), 'red')),
  (builtin_pattern,
   lambda s: colour(s, 'blue')),
  (keyword_pattern,
   lambda s: colour(bold(s), 'blue')),
  (string_pattern,
   lambda s: colour(s, 'green'))
)
</python>
```

A lambda function is an anonymous function. Nonetheless, these anonymous functions can be passed around as parameters – we'll be passing them into our regex text substitution function, for example, and we'll use another lambda function to do this.

The Final Script

General Points

The final script appears at the end of this article. Most of its interesting points have already been discussed. Note, though:

- The lambda function used for pattern substitution, which binds its second argument to our markup rules enabling us to pass our markup rules into the markup function; this works around the fact that a function passed into `<code>re.sub</code>` must only accept a single match object argument.
- The use of the built in `filter` function, applied to the groups present in our match object. This function accepts a predicate function and a list, and returns a filtered list containing only those elements for which the predicate is true. If the predicate is `None`, the returned list has its zero or empty elements removed.
- The string literal match pattern, which introduces a negative look behind assertion, `(?<!\\)`. The pattern itself `".*?(?<!\\)"` will match an opening double quote followed by a non-greedy sequence of any characters followed by a closing double quote, provided that the closing double quote is not preceded by a backslash – is not escaped in C terminology. This is necessary in case we encounter a string literal of the form:

```
<cpp>
  char const * what_he_said = "He said \"Hi!\"";
</cpp>
```

- The `cookForHtml` function. The term "cook", in this context, comes from ([3]). Python's cgi module provides the required function directly.

Marking Up Python Code

I think it's clear that this script could be extended to handle other highlighting schemes or source data of other types. It's just a matter of defining the markup rules and patterns.

The thought of creating a pattern to match Python's various flavours of string literal is rather daunting. Fortunately, the language itself provides a `tokenize` module which does the job more easily and more accurately. The final script also provides a Python to html conversion function `python2html` based on the `generate_tokens` function available in this module.

This function embeds a sub-function, `popLine()`, which is used as the callable object required as a parameter to `generate_tokens`. This function can directly access the list of lines in the enclosing function's scope. An alternative would have been to create a functor object initialised with a reference to (or a copy of) the required list. In this case, I prefer to keep related code as close together as possible, and choose the sub-function.

Weaknesses

The script, although powerful, is far from perfect. It bundles data sets together with a number of unrelated utility functions to get the job done. Properly, it should be partitioned into modules: perhaps one for the C++ data tuples, one for the pattern creation utilities, one for the general purpose html markup utilities and one for our main functions. This partitioning is not hard to achieve, but will have to remain a subject for a future article.

To provide a more flexible markup tool, the functions `python2html` and `cpp2html` should provide a mechanism for clients to supply their own markup rules. The obvious solution here would be to allow these rules to be supplied as optional function parameters.

Finally, despite Python's support for unit testing, this script contains no unit tests.

Conclusions

Python's support for lambda functions and list comprehensions have allowed us to create a deceptively simple script.

The script by no means demonstrates all of Python's functional programming capabilities. A good reference book ([4], for example) describes these in more depth.

Thomas Guest

References

[1] Thomas Guest, "A Python Script to Relocate Source Trees", *C Vu 16.2*.
[2] `http://www.python.org`
[3] Jeffrey E. F. Freidl, *Mastering Regular Expressions*. The definitive reference on regular expressions, though note that it focuses mainly on Perl's regex support.
[4] David Beazley, *Python Essential Reference*.

Credits

Thanks again to Dan Tallis for his expert review of an earlier draft of this article.

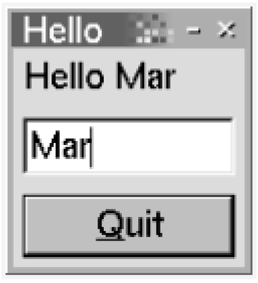
```
<python>
import re

# Reference ISO/IEC 14882, "The C++ Standard",
# Tables 3 & 4.
cpp_keywords = (
  'and',
  'and_eq',
  'asm',
  'auto',
  'bitand',
  'bitor',
  'bool',
  'break',
  'case',
  'catch',
  'char',
  'class',
  'compl',
  'const',
  'const_cast',
  'continue',
  'default',
  'delete',
  'do',
  'double',
  'dynamic_cast',
  'else',
  'enum',
  'explicit',
  'export',
  'extern',
  'false',
  'float',
  'for',
  'friend',
  'goto',
  'if',
  'inline',
  'int',
  'long',
  'mutable',
  'namespace',
  'new',
  'not',
  'not_eq',
  'operator',
  'or',
  'or_eq',
  'private',
  'protected',
  'public',
  'register',
  'reinterpret_cast',
  'return',
  'short',
  'sizeof',
  'static',
  'static_cast',
  'struct',
  'switch',
  'template',
  'this',
  'throw',
  'true',
  'try',
  'typedef',
  'typeid',
  'typename',
  'union',
  'using',
  'virtual',
  'void',
  'volatile',
  'wchar_t',
  'while',
  'xor',
  'xor_eq'
)

cpp_builtins = (
  'bool',
  'signed',
  'unsigned',
  'char',
  'short',
  'long',
  'float',
  'double',
  'wchar_t'
)

def preformat(str):
  """Return a preformatted version of the
     string."""
  return '<pre>' + str + '</pre>'
```

```
def italics(str):
  """Return an italicised version of the
     string."""
  return '<i>' + str + '</i>'

def bold(str):
  """Return a bold version of the
     string."""
  return '<b>' + str + '</b>'

def colour(str, colour):
  """Return a coloured version of the
     string."""
  return ('<font color=”' + colour + '“>'
          + str +
          '</font>')

def orPatterns(patterns):
  """Return a pattern which matches any one of
     the input patterns."""
  return '|'.join(patterns)

def cookForHtml(src):
  """Return the input data cooked for html."""
  import cgi
  return cgi.escape(src)

def markup(match, rules):
  """Markup the text matched by the input
     match object."""
  # Sanity check: the rules match the groups
  hits = filter(None, match.groups())
  assert(len(hits) == 1)
  assert(len(match.groups()) == len(rules))

  ix = match.lastindex # Last and only index,
                       # in fact.

  # Careful! - rules are indexed from 0 but
  # matchObject groups are indexed from 1.
  markup_fn = rules[ix - 1][1]
  return markup_fn(match.group(ix))

def cpp2html(cpp_src):
  """Return C++ source code marked up using
     html."""
  comment_pattern = orPatterns([
    r'//.*?$',   # C++ style comment
    r'/\*.*?\*/' # C style comment
  ])

  builtin_pattern = orPatterns([
    r'\b%s\b' % bt
    for bt in cpp_builtins
  ])

  keyword_pattern = orPatterns([
    r'\b%s\b' % kw
    for kw in cpp_keywords
  ])

  string_pattern = r'".*?(?<!\\)”'

  markup_rules = (
    (comment_pattern,
     lambda s: colour(italics(s), 'red')),
    # Need builtins before keywords – there’s
    # an overlap, since some keywords are
    # also designated as builtins.
    (builtin_pattern,
     lambda s: colour(s, 'blue')),
    (keyword_pattern,
     lambda s: colour(bold(s), 'blue')),
    (string_pattern,
     lambda s: colour(s, 'green'))
  )

  # Create a regex group for each
  # pattern in the markup rules, and
  # combine these groups into a single
  # pattern.
  cpp_pattern = orPatterns(
    ['(%s)' % p for p, f in markup_rules]
  )

  cpp_re = re.compile(
    cpp_pattern,
    re.DOTALL |
    re.MULTILINE # C-style comments can
                 # span multiple lines
  )

  cpp_src = cookForHtml(cpp_src)

  cpp_src = cpp_re.sub(
    lambda m: markup(m, markup_rules),
    cpp_src)

  return preformat(cpp_src)


def python2html(srcdata):
  """Return Python source code marked up
     using html."""
  import keyword
  import token
  import tokenize

  lines = srcdata.split('\n')
  def popLine():
    line = ''
    if len(lines) > 0:
      line = lines.pop(0) + '\n'
    return line

  marked_up = ''
  row, col = 0, 0 # Location of the end of
                  # the previous token

  for tok in tokenize.generate_tokens(popLine):
    # The tokenizer skips whitespace.
    # We must put it back.
    srow, scol = tok[2]
    if (srow > row):
      col = 0
    if (scol >= col):
      marked_up += ' ' * (scol - col)

    tok_name = token.tok_name[tok[0]]
    str = cookForHtml(tok[1])

    if (tok_name == 'STRING'):
      marked_up += colour(str, 'green')
    elif (tok_name == 'COMMENT'):
      marked_up += colour(italics(str), 'red')
    elif (tok_name == 'NAME' and
          keyword.iskeyword(tok[1])):
      marked_up += colour(bold(str), 'blue')
    else:
      marked_up += str

    row, col = tok[3]

  return preformat(marked_up)

</python>
```

Going GUI with Qt

Mark Summerfield

Welcome to the first installment in a series of articles exploring GUI programming with the Qt C++ toolkit. Applications developed with Qt can be run on all the main computing platforms in use today, including Windows 95–XP, Mac OS X, Linux, and most versions of Unix with X11, making C++/Qt applications almost as portable as standard C++ applications. Furthermore, Qt was designed as an object oriented library from its inception, which makes it very clean and pleasant to program with.



We're going to present and explain a very small "Hello" application that gives some of the flavour of Qt programming. The application will display a label, a line editor, and a Quit button, laid out vertically one above the other. The label's text will change whenever the user changes the text in the line editor, and the program will terminate when the user presses the Quit button.

We'll implement the application using three files: `main.cpp`, `window.h`, and `window.cpp`. Let's start with `main.cpp` which just contains a simple `main()` function:

```
#include <qapplication.h>
#include "window.h"

int main(int argc, char **argv) {
  QApplication app(argc, argv);

  Window *window = new Window;
  app.setMainWidget(window);
  window->show();

  return app.exec();
}
```

We begin by creating a `QApplication` object on the stack. Only one of these objects is created in a Qt application; it provides the GUI event loop and access to useful global information such as the desktop's size. Next we create a `Window` object – this is our custom main window, which we'll look at in a moment. For simple applications like this it is convenient to identify the main window as the application's "main widget"; this ensures that the application will automatically terminate when the main window is closed. Then we show the window on screen, and finally we start off the event loop with the `app.exec()` call. Most Qt applications have `main()` functions just as short and simple as the one shown here.

Now we're ready to look at the implementation of our `Window` subclass, starting with the header.

```
#ifndef WINDOW_H
#define WINDOW_H
#include <qdialog.h>

class QLabel;

class Window : public QDialog {
  Q_OBJECT

public:
  Window(QWidget *parent=0);

private slots:
  void updateName(const QString &name);

private:
  QLabel *label;
};

#endif
```

The first unusual thing to notice about the header is the `Q_OBJECT` macro. This signifies that the class has some Qt extensions to C++ which we'll discuss later. The `private slots` are private methods that can also be called by Qt's signals and slots mechanism. Like most GUI toolkits Qt provides low-level access to events such as mouse presses and releases, and so on. But very often we want to work with higher level concepts: we don't care *how* a button was pressed (whether by a mouse click, tabbed to then Spacebar, or Alt+underlined letter), we only care *that* it was pressed. Qt's signals and slots mechanism is a solution to this.

Qt widgets emit "signals" when state changes take place, for example when the text changes in a line edit, or when a new item is selected in a list box. These signals can be "connected" to slots (methods), so that when a signal is emitted any slots it is connected to are called. Widgets that are connected to one another are independent components since they don't need to know anything about each other.

We're now ready to look at `window.cpp`, starting with the `Window` constructor.

```
Window::Window(QWidget *parent)
             : QDialog(parent) {
  setCaption("Hello");

  label = new QLabel("Hello CVu!", this);
  QLineEdit *edit = new QLineEdit(this);
  QPushButton *button
             = new QPushButton("&Quit", this);

  QVBoxLayout *vbox
             = new QVBoxLayout(this, 5, 10);
  vbox->addWidget(label);
  vbox->addWidget(edit);
  vbox->addWidget(button);

  connect(edit,
          SIGNAL(textChanged(const QString&)),
          this,
          SLOT(updateName(const QString&)));
  connect(button, SIGNAL(clicked()), this,
          SLOT(accept()));

  resize(120, 90);
}
```

In Qt, every widget ("control" in Windows-speak) has a parent. If the parent is 0, the widget is a top-level window. A top-level window has any number of child widgets, nested to any extent. It is very rare to write destructors or to call `delete` on widgets in Qt, because when a top-level window is removed, it recursively deletes all its children automatically.

We set the widget's caption; for a top-level widget this is the window title. We then create the child widgets we need. In our case we are going to refer to the label in another method, so we've kept a private pointer to it. Since we want the widgets to appear vertically one above the other we create a vertical box layout and add each widget to it. The extra parameters (5, 10), are used to specify the layout's margins and spacing. Qt also provides a horizontal box layout and a grid layout; layouts and widgets can be arbitrarily nested. Qt also supports absolute positioning, but layouts are more convenient since they automatically adapt to the widgets they contain; for example labels will grow to accommodate longer text.

The Qt `connect()` method takes a `QObject` (all Qt widgets are `QObjects`), and associates one of its signals with a slot in another `QObject`. In our case we've said that whenever the line editor's text is changed our `updateName()` slot should be called. The `QDialog` base class provides an `accept()` slot which is called when the window is closed and accepted, and a `reject()` slot which is called

[concluded at foot of following page]

Reviews

Bookcase

Collated by Christopher Hill
`<accubooks@progsol.co.uk>`

A Note from Francis

As there are lots of reviews this time I will not say very much. However our new Editor wants to change the way we rate books by introducing a star rating. Please think about this. Personally I do not like star ratings because even one star looks OK. What do you think?

Before I hand over to this issue's reviewers there is an error from the last issue that needs correction. Somehow I left out the book details for the last review in the April Bookcase. The omission was then missed by the editing processes. At a casual glance one could think that I was adding a review to the book being reviewed immediately before my review in fact the book was:

Core CSS bye Keith Schengili-Roberts (0-13-009278-9), Prentice Hall, 818pp @ £39-99 (1.25)

Francis

The following bookshops actively support ACCU (the first three offer 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 me know so they can be added to the list

Computer Manuals (0121 706 6000)
`www.computer-manuals.co.uk`
Holborn Books Ltd (020 7831 0022)
`www.holbornbooks.co.uk`
Blackwell's Bookshop, Oxford (01865 792792)
`blackwells.extra@blackwell.co.uk`
Modern Book Company (020 7402 9176)
`books@mbc.sonnet.co.uk`

An asterisk against the publisher of a book in the book details indicates that Computer Manuals provided the book for review (not the publisher.) N.B. an asterisk after a price indicates that may be a small VAT element to add.

The mysterious number in parentheses that occurs after the price of most books shows the dollar pound conversion rate where known. I consider a rate of 1.48 or better as appropriate (in a context where the true rate hovers around 1.63). I consider any rate below 1.32 as being sufficiently poor to merit complaint to the publisher.

C & C++

Parallel and Distributed Programming Using C++ by Cameron & Tracey Hughes (0-13-101376-9), Addison Wesley, 692pp @ £43-99 (1.25)
reviewed by Nicola Musatti

When I first heard about this book I almost bought it sight unseen, on the basis of a very favourable review of a previous work by the same authors. In retrospect I am glad I did not.

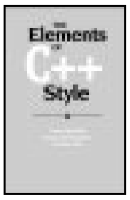
This book describes tools and techniques for parallel and distributed programming on Unix and Linux systems, using C++ as implementation language and focusing on standard API's and freely available libraries. These are the standard Unix process management API, Posix threads, PVM (Parallel Virtual Machine), MPI (Message Passing Interface) and CORBA. Note that, with the exception of CORBA, all these libraries only provide a C interface so most of the examples for this part of the book do not really contain C++ specific techniques.

Additional chapters are devoted to a theoretical introduction to parallel and distributed programming, error and exception handling, agent oriented and blackboard architectures and UML. Such a wide range of topics makes it impossible to cover any of them thoroughly; in my opinion it would have been better to concentrate on a more limited selection in favour of a deeper treatment. There is also an appendix that contains the full Posix Threads standard reference, which in my opinion could have been omitted.

The book's most evident defect is the very inaccurate, almost sloppy, way in which it is written; to the extent that I cannot help feeling that an early draft went to print by mistake. It is riddled with errors, from typos, to grammar ones, to technical ones. A few examples: very often a singular pronoun is used to refer to a plural subject; in almost all the code examples standard library entities are used without namespace qualification and without using declarations or directives; in the chapter on exceptions it is stated that C++ exception specifications are checked at compile time, while the language standard requires that they be enforced at runtime.

The authors write in short sentences, but this is not enough to make the text clear and in my opinion some important notions are not well explained; for instance I haven't understood what they think the difference is between parallel and distributed programming. The code examples are not good either; most of them are only program fragments that cannot be compiled and the few complete programs do not perform any interesting task, thus failing to support the reader in mastering the topics they should illustrate.

I found this book very disappointing, especially because nowadays publishers tend to concentrate on fashionable topics, thus publishing books that date quickly. I feel that an important opportunity has been badly missed. It is a pity because it is clear that a lot of effort went into this book, yet a lot more is required to turn it into a worthwhile one.



The Elements of C++ Style by Trevor Misfeldt et al. (0 521 89308 9), CUP, 182a6pp @ £9-99 (1.40)
reviewed by Francis Glassborow

This is a little book (literally) of 175 software development aphorisms. The expansion of each entry is well done and

when the window is closed and rejected, for example when the user clicks the window's X button, or presses Esc. Our second connection tells Qt to call the `accept()` slot when the Quit button is "clicked" (by whatever means). When the window is accepted (or rejected), because of the `setMainWidget()` call in `main()`, the application will terminate.

```
  void Window::updateName(const QString &name) {
    if(!name.isEmpty())
      label->setText("Hello " + name);
  }
```

The `updateName()` slot is trivial. And that completes our GUI application – in just 72 lines of code.

But what about `Q_OBJECT`, `public slots`, and other Qt keywords? The C preprocessor turns them all into pure C++ so the compiler never sees them. The machinery to handle Qt's extensions is generated by the moc (the Meta Object Compiler) which reads your source files and generates some additional source files to implement what you've used. Adding the rules for moc to a makefile isn't very difficult, but the majority of Qt users use Qt project files (`.pro` files). For this example we would put `main.cpp`, `window.h`, and `window.cpp` in a directory of their own; then we'd run `qmake -project` to generate the project file, then `qmake` to generate the makefile, then `make` as usual.

In later installments we'll explore Qt further, seeing how to create `QObject` subclasses with our own signals and slots, applications with menus and toolbars, and how to create custom widgets.

Mark Summerfield

Mark Summerfield is Trolltech's documentation manager, and co-author of C++ GUI Programming with Qt 3.

from time to time extra bonus information nestles either in the main text or in a footnote. For example, item 36 ‘Generate API Reference Documentation Directly from the Source Code’ has a footnote which gives `www.literateprogramming.com` as a useful source. Yes, it is a marvellous source, particularly if you follow such options as the ‘tools’ one.

The only warning I would give is that you should always read the whole of an entry because the first example in an entry is usually an example of what not to do.

There are a few experts who would find nothing new in this book, but most of the rest of us would benefit from it. Almost any time that a programmer chooses to ignore one of the aphorisms they need to justify their choice which makes it a good basis for communicating why a piece of source code either needs more comments or a different approach.

Buy a copy, keep it in your pocket to browse through while waiting in line for a meal, a bus etc. When you know it all, pass your copy on.

.NET & C#

.NET Common Language Runtime Unleashed by Kevin Burton (0672 32124 6), SAMS, 990pp @ £43-99 (1.36)

CIL Programming: Under the Hood of .NET by Jason Bock (1 59059 041 4), Apress, 340 hpp @ £35-50 (1.40)
reviewed by Frank Antonsen

There has been a lot of hype surrounding Microsoft’s .NET platform and the new C# language. As a consequence a number of authors and publishers have scrambled to cash in on this hype, and many books on these topics ought never to have been published. These are two of the better books.

Let me start with the .NET CLR book. The first two parts of this, amounting to some 180 pages, contain a good introduction to the Intermediate Language, or IL. An excellent introduction to the opcodes of the IL, the structure of the assembly and the available tools is given. The code examples are in both C# and VB.NET, and occasionally also in Jscript, J#, Perl or VC++. That they are short and not very informative examples of high-level languages I am willing to accept, as their main purpose is to illustrate the various aspects of IL. I consider that to be the best part of the book.

So far so good. Unfortunately, the remaining sections become increasingly flawed. The writing becomes very repetitive at times and could certainly have benefited from some editorial guidance. Similarly, I have a problem with the actual code examples. When an author decides not to show the entire code for a problem, he or she should think very carefully about what parts of the code do get shown. Boilerplate code and trivial expressions can certainly be left out, but far too often they are left in and the more important parts of the code are excluded. This problem is reduced, as the full source code is available for downloading. But I find it annoying when several pages are dedicated to developing an application and all that is used to illustrate it is numerous screen shots instead of showing the actual code. And when you lay down coding guidelines, you should at least follow them yourself.

On the other hand, one of the very good aspects of the book is the coverage of command line tools.

The second book (CIL Programming) complements the first quite nicely. The first part is an introduction to the structure of the assembly and to the IL opcodes. This part is not very well written, but the inside of the back cover contains a table of the basic opcodes. The fourth chapter contains a detailed walk-through of a complete project. The back of the book promises “Loaded with source code and real-world examples – no toy code”, which is a bit grand given that we are now at page 121, and we haven’t seen an executable program yet. The actual project itself is rather simple – it is a random number generator with a GUI test harness.

It is from here on the book becomes interesting. There is a chapter on debugging, covering both of the tools shipped with the SDK. We have a chapter on how various constructs in various .NET languages (VB.NET, C#, Oberon and Component Pascal) are translated into IL, and how the optimiser handles these. I haven’t seen such a chapter before, and I found it very interesting. And finally, there are chapter on emitting types (i.e. writing code which emits IL), dynamic proxies and appendixes on where .NET is heading and what languages are available (including those already mentioned, Perl, Python, Forth, and the functional languages Mondrian and F#). The examples are still relatively simple, but, as mentioned above, this is hardly a big problem.

The second half is, in my view, by far the best part of the book.

To conclude, both books show clear signs of having been written too quickly with little or no editorial guidance. This is a real pity, because if you could somehow merge the two books, with the proper editing, this could have been an excellent book. Still, if you happen to come across a cheap, second-hand copy you may want to take a closer look at it.

Recommended with reservations.

Mastering Visual Studio .NET by Ian Griffiths, Jon Flanders, & Chris Sells (0-596-00360-9), O’Reilly, 397pp @ £28-50 (1.40)
reviewed by Jon Steven White

As an experienced developer I’ve never found it necessary to refer to a book on a development tool. I’ve always managed to find my own way around packages such as Visual Studio, and by sharing knowledge with peers I feel that this gives me a good understanding of the product. I was therefore a little sceptical of just what this book could actually offer to its target audience, the experienced programmer.

As with most O’Reilly books, this one is well written and communicates useful and accurate guidance to the reader. It is up-to-date, covering the Visual Studio .NET 2003 development environment. The book flows sensibly through the fundamentals, introducing the reader to Projects and Solutions, as well as the various forms of file editing capabilities that VS.NET provides. I found an early chapter on Debugging to be very good, including some useful tips and excellent coverage of the VS.NET debugger features. Various aspects of Web, Database and Setup/Deployment projects are adequately covered, although I noticed that Web Services are neglected.

The more advance topics featured in the book include component integration, automation/macros and wizards. The chapter on automation, macros and add-ins was my favourite because it introduced me to the VS.NET Automation Object Model for the first time. I found this to be very interesting, and would like to refer back to this chapter when I eventually find time to utilise these features. The VSIP (Visual Studio Integration Program) is covered at the end of the book, but only has a brief mention. I don’t think this final topic is covered in enough detail for anyone to find it useful.

I have conflicting opinions of this book. I feel that it is good and does its job well, so if you really want a book on VS.NET then I would not hesitate in recommending this one. However, I’ve been developing with VS.NET throughout my two months company with this book, and have not found any need to refer to it at all. The basics of VS.NET Project, Solutions and Debugging are covered in many .NET related books at the moment, so chances are you’ll only be interested in the more advanced topics. A good book, but one that I wouldn’t rush out and pay £28.50 for.

Java

Inside Java 2 Platform Security 2ed by Li Gong, Gary Ellison & Mary Dageforde (0-201-78791-1), Addison-Wesley*, 356pp @ £34-99 (1.29)
reviewed by Christoph Ludwig

When Java first hit the market in the 1990s, the most prominent feature of its security architecture was the so called sandbox: A Java applet downloaded from the Internet could not access local files etc. unless the user explicitly granted permission to do so. A Java application loaded from the local disk, however, had unrestricted access to all resources.

Almost a decade later, this coarse approach has evolved into an architecture that provides much more fine-grained control at the cost of a considerably higher complexity. “Inside Java 2 Platform Security, Second Edition” promises a comprehensive overview of the Java security architecture and to teach its readers "everything they need to protect their information assets" [cover text]. While the book certainly does not leave many security related aspects of Java uncovered, it fails to live up to its latter promise.

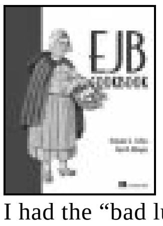
Two of the authors were at some point responsible for the design and implementation of the security components of the Java platform. They have a lot of inside knowledge about the design decisions to share and also point out compromises made and interfaces that are likely to be revised. The necessary background is provided by a lot of references to relevant standards, research papers, and textbooks.

But most of the time, the description stays too abstract. The text explains permission classes, security managers and so on in detail and it probably mentions all customization hooks there are – but it fails to demonstrate on a concrete example how to make use of all these interfaces. This may not be a severe shortcoming with respect to the coverage of the cryptography APIs (JCA/JCE) and the secure networking APIs (JSSE and Java GSS-API). These APIs can serve a purpose that almost every security conscious developer will have learned about when using the Internet, e.g. secure log-on, SSL, or signed email. Hence, most readers can easily imagine use cases.

But its not that easy to come up with a use case where you have a compelling reason to install, e.g., your own class loader. Therefore, I missed that the authors outline a scenario that justifies modification of such fundamental parts of the runtime environment and use it to exhibit the interaction of all parts of Java's security architecture as well as potential security holes that must be avoided.

The chapter on deploying the Java security architecture is bound to Sun's Java implementation, but it offers much more examples than the chapters before. If you have to install your own Cryptographic Service Provider, for example, then this chapter shows you which configuration files need to be modified. All of this can be found somewhere in Sun's Java SDK documentation on the web, of course, but it is quite convenient to have this information subsumed in a book.

This book offers an extensive in-depth reference of Java's security aspects for developers who are already familiar with the pitfalls of IT-security. But it is not a first choice if you are looking for a tutorial on how to develop secure Java applications.



EJB Cookbook by Benjamin G. Sullins & Mark B. Whipple (1930110944), Manning, 325pp @ £38-50 (1.12) reviewed by Christer Lofving

I had the "bad luck" to start work with Enterprise Java Beans in real world solutions at an early stage. Bad luck because maybe the techniques were mature, but the documentation and published material around it was not.

Consider the concept this delicately sized book (300 pages) is built upon. "Recipes" to solve both common and less common problems a J2EE (Java 2 Enterprise Edition) developer is doomed to encounter in her everyday work. Sooner or later. An appealing concept indeed, but some remarks must to be added.

First, the tips and hints here are of little value if you are just an enthusiastic apprentice in the "EJB-kitchen". You must have a solid J2EE-knowledge as a foundation, AND some experience as well. Second, it is a pain to read this book from start to end. Even trying to read a single chapter feels annoying. Truly, you are not meant to read it this way. But when you scan the wrapping or read synopsis you are seduced into that impression

Finally the code samples/solutions in this book are clean and easy to read, but testing the code reliability in an acceptably simple way is not easily done because of the required J2EE-framework. In the EJB world one just does not write a small "main" program and watch how things work out.

At the end there is a chapter dedicated to this testing problem. "A deployment and testing appetizer" which explains how you can use the open source products Apache and Ant as testing tools. I haven't tried this approach for myself.

All the "recipes" and the disposition of them in chapters is tastefully done. Also note that the book is aimed for the EJB developer (programmer) in first hand. Not for the system architect or the EJB deployment roles. Conclusion use of this book as a tutorial tool will probably fail, while it is a good buy for the more experienced J2EE-programmer.

Swing 2ed by Matthew Robinson & Pavel Vorobiev (1930110-88-X), Manning, 876pp @ £38-50 (1.30) reviewed by Christer Lofving

I have always found Swing one of the coolest parts of the Java programming language. The Swing concept has also grown since its first advent.

The mission this book is dedicated to navigate the reader through this vast API. I am impressed how the 800+ pages of this heavy book with a lot of code listings, still can remain such a pleasure to read. In fact, it is sometimes even hard to stop reading!

Part 1 covers the foundations of Swing and AWT. Fundamental concepts like Graphics and threads are clearly explained. As a reader you really get the big picture how Swing works together with the hosting operation system and its handling of the graphic environment.

Parts II and III "basics/advanced" topics seem aimed more for reference and handy help for everyday work solutions – "How to add a menu" for example. Even the samples are nice to read and easy to follow in this book.

The final part is about some specialized topics – like printing (yes, really a part of Swing). The concept of Windows drag and drop has a dedicated chapter of its own. The code samples are clean and well outlined. In spite of the implicit target group of "advanced" programmers, I think the text is also useful for the ambitious beginner on Java and/or Swing.

Learning by building a complete application (a JPEG-editor in this book) is certainly a both fun and rewarding to way to learn any programming! Recommended.

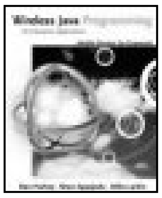
TY Java 2 in 21 Days, Professional Ref. edition 3ed by Rogers Cadenhead & Laura Lemay (0-672-32455-5), SAMS*, 853pp + CD @ £28-99 (1.72) reviewed by Paul F. Johnson

This is definitely not an easy book to review. On one hand, there are some well-presented parts with a clear line of thought; and on the other, parts that just aren't right. They may be technically correct, but they presume things such as having a network connection for parts (such as Day 6).

The book does come with all of the source code and the Java 2 SDK on a CD for Windows, Linux and MacOS, which is a bonus.

As with the C++ in 21 days series, the title really is a misnomer; it is not possible to cover anything more than a surface skim of an absolutely huge language in 21 days. I actually did try to learn Java in 21 days and found that at the end I could write code in Java, but I didn't have enough knowledge to plan a program and then sit at the desk and code, as I would with C, C++ (or even C#). Bits were missing.

If you are being taught Java as part of a lecture course, then the book will come in handy. But if you are sitting at your desk thinking, "Hmm, I think I'll learn Java" then you will need additional books to fill in the blanks.



Wireless Java Programming for Enterprise Applications by Dan Harkey et al. (0-471-21878-2), Wiley, 690pp @ £44-50 (1.32) reviewed by David Caabeiro

Ever felt overwhelmed by acronyms such as CDMA, GPRS, i-Mode, 3G?

I found this book to be a pretty good introduction to the Java wireless world, as it covers most technologies and platforms found in the field.

Unfortunately, its strength happens to be its weakness. Trying to span every single topic looks like an over ambitious goal. The result is a book that serves as a good introduction for the novice, but lacks of detailed information for the professional.

The book is organized in four parts. The first part is an overview of wireless foundations – aiming to the beginner – which gives a general idea to protocols, operating systems, cell phones, pagers and J2ME (Java 2 Micro Edition).

The second part is less general, and covers wireless device programming, such as the MIDP, VoiceXML, WML and JavaCard.

Even though the authors assume you have at least a basic knowledge of Java programming and markup languages, the level is not at all exhaustive.

At last, in part three the authors cover enterprise applications with J2EE, introducing a nice application example: a campus room finder. This application will serve as the grand-finale of the book. In part four, topics such as security, transcoding and personalization are discussed using this application.

All in all, a well-written book, differing from most introductory books by covering more than just J2ME.

Recommended.

Embedded Systems

Designing Embedded Communications Software by T. Sridhar (1-57820-125-X), CMP, 207pp @ £29-99 (1.33)

reviewed by Chris Hills

This book has a confusing title: Embedded communications but no mention of CAN, field bus, RS484 or 232 etc This is a book about network communication systems. If CISCO, PowerPC, Vxworks, SDH or sonnet means something to you then this book is aimed at your sort of embedded communications.

The book is rather short at about 170 pages and really only offers an overview. Though the book does cover all that is needed in 9 chapters. There are no circuit diagrams or source code in this book: it is at a much higher level. There is a good description of the OSI stack and diagrams of various processes e.g. TCP/IP implementation on Unix. There is a case study of a 2-layer switch. These are multi-board and multi-processor systems. If you are looking for more information on the nuts and bolts of how the Internet, national (digital) phone systems or WANs work then this is a good book.

Managers will find this book useful as it does explain how and why many choices are made and other pros and cons. For example the make verses buy arguments for some software modules. The relative merits for COTS RTOS over your own. Yes, in high-end communications software where 20 year MTBF is required, many still roll their own OS. Other topics discussed are things like HW redundancy schemes, hot and warm swap, fault tolerance etc.

One area I found interesting was the tools and debug. What was there was good but again it was rather short and superficial. Not really good enough if, as a manager, you are going to sign off in excess of 10,000 GB pounds (15K USD) on test gear for a project.

I think as an introduction it works but you will need other books to go any deeper, certainly for any implementation. I have free company specific and tool guides to networking with as much in as this book. However, it is well written and the author clearly understands the field. The author may want to follow up this book with some more (larger) volumes on specific areas that really can be used by engineers. Possibly a much enlarged second edition of the current book.

I think this would be useful for students and managers and possibly an introduction for engineers new to high-end communications systems.

8051 Microcontroller. An Applications Based Introduction by Calcut, Cowen & Parchizdeh (0-7506-5759-0) Newnes, 408pp

reviewed by Chris Hills

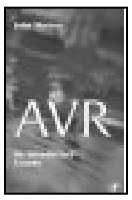
This is another book with a misleading title; it claims to be a book of 8051 applications. However, about a quarter of it is on the XA micro that, despite a lot of confusing marketing by Philips and the book's authors, is NOT the 16-bit version of the 8051. The 80251 is the 16-bit version of the 51. However, the 251 is not that widely used and the XA even less so. I can see no point in devoting a quarter of an 8051 book to the XA. About half the book simply contains information that is in the data sheets. This leaves about a third of the book. Not a good start.

The other unfortunate aspect of having the 51 and AX is there is a whole chapter describing how to use two different compiler suites. I am not sure why this chapter is present at all as the free evaluation CD (at least for the main 8051 compiler) has plenty of tutorials and screen cam demos, help files, application notes, example source code and virtually every 8051 datasheet there is. Unfortunately there is no mention of this in the text. There are also a couple of inaccuracies in the text regarding the Keil compiler suite.

There are a couple of simple complete projects (one of which is for the XA) but these are based on PCB layouts that have been included on paper only. The authors are not making these boards available as hardware or as electronic cad files. (There are many free/evaluation CAD packages about.) A Vero board layout would have been a lot better as it is unlikely students at this level will have access to PCB making equipment. The assembler source code in the book is also not available electronically.

It appears that this book has been written for a specific course without much thought for other readers. Some of the chapters go into detail such as giving blow-by-blow instructions for opening a particular dialogue in the IDE. Almost as lesson notes where time is short and you want a class to work through an example rather than a reader who will be experimenting and not need to complete a section in 40 minutes.

Overall this book may be of use to students on the course the lecturers teach but for other people (especially for those who have access to the Internet) that are many better options about. Talking of Internet: this book (and all others) should have had a web site where code and other files can be obtained. This is not book I would bother with.



AVR An Introductory Course by John Morton (0-7506-5635-2) Newnes, 238pp

reviewed by Chris Hills

This is exactly what it says an "Introductory course". If you want to understand micros and embedded work this is a good place for students, GCSE to first year degree and home users. Most engineers I would hope would not need this book just the AVR datasheets.

The book works in assembler and there are free assemblers and cheap dev kits out there for AVR. The most common AVR assembler is Atmel's own AVR Studio. It is nearly 7MB so you will need broadband to get it from `www.Atmel.com`!

There is also the, almost obligatory, section on hex, binary and number-bases. It also teaches good practice with flow charts and source code templates. The text not only tells you what to do but also explains why in clear and easy to understand terms. It does not however tell you how to use specific tools or assemblers. This is a good thing as tools (especially IDEs) change over time and this lets the reader choose their own tools, which are going to come with their own, help files and manuals.

On the down side there is no website or disk with the book. Therefore you will have to copy type the last 48 pages of assembler if you want to use the sixteen example programs. Or email the author and politely ask for an electronic copy of the source.

The projects cover use of most of the AVR peripherals and in assembler gives a good insight the how the parts work. The projects include pwm, motor-drive and RS232 comms to a PC as well as the usual switches and digital IO. There are exercises, some in the projects themselves with answers in the back of the book, which illustrate specific parts of the projects. I think this is a good idea. It gives additional help in the crucial areas.

There is a quite novel appendix "when all else fails, read this" that gives some very useful tips. Just 8 of them but they will solve a lot of the silly problems novices make.

If you want a hands on introductory course to micros or AVR and you are not an experienced embedded engineer this is the book. The AVR Studio contains a simulator so conceivable this is all you will need, but realistically you will also need to buy an AVR dev-kit. I tried it on a 17-year-old electronics student and he took to it. Lecturers using the AVR should look at this book. Recommended.

Smart Card Handbook 3rd English Edition by Rankle & Effing (0-470-85668-8) Wiley, 1096pp @ £95

reviewed by Chris Hills

What is there to say about this book? At 1086 Pages it is 40% larger than the second edition, which was 80% larger than the first. Of the first edition I said: "This is **The** smart card book against which all others will be measured". It is still the case. All smart card

Engineers should have access to this book. Buy it!

For those with the second edition: the third has new sections on PKCS #15, USIM, Tachosmart, Smart card terminals (MUSCLE, OCF, MKT, PC/SC) contactless data transmission. Revised information on the telecoms sector (GSM, UMTS, (U)SIM App toolkit, and decoding GSM files) also smart card security including attacks and counter measures. Apart from that the rest of the book has been updated and revised. You may want to get this edition anyway.

For those who don't have the second edition (including those with the first) **buy this edition**. It contains everything you need to know about smart cards as development engineer. However, it is too much for project managers, marketing etc there are other books that give an overview. This is the encyclopaedia.

It is expensive at 95 GBP (around 140 E) but compared to the standards you will need to buy it is good value for money. You might ask why you need this book if you have the standards. Simply put: this book is the commentary on the standards; that is standards plural. It fills in the gaps **between** the standards. I have seen an Engineer solve a problem with this book in 20 minutes that had left him stuck for two weeks with just the standards. I cannot promise that will always happen but it gives you the edge.

As I said it contains "everything" you need to know about smart cards. I would be hard pressed to find something that engineers would need that is not covered. No, it does not tell you how to hack cards, recharge satellite TV and telephone cards etc but it does have card OS information including Linux, windows for smart cards, Java and Multos.

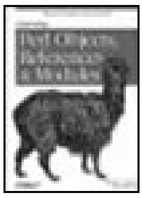
This book gives you all the building blocks for you to construct your own systems but it does NOT contain source code. For obvious reasons it cannot.

The useful part of the book is that it covers (almost) "everything". Making it an ideal reference for areas you are not directly working on. I found the section on card physical testing and production useful. Usually I grab this book before the standards for problem solving.

The style is factual and clear. It has been well translated from the original German.

This is **The** Smart Card Book, there is no other. Highly Recommended.

Other Languages



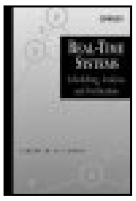
Learning Perl Objects, References, and Modules by Randal L. Schwartz with Tom Phoenix (0-596-00478-8), O'Reilly, 204pp @ £24-95 (1.40)
reviewed by Mick Brooks

This book picks up from where *Learning Perl* left off, and aims to be a "gentle introduction to advanced programming in Perl". Those familiar with the former title will know exactly what to expect: short chapters written in simple language with an exercise or two at the end, and a sprinkling of (sometimes interesting) details in the numerous footnotes. The exercises are written with a class based course in mind – each has a time for completion in minutes, and tends to require direct application of the material just covered. Suggested solutions to most problems are provided in an appendix.

The first half of the book concentrates on introducing the bits of the language that are needed to form the Perl object system. There is some really good stuff here, and my previous knowledge was made to rearrange itself on more than a couple of occasions. The second half introduces Perl objects and the facilities for re-use and distribution of code, tests and documentation.

I found it interesting that the use of objects is first motivated not by encapsulation (probably because Perl can't really do it), but by code re-use through inheritance. The system for writing classes is explained well. I finally understand why so many people dislike it, and why so much effort is being put into re-working it for Perl 6.

The book achieves its aims well, and I think it will prove a useful quick read to many. I enjoyed seeing plenty of idiomatic Perl alongside understandable explanations of it. Reference and re-read value will probably be low, but I do expect to be able to extract much more from Programming Perl having digested it.



Real-Time Systems Scheduling, Analysis & Verification by Albert Cheng (0 471 18406 3), Wiley, 524pp @ £62-50 (1.68)
reviewed by James Amor

This book describes itself as "an introductory text and a handy reference"; it fails to meet either of these claims. The level of content is inconsistent, interleaving basic concepts with expert descriptions; an approach that would see experienced engineers quickly losing interest, and new comers rapidly becoming overwhelmed. The book does not appear to have a target audience, in some places concepts are well explained, however others are simply skipped over leaving a gap in the readers knowledge that is called upon later in the text. My main criticism of this title is that it assumes the reader is fully conversant with formal notation. In an attempt to cater for readers without this knowledge an introductory tutorial is provided, however it falls drastically short of imparting the level of knowledge required to successfully appreciate the books content.

After hours of perseverance all I gained form this book was a headache! To ensure that this was not down to my interpretation I circulated the title amongst my colleagues all of whom were of the same opinion (quite an interesting conclusion considering we develop real time software for combat aircraft!).

I feel to recommend this title would be irresponsible. I do not feel the explanations given were sufficient to make this title suitable for beginners, whilst the irregular pitching of the content would quickly frustrate readers with an intermediate level of knowledge, and due to this title claiming to be an introductory text I would not expect it to be applicable to experienced readers.

Web Programming in Python by George K Thiruvathukal et al (0 13 041 065 9), Prentice Hall, 745pp @ £35-99 (1.25)
reviewed by Ratul Bhadury

The book's overall objective is to provide a foundation in Python, Linux, Apache web server, and MySQL for developing web-applications. It includes some sample, fully functional web applications developed with these technologies.

I was quite impressed with the chapters dealing with the Python language. Surprisingly the chapters were fairly detailed. Of course, the website tutorial is still an indispensable resource and must still go hand-in-hand with these chapters.

As a novice in Linux I did find the one chapter covering its basics rather lacking including only what the authors felt you would need to know while developing a web application. For example, when dealing with various Unix commands like cat, grep, kill etc., the book only mentions a few of the available switches. I would have preferred if the authors covered more of them and left it up to the readers to decide which ones would be of more practical use. The reader shouldn't expect this book to provide groundwork in Linux; and the authors don't claim it will.

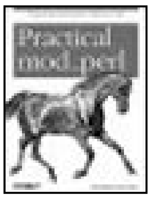
A chapter covers the basics of Internet working and HTTP. This includes a brief history of networking, network models, HTML, transport layer protocols, sockets and ports, HTTP etc. While it does cover topics like the OSI model, which are probably of little practical use, it is nevertheless a good, summarising read.

The chapter on Apache only covered what the authors felt was essential, and that too fairly tersely. I would have also preferred if they could have covered some GUI tools like Comanche. You will definitely need to hit some other books or websites to really understand the Apache topics covered here.

Of course you need to be fairly comfortable with relational databases and SQL in general to understand what's going in the next chapter on MySQL; for those who are, this chapter provides you with enough knowledge to get you started with MySQL. More advanced resources will obviously be required subsequently.

The next few chapters deal with the actual integration of what has been covered till then, in the form of Python CGI programming, various helper modules, and the Slither application framework for Python web programming. These will be of immense help if you already have a project in mind you wish to work on.

I was quite pleased with the book, and would recommend it to anyone wishing for an introduction to these technologies. If you are more experienced I would suggest you strengthen your weaker areas with a more specialised book.



Practical mod_perl by Stas Bekman and Eric Cholet (0-596-00227-0), O'Reilly, 893pp @ £35-50 (1.41)
reviewed by Richard Lee

This book tries to appeal to widest possible audience by including server administration with the mod_perl programming issues.

Prerequisites are an understanding of the Perl programming language and the Apache web server. Elements handled differently by mod_perl and most of the common configuration options are explained, but this is neither a book on how to program nor an administrator's guide.

To get the most out of this book, you will need to be running Apache 1.3 on Linux or similar Unix type operating system. There are bits devoted to Windows but they are riddled with inaccuracies. There is also a section on Apache 2 but this is stuck at the back almost as an appendix.

When discussing programming, the authors' knowledge becomes apparent. Common problems are concisely, if occasionally a little tersely, explained and solutions presented. I would go as far a saying chapter six is a must read for anyone writing or converting scripts to run on a mod_perl enabled server.

The administration side is the opposite. The authors have a habit of wandering and advice gets vague. Chapters on performance and benchmarking are useful but feel bloated.

By attempting to cover everything information is often presented in bits and pieces, with administration information mixed with programming. Useful nuggets of information get sandwiched and are easily missed. The lack of reference tables could also be a problem.

It is difficult to recommend this book as it stands as the quality of the programming advice is drowned out by noise around it.

Programming Web Services with Perl by RandyJ. Ray and Pavel Kulchenko (0-596-00206-8), O'Reilly, 470pp @ £28-50 (1.40)
reviewed by Tim Pushman

"Programming Web Services in Perl" contains no instructions for installing Perl under Windows, no description of the CPAN network, and no short introduction to programming in Perl. The book is exactly what the title describes, programming Web Services using the Perl programming language. The authors are both closely involved with the design and development of Web Services and the related Perl modules, so the level of detail is very high. In fact, it makes for quite a dense read, but works well as a reference book.

The book starts with a couple of chapters describing the overall design and purpose of Web Services and a quick description of the XML and HTTP protocols that the services are based on. The remaining chapters jump straight in to writing code for the different toolkits that are available in Perl, starting with XML-RPC, through SOAP and WSDL to UDDI, REST and more advanced topics. Each topic is covered very thoroughly with lots of sample code. The book ends with five appendices containing Perl module references and the complete sample code used in the book. The examples cover both client and server side applications.

Each set of chapters creates a small example application to demonstrate how to use the toolkits. This also allows an interesting comparison in how the code can be clearer with the higher-level toolkits and more powerful. Running the code brings mixed results, some of the URLs seem to have changed since the book was written, but, hey, it is a good way to learn what is happening.

Although the book is aimed at Perl programmers interested in Web Services, some of the protocols are described clearly enough that the book may also be useful for programmers in other languages, such as PHP or C/C++. If you are such a Perl programmer then this book is "Highly Recommended", indeed, almost essential.

Programming Web Services with Perl by Randy J. Ray and Pavel Kulchenko (0-596-00206-8), O'Reilly, 470pp @ £28-50 (1.40)
reviewed by David Caabeiro

This book describes the J2ME standard, aiming at the development of applications for the growing market of consumer devices such as mobile phones and PDAs. These standards were created by Sun and other top companies such as Nokia, Motorola, Samsung, Siemens and others, in an effort to homogenize the myriad of proprietary environments (even though the so promised WORA (Write Once, Run Anywhere) was not fully achieved).

Written by members of these standard bodies, this book is a good aid to the experienced developer who seeks for a reference of the new MIDP 2.0 and CLDC 1.1. The book contains a thorough description of the API available, with a good deal of code examples. One nice plus is the advice the authors give to (mis)using some of these APIs. Changes from the MIDP 1.0 are properly highlighted with an icon. At the end of the book, you can find both CLDC and MIDP APIs in almanac format, a nice detail for experienced developers.

Note that if you're looking for a book for beginners, then this may not be the best choice. This is a reference work, not a tutorial. The only objection I have with it is that other JSR approved standards, such as JSR82 (Bluetooth), JSR120 (Wireless API), etc. were briefly mentioned. Given the importance of these new technologies, more discussion would have been a nice plus.

Open Source Web Development with LAMP by James Lee & Brent Ware (0-201-77061-X), Addison-Wesley, 460pp @ £40-99 (1.10)
reviewed by Alan Lenton

The title of this book is something of a misnomer I discovered when it turned up. It is really all the varieties of Perl you can eat, with a few side dishes thrown in. Perl is obviously the first love of the authors and it shows.

That said, if you plan to use Perl as your scripting language to build an open source web site, then the book is probably worth considering. The avowed purpose of the book is to allow web developers to 'hit the ground running' if they come to open source from other areas.

The acronym LAMP stands for Linux, Apache, MySQL and either Perl, PHP or Python according to taste – although in this book the P stands unequivocally for Perl with a brief nod to PHP. The book does not cover content management systems (like Zope) at all, which is an unfortunate oversight. It also doesn't cover either XML or JavaScript, and I find the latter, at least, somewhat surprising.

The first part devotes a chapter each to the Linux operating system, the Apache web server, Perl and the MySQL relational database system. The chapters give you the basics, but you would need a proper manual to use them efficiently and safely.

The second part looks at static web site development and introduces the Website META Language (WML). I have to confess that I'd never heard of this before, and to be truthful, it seemed to me a rather complex way of building a static site. I suspect that it's a product of the fact that most open source web creation tools don't yet have sophistication of their Windows counterparts. Once they attain that level I suspect that languages like WML will fade away.

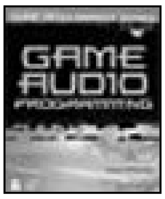
Part three deals with dynamic web sites, and starts, as you would expect, with a general introduction to the Common Gateway Interface (CGI), using, of course, Perl as the language to write CGI scripts. The remaining chapter of this part covers mod_perl, Apache's built in web server.

Part four covers what the authors call embedded programming (server-side programming) starting with Server Side Includes. Perl enthusiasm then switches into high gear and we have a chapter on another variety of Perl called Embperl, followed by another chapter on a fourth variety of Perl called Mason.

Finally, stuck on the end, a bit like a fifth wheel on a carriage, we have a chapter on PHP. I'm left with the feeling that after they delivered the book their editor said, "Hey, guys. What about the other 'P' scripting languages?"

As the authors themselves say while discussing the options for writing CGI scripts (p187), 'CGI scripts can be written in almost any language; we like Perl.'

I'd never have guessed!



Beginning Game Audio Programming by Mason McCuskey (1-59200-029-0), Premier Press, 352pp + CD @ £29-99 (1.33)
reviewed by Dáire Stockdale

Although I have mixed feelings about many of the books in the Premier Press 'Game Development Series', this is a good book. It is unfortunately written in the first person, colloquial style of the other books; however it is possible this appeals to the younger, perhaps less experienced, audience that these books hope to address. The title of the book explains that this is an introductory text, and it functions well in this role.

The book is composed of seventeen chapters, with each chapter providing just enough introduction and sample code to allow a user to accomplish something, get something working, and explore further as required. The book is to be complimented on its breadth: the author takes in a lot of ways of generating and playing sounds in games, including DirectX (of course), MIDI, MP3, Ogg Vorbis, CD, OpenAL and 'tracked' music. Of course, none of these are covered in great depth, which is fine, as the book is intended only as an introduction. The author appears to be both well informed and passionate about his subject, which makes the book pleasant to read. The book also serves as a useful introductory text to some areas of DirectX that can appear very dry and daunting, such as DirectX's dynamic soundtrack generation features, and the DirectPlay voice communication features. With a nod towards more advanced or ambitious readers, the author provides a nice explanation of discrete Fourier transforms, and their use in visualisation of music. He doesn't claim to be explaining anything to reference or academic depth, but instead provides useful links for further reading.

Although OpenAL and Ogg Vorbis do get a look-in, the book is primarily Windows oriented, and makes heavy use of DirectX for most of the chapters. It comes with a CD that provides working executables and Visual Studio projects to build them, which would be useful to any beginner. The CD also contains the DirectX, OpenAL, fmod and Ogg Vorbis SDKs. This is nice, as too many similar books neglect to provide on their CDs the really useful stuff, and just assume the reader will have access to a fast internet connection. The book also nicely avoids the habit this series of books has of spending several chapters just explaining Windows programming. In the introduction the author says the reader should already be familiar with C++ and Win32 programming. In all, this would be a good book for someone interested in adding game style sound features to an application.

Design

Large-Scale Software Architecture by Jeff Garland & Richard Anthony (0-470-84849-9), Wiley, 255pp @ £24-95 (1.60)
reviewed by Giles Moran

This book is another title in the growing field of software design books. The stated purpose of the book is to enable software architects to become more efficient in directing the course of large-scale projects. This is achieved by providing advice on which UML diagram should be used in which part of the project. One thing I noticed straight away was the comprehensive recommended reading section at the end of each chapter.

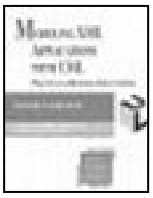
The book starts by defining software architecture and the role of a software architect. The next chapter details the role of a software architect in more detail and explains how this role may sit within a large management structure. A chapter on software architecture and the development process follows. These first three chapters offer a useful primer on the role of a software architect.

The book then moves onto using the UML to describe various aspects of a project. Each project phase of a project is split into a number of viewpoints. Each viewpoint employs a part of the UML to illustrate both software architecture and interactions within a project. Each view is detailed in a separate chapter and provides a detailed description of how the authors have used the UML to model distinct parts of a project cycle. A banking system is used to illustrate each phase of the project and corresponding viewpoint.

An initial chapter describes which UML diagrams are to be used and how the authors have developed them, to improve the clarity, by augmenting the diagrams with tabular data and some minor customisations of each diagram.

Different stages of the project then employ a model view to describe various aspects of the design. For example, component instance, state chart and activity diagrams are used in the section on component modelling.

This is a welcome addition to the growing literature on software design and the UML. It takes the UML, adapts it slightly and offers the reader the benefit of the authors' experience. This book will not teach you UML, but it will show you how effective it can be in planning a project. Recommended if you need to use the UML for a large project and are unsure of how to employ it to full effect.



Modeling XML Applications with UML by David Carlson (0 201 70915 5), Addison-Wesley, 333pp @ £26-99 (1.29)
reviewed by Emma Willis

This book is split into three sections: Foundations, XML Vocabularies and Deployment.

The first section is an overview of the processes involved in modelling your XML applications with UML. I found this section to be largely irrelevant, and for a section entitled 'Foundations', I found it overly complex and hard to get through. The author used UML diagrams and a description of these diagrams to represent the tasks involved in designing an XML vocabulary.

The second section brightened me up a bit. Although at times, I still felt a bit lost when it came to the XML example used, the chapters in this section successfully achieved two things; Firstly they gave me a thorough introduction to the concepts being presented e.g. mapping UML class diagrams to XML; linking XML elements and defining DTDs and Schemas; and also to a whole range of XML technologies such as XPath, XPointer, and RSS. Secondly, they really got into the nitty-gritty of each of these topics. This is where the complex example came in handy – it gave the author the opportunity to demonstrate each of the concepts in meaningful terms.

The final section on deployment covers the real-world business scenarios in which your newly learned skills will be deployed. The book concludes with a brilliant introduction to the business case for XML Web Services.

Overall I would recommend this book as an accompaniment to the host of web-based materials that there are on the topic.

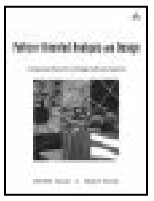
Learning UML by Sinan Si Alhir (0-596-00344-7), O'Reilly, 234pp @ £24-95 (1.40)
reviewed by Jon Steven White

This book is the second O'Reilly UML offering from Sinan Si Alhir, his first being 'UML in a Nutshell'. I found his initial book to be a very unstructured and unfriendly read, and was therefore hoping that things would be a lot better on this second attempt. I certainly wasn't disappointed because Learning UML is a vast improvement, and at last provides the O'Reilly fans amongst us with a UML book worthy of purchase.

Learning UML is aimed at a very wide audience encompassing anybody interested in learning and effectively applying the UML. The book breaks down the monstrous subject of UML into the usual chapter-per-diagram approach that is common across UML books. As expected, the fundamentals are discussed in the first part, and the author does a very good job here using written language in a metaphorical approach to UML. The second part of the book covers Structural Modelling, including Class, Object, Use-Case, Component and Deployment diagrams. The third part covers Behavioural Modelling, including Sequence, Collaboration, State and Activity diagrams. Each diagram chapter gives a good introduction, and gradually leads the reader from the basic concepts through to reasonably complex examples. The chapters conclude with a set of exercises that help the reader apply the concepts covered.

I found that the approach taken throughout this book works reasonably well, with each chapter covering the basics through simple examples before building further with increasingly complex examples. In my opinion, some of the examples covered were over-complex and inappropriate for the books scope. I found that the Author seems to 'over-do-it' in places, bogging the reader down too much.

Overall, although the book has weaknesses, it also provides a good platform for learning UML. If you read through the book in sequence and pursue the examples that are included, I think that you'll gain a great deal from this one. I would happily pay £27 for this relatively small book, simply because I know that it is going to be useful to me during my everyday working. I would certainly recommend this book to anybody, but would probably advise beginners to go for Martin Fowler's 'UML Distilled' first.



Pattern-Oriented Analysis and Design by Sherif M. Yacoub & Hany H. Ammar (0-201-77640-5), Addison-Wesley, 385pp @ £45-99 (1.30)
reviewed by Michael Pont

In "Pattern Oriented Analysis and Design", Yacoub and Ammar aim to describe a methodology – POAD – for assembling systems from design patterns. The book consists of description of their proposed methodology, a selection of case studies illustrating the use of this methodology, and a final section looking at ways in which their approach might be automated.

There are some interesting ideas in this book, including the proposal that patterns should be represented as components, in order to make it easier to support integration. Some of the current problems with patterns – such as the lack of complete "pattern repositories" or "pattern catalogues" – are also discussed.

Overall, however, I found this book unduly long, and difficult to read. Will POAD be of value to software developers? Time will tell, but – by the time I put this book down – I remained unconvinced by the case the authors had made for their approach.

Server Component Patterns by Markus Volter et al. (0 470 84319 5), Wiley, 462pp @ £29-95 (1.83)
reviewed by Michael Pont

This book describes a set of well-proven techniques for creating applications using components. Most of the examples use Enterprise JavaBeans (EJB), but the nature of the book means that many of the techniques described can also be used with other technologies, such as CORBA Components (CCM) or COM+.

The book is split into several sections. After an introduction, the core techniques are described: these techniques are generic, and small examples for EJB, CCM and COM+ are given throughout. Following this general section, a series of more detailed examples – using EJB – are presented. Finally, in what the authors call "a story", there is a case study describing the creation of an e-commerce applications using EJB.

Following the best pattern tradition, the techniques presented here have been reviewed extensively by "shepherding" and detailed discussion at the (2001) EuroPLoP patterns conference. This has helped to ensure that the material presented is of consistently high quality. One slightly unusual feature of this book is the splitting of the core technical material and the examples. In this case, I found the approach to be effective.

Test-Driven Development By Example by Kent Beck (0-321-14653-0) Addison-Wesley, 176pp @ £22-99 (1.30)
reviewed by Paul Grenyer

[Alternative Review]
Test-Driven Development (TTD) is split into three parts: Part I: The Money Example; Part II: The xUnit Example; and Part III: Patterns for Test-Driven Development.

I have some TTD experience and as I read Part I found it reassuring to see much of the process I use described in the book. It also taught me a few new things.

However, I had some of difficulty in following some of the code examples. This was mostly due to the code snippets being shown out of context and the lack of full code listings. This Part is split into a number of sections, and I think the book would be vastly improved if complete code examples were included at the end of each section.

Kent does state that the best way to appreciate the examples is to try them, but I feel this shouldn't be necessary and I use reading as a way of getting away from the screen. Despite the sparse code examples I managed to follow Part I, but only because I have used this sort of process before.

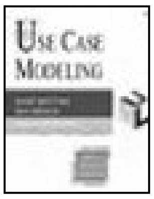
In Part II Kent develops a Python testing framework. During the introduction he states that he will "give a little commentary on Python". The commentary didn't materialise. Having done some Python in the past I just about followed it, but it took me a while to remember things such as data members in classes being declared the first time they are used. I also think that Python idioms like public data members and the need for 'self' to be passed to each member function should have been explained for those who haven't used a language like Python before.

Part III felt like a poorly thought out add on, trying to cover some of the material from the Gang of Four's Design Patterns book and Martin Fowlers Refactoring book in a much smaller space. The description of patterns doesn't seem to fit with the rest of the book. However, refactoring is a recurring theme throughout TDD and perhaps should have been given more space.

Throughout the book I found Kent's chatty style irritating, although perhaps it would appeal to those people who are as keen to enjoy the book as I was to extract and learn as much information as possible from it in the shortest period of time.

I would not recommend this book to someone I was trying to convince to use TDD, but I would (and have) recommend it to a colleague who has a basic understanding of the concept already.

This is the first programming book in a very long time that I haven't liked or enjoyed reading. However, this doesn't make it a bad book, just not a particularly good one.



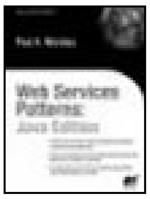
Use Case Modeling by Kurt Bittner and Ian Spence (0-201-70913-9), Addison Wesley, 347pp @ £26-99 (1.30)
reviewed by R.D. Hughes

If you want to really get into use cases, then this book is a pretty good starting point. There is a strong emphasis throughout the book on understanding what you should and should not be looking to get from use case modelling. This puts things into perspective, and for anyone who sees use case modelling as diagrams with stick-men, ovals and not a lot of useful information this book should be something of an eye-opener. The strength of use cases lies in the documentation and descriptions that should accompany use case diagrams rather than in the diagrams themselves.

A lot of the book is given over to considering what kinds of information should be captured in a use case. But in addition to this, considerable effort is made to address procedural and organisational aspects of use case modelling. This is important, because for use cases to be used effectively, it is vital that all concerned parties (I can't bring myself to use the S word…), many of whom are not technically oriented, understand what they are doing and what they should expect to gain from it.

The authors go out of their way to try and make the transition from theory to practice as easy as possible. Very often, the ideas and theories presented in books sound great, but prove very difficult to get to grips with on the ground. Two worked examples of use case descriptions are given in the appendices, which really help the reader to understand the kind of information they should be looking to capture.

This book is certainly a useful and thorough guide to use case modelling. That said it is probably best to be selective about which bits of the recommended approach are used. It isn't hard to see project teams getting bogged down in producing reams of documentation at the expense of generating real shared understanding of the project requirements.



Web Services Patterns: Java Edition by Paul B. Monday (1-59059-084-8), Apress, 332pp @ £32-00 (1.41)
reviewed by Matthew Strawbridge

When the review copy of this book arrived, I exercised remarkable self-constraint by writing down, before opening it, what I expected it to contain. Looking back on this list now, the book fulfilled a number of my expectations: there are plenty of UML diagrams; there is coverage of both EJB and simple two-tier architectures; and plenty of cross-referencing between patterns. Of course, some things I had hoped for did not materialise. There are no anti-patterns (such as the "magic button"), or methods for marshalling between Java member data and its XML representation. There is little discussion about the alternatives to SOAP, or of the future for Web Services where ‘bots’ could negotiate for services autonomously.

I also wrote down some things that I hoped weren’t in the book: long-winded introductions such as “what is Java?”, “what are Web Services?” and so on (this is, after all, a book aimed at professionals); coverage of general patterns that don’t apply to Web Services (there are other books for this); or tie-in to a single deployment platform. I am glad to say that the only one of these that was present was the tie-in (to Apache Axis), and I no longer think this is a bad thing, since it has enabled the author to provide real example code that people can easily deploy and experiment with.

In fact, one of the best things about this book is the example code (although I must confess that I haven’t tried to run it). The author uses a single case study of the P. T. Monday Coffee Company (based on the company owned by the author’s grandfather in the early 1900s) throughout the book. Each example extends the company’s capabilities, and the progress towards the final solution develops much the same as it will for someone creating analogous systems for their own company. All of the example code is configured as a project on `SourceForge.net`, which is an excellent idea and ensures that it will continue to be maintained and improved upon.

Unfortunately, the language used throughout the book really lets it down – to such an extent that if I had been browsing this title in a bookshop, I would probably have put it back on the shelf. The text isn’t always very clear (sometimes I needed to reread a sentence several times before I understood its intent), but this is made much worse by the author’s insistence on ‘management speak’. The worst example of this is gratuitous use of the word “leverage” – every Web Service must be leveraged, never used or applied or even utilized. The “what you will learn from using this book” section uses the dreaded word three times... and back on the shelf it goes.

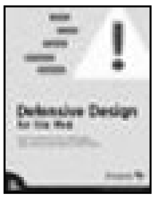
What the wording starts in terms of tripping up the reader, the layout finishes. It took me a while to realize that, even though the body text is left aligned, quite a few words are broken across lines. Perhaps this might seem minor, and it probably saved a tree or two, but these unexpected hyphens don’t help the beleaguered reader.

The factual content is pretty sound, and covers fifteen patterns: Service-Oriented Architecture, Architecture Adapter, Service Directory, Business Object, Business Object Collection, Business Process Composition, Asynchronous Business Process, Event Monitor, Observer, Publish/Subscribe, Physical Tiers, Faux Implementation, Service Factory, Data Transfer Object, and Partial Population.

There’s not much here that you wouldn’t find in any other book on Web Services, just organized slightly differently – the author is leveraging the popularity of patterns to facilitate mindshare with the readers!

If you want to fully understand all of the options for designing Web Services then this isn’t the best place to start. However, if you know you want to host SOAP-based Web Services running on Apache Axis, want a comprehensive suite of code to get you going, and won’t be put off by the style, then this book would be useful.

Recommended with reservations.



Defensive Design for the Web by Matthew Linderman with Jason Fried (0-7357-1410-X), New Riders*, 246pp @ £18-99 (1.32)
reviewed by Francis Glassborow

I find it sad that books such as this one are necessary. What I find even sadder is that the examples of bad practice so often come from the sites of major international companies.

The book consists of 40 guidelines for websites that interact with the surfer. These guidelines are broken down in to eight chapters that are book-ended by an opening chapter explaining defensive design and a closing one that gives you a nice little design test to apply to any site that you want to assess.

Every one of the forty guidelines is ‘obvious’ even if most are widely honoured in the breach. For example Guideline 1 is:

Give an error message that’s noticeable at a glance.

The author gives three examples of bad practice.

`www.qwestdex.com` which is a yellow pages like site covering a number of US States. `www.mothernature.com/topsellers` and `www.Hilton.com` were the other two sites. Of course, as you might expect, the problems detailed in the book are no longer ones with these sites (at least they had the sense to act when they were held up as an example). The last guideline (40) is

Show similar items that are available.

You have to know the context of that guideline for it to make complete sense. Chapter 9 is titled: *Out of Stocks and Unavailable Items*. A commercial organisation that does not offer alternatives when an item is out of stock isn’t exactly trying very hard. One of the examples of bad practice is ‘Barnes & Noble’. No doubt they will blame Amazon for all their commercial problems, but it is noticeable the Amazon has worked very hard to make its sites easy to use. I think they can still do better.

If you work for a company that promotes itself over the Internet, or if you have your own site where you hope for readers to interact with you, I think this book might be worth reading. Remember that if your employers are not promoting themselves well the chances of your having to find another job go up. Positive suggestions and examples are more helpful and this is where this book scores. The guidelines are a useful base from which to work, the examples of bad practice can increase the comfort factor for those who are being criticised and the examples of good practice show that it can be done.

Professional Development

Software Requirements 2ed by Karl E. Wiegers (0-7356-1879-8), Microsoft*, 516pp @ £28-99 (1.38)
reviewed by Chris Hills

I approached this book with interest, a Microsoft book on software specifications? I could hear the jokes already… This is a fascinating book. It takes specifications almost to the level of project management. Not only does it cover the requirements of the software but the test requirements as well. If you cannot write a test spec then the requirements are not complete. One chapter I like is where it discuses prototypes. The premise is customers never know what the want but they know what they don’t want! So do prototypes to show them what they asked for to find out what they really need. Most forms of engineering have prototypes software engineering is no exception. This came up in a paper on 61508 where it stated that 44% of failures in safety critical control applications came from errors in specifications.

On the subject of project management there are discussions on planning releases, which features go in which release etc. partitioning and future requirements you have to cater for now etc. Also version control is obviously touched on especially in the terms of controlling the requirements documentation. Controlling versions of software is one thing but making sure everyone is using the same version of the specifications for the software and the testing is another.

There are plenty of examples that show the author has done this for real… theoretically accountants don’t get involved in software specification but some of the discussions will be of use to engineers and project managers who have to argue with higher management, and for tools and time to do things. Of course risk analysis, risk, benefit, costs, priority are all covered in several areas along with comments on side effects.

There are views on things from several perspectives, which is useful for a project manager to understand where the others are coming from. One of the stakeholders is “maintenance programmers”. Something not often thought about when doing the specification.

Another acceptable section is an adequate selection of commonly used, state of the art words, and the problems they can cause if they are not suitably (and precisely) defined, as much as practicable under the right conditions… It is an acceptable attempt at clarification. English can be so ambiguous!

This is a cookbook of ideas and methods, tip and techniques rather than a single regime. Requirements are looked in respect of both UML Use Cases and structured methods and using in general the V model so it should be relevant for most users. Also there are sections on how to tighten up any process you currently have. There are symptoms with possible causes and remedies. This makes this book useful for virtually everyone who does not have a complete and smoothly running system.

The book is readable and easy to read. It is also easy to dip into, as the chapters are self-contained on a subject area so it is worth reading the preface for the guidance notes.

I would say that this book covers not only general software requirements for most projects or companies but also project management and high-end test strategies for small companies or projects where there are no other systems in place. It is pragmatic and real world. It is a book you can dip into for ideas and inspiration. I thoroughly recommend it.

Professional Software Development by Steve McConnell (0-321-19367-9), Addison Wesley, 241pp @ $29.99
reviewed by Chris Hills

I looked at this book with some enthusiasm as I had just written a piece on similar lines for an Embedded Systems Engineering magazine. (see the ESE columns on the ESE web site or my own `www.phaedsys.org`). This book looks at what makes a software engineer, personality types, the way we work, and the author's own "pilgrims progress" of how he moved from being a interested in programming though to wanting licensed software professionals. It is fascinating and raises many points.

Rather than a book on project management or code development, which it does mention in passing, this book looks at the philosophy, ethics and practicalities of being a software engineer. It looks at working practices, including stock options, overtime and free t-shirts. Engineering, art, bridge building and software are all linked with contrasts in their professions and discipline. All thought provoking stuff.

A large part of the book, perhaps half, looks at licensing professional engineer and sets out how this could be done in the US. It covers qualifications, experience, training and industry exams in fact virtually everything that BCS or IEE membership and Chartered Engineer requires. The author mentions every US body but chooses not to mention the IEE, BCS or the C.Eng in the book although he does know about them and looked at them prior to writing this book.

Whilst the book is well done and well thought out it is a totally US centric view that re-invents the wheel and ignores all the current systems in Europe, which makes it highly recommended for US readers but of little use for anyone else. In fact it could be misleading for European readers who may not realise that there is already a UK and European wide professional Engineer (and Professional Software Engineer) system that has been running for some decades. This is a pity as I fully support the reasoning and the thrust of the book. So I can only recommend this book for US readers. In fact I would suggest that it is essential for all US programmers or software Engineers. UK/European based programmers can get the same information by joining the IEE or BCS.

Specifying Software by R D Tennent (0 521 00401 2), CUP, 288pp @ £17-95 (1.45)
reviewed by Chris Hills

This book is very different to Software Requirements by Karl E. Wiegers I have just reviewed. That was a conventional software requirements book. This one is a little different. This one is closer to formal methods… Well there goes 80% of the audience! I picked up this book hoping for a conventional book on software specifications so I was a bit unhappy. Perhaps it should have something about formal methods in the title? Fortunately for those who by online the contents page does give some indication of the direction of this book.

Upon flicking through the book I was pleasantly surprised to see the examples of code were in C. This led me to hope that I might be able to apply formal methods to my C code specifications. As I work in embedded systems I come across a lot of safety and mission critical systems. Most books on formal methods are in Z, VDM etc or target things like Ada. So a book with formal methods and C would be welcome.

However when I looked at the first code example I found the include file "specdef.i" , apparently" .i" for include, contained a function with executable code in it and other include files! The short program contained a for loop where the statement following the control statement did not have braces around it… the excuse of saving page space is not acceptable in a book that is intending to teach rigour and formal methods.

On further examination I found the standard of the C throughout the book to be well below that which I would expect for any book looking to raise the quality of software engineering. In fact it is down right poor. I suspect that a mathematician rather than an engineer wrote this book. Single letter variables may make it look more like algebra but are not a good idea in software engineering.

Many of the things in the source code of this book are the some of the first things outlawed in coding standards. One of the "more efficient" code fragments uses multiple gotos! I then looked at the introduction to discover why this book had been written. Basically it is the notes for a module on formal methods for a university course. In some universities lecturers may recommend their own books as course books.

You may wonder why I have looked so long at the source code rather than the actual subject of the book… simply because it is no use having a good grasp of theoretical specification if you translate that into such poor quality code you make it difficult to debug and maintain. Not to mention test. You have to be able to test the code against the specification. Also there is some attempt to explain (teach?) the C as the book goes along. Actually it is C/C++, classes with `printf` statements in them! Strangely the references for C and C++ are the K&R and Stroustrup books, not a mention of the ISO standards in sight. So much for formal specifications!

The other problem is that there are exercises in the book, which one is expected to implement in C/C++ but only "selected hints" at the end of the book and no solutions.

There is a heavy reliance on assert. The problem is that formal methods tend to be used in safety critical areas. These are often embedded systems where there is often no assert function for obvious reasons. The other problem is that code must be shipped as tested and it is not usual to ship with assert code in it.

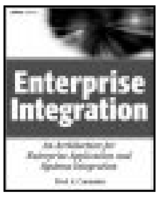
Having found so much I did not like about this book I must admit I did not work too hard to look at the more formal mathematical parts of the book.

This book may be usable for the students of the class where it is taught but I cannot find any reason to recommend it for anyone else. There are better books on formal methods, C, C++ and specifying software. This is a pity as a good book on more rigorous specification of C programs would useful. Not Recommended.

Guidelines for Application Integration by Microsoft (0-7356-1848-8), Microsoft Press*, 135pp @ £14-99 (1.33)
reviewed by Francis Glassborow

This is another book that is not for the programmer but if your work extends to general consultancy for medium sized companies (large ones are beyond the scope of the individual) you may find this book worth reading. Almost twenty years ago, in the first essay in 'Programming Pearls', Jon Bentley was making the point that we should understand the problem rather than simply answer the question we have been asked. So often we are asked 'How do I…?' usually the only correct response is 'Why do you want to do that?' People are very good at deciding that if only they knew how to … they could fix their problem. If only they would present the problem we could tell them how to fix it. Very often the effective fix is very different from what they wanted to do.

The above is not a digression, as software developers we have a responsibility to our clients. That responsibility extends to recognising circumstances where more software or replacement software is not what the client needs. This book addresses a particular and widespread problem; that of getting already existing applications to work together. Sometimes the solution can be as simple as changing a few work habits. At other times it can requires some special bespoke software to mediate between an application supplying data and one consuming it. If you are faced with a client that has several applications that need to work together this is a book that you should read. The rest of us have better and more profitable things to do with our time.



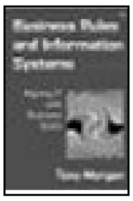
Enterprise Integration by Fred A Cummins (0 471 40010 6), Wiley, 467pp @ £33-50 (1.34)
reviewed by James Roberts

I did not enjoy this book. My main problem with it was that it seemed rather more interested in providing definitions of various parts of a system architecture, rather than describing their functions, and more importantly how they might fit together and why this would be a good thing. There was a complete absence of examples to illustrate points, which made the concepts in the book hard to grasp. The book as the feel of an exam crammer, which was not what I would have expected from the cover.

I also found the architecture described in the book rather prescriptive. It made recommendations (e.g. 'future business systems should be event-driven') without any discussion of what balancing choices might be necessary depending on system requirements or constraints.

The last source of irritation was the occasional redefinition of general terms to describe specifics. For example, the term 'loosely coupled' was defined as meaning asynchronous connections of business entities – whereas I would have considered this as a specific meaning of the phrase that has more general connotations (for example regarding build-time dependencies as well as run-time dependencies).

In summary I found this book somewhat disappointing. There may be people who would find it useful, but I would not recommend this for a general audience.



Business Rules and Information Systems by Tony Morgan (0 201 74391 4), Addison-Wesley, 348pp @ £30-99 (1.29)
reviewed by Lawrence Dack

A Business Rule is a compact piece of logic of logic capturing a small snippet of the way an organisation goes about its business: the rules by which a bank approves loans; or a stock market speculator picks stocks. The basic premise of this book is that a business rule-centred approach to information system development will dramatically improve both the development process and the final product.

The ultimate vision presented in this book is of a software development process that removes the need for software developers; instead a business owner would be able to create a structured, rule-based definition of the required information system, and automatic generators would produce the finished article.

That said, the bulk of the book concentrates on more attainable objectives. After a couple of introductory chapters, in which this vision is outlined and set in the context of the Zachman architectural framework, the book focuses exclusively on business rules. It moves progressively from rule discovery, capture, and description (in UML); through to the evaluation of rules and the reconciliation of contradictory rules and finally on to the clustering, management, and approaches to the implementation of rule sets. All of this is described in a lucid and engaging manner, well illustrated with a long-running example, and accompanied by pragmatic suggestions of how the ideas described might be implemented.

I found this book to be well written throughout and thought provoking in parts. However, it seems to assume, almost without discussion, that the reader shares the author's belief that information systems can be specified in sufficient richness using business rules alone. If, like me, you lack that assurance, then this book will probably not deserve a place on your favourite bookshelf, although it remains a good introduction to a school of thought that will certainly be of value in some parts of the development process.

Tools

High Integrity Software The Spark Approach to Safety & S by John Barnes (0-321-13616-0), Addison-Wesley, 430pp + CD @ £38-30 (no $)
reviewed by Michael Pont

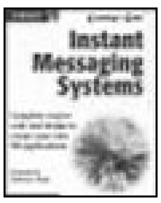
This is a revised version of a previous book "High Integrity Ada", by John Barnes. In the present book, Barnes describes how to use "Spark" (a programming language based on Ada) to write software for applications where predictable system performance is a key requirement.

The use of Spark – with appropriate tools – allows a high degree of static analysis to be carried out during program development, with the aim of obtaining what Barnes calls "correctness by construction". For example, the programmer can specify pre- and post-conditions for functions, in a (comparatively) simple way.

There are many interesting ideas here. However, around a third of the book is a cut-down "user guide" for a set of tools produced by Praxis Critical Systems. In addition, the examples presented throughout the text are very small, and even the (so called) case studies are rather too brief to be very convincing. I was also struck by the very limited coverage of issues relating to I/O (crucial in most safety-critical systems), and the fact that interrupt handling (another important issue) is not even listed in the index.

Overall, this book may be of greatest value to users of those who have purchased Spark tools and want an introduction to the language and its use.

Internet



Instant Messaging Systems by Dreamtech Software team (0 7645 4953 7), Wiley, 684pp + CD @ £37-50 (1.33)
reviewed by Alan Lenton

This was a very difficult book to review. I have been sitting on it for longer than I intended, wondering whether in fact I was qualified to review it. For a start, the title, on the basis of which I offered to review it, is a complete misnomer. It's not about instant messaging systems. It's about one such system – Jabber. And it's about developing one Jabber application twice – once in Java and once in C#.

Since I am not a Java or C# programmer, I nearly handed the review copy back. However, I do know something about network programming, and about Jabber, having implemented a version of the server some time back, so I thought it was worth persevering. Well for a while, anyway. The problem is that the bulk of the book is taken up with reams of listing of the source code for the application the authors are developing. All of it, as far as I can make out, the boring bits as well as the interesting and important bits.

Come on guys, you included the source code on the CD at the back of the book. Presenting all the source code in this way means that the material that readers really need to read and understand is completely lost.

I cannot really comment on the quality of the code, but I can comment on a couple of things that definitely didn't impress me. First of all the authors hard coded the TCP/IP port numbers into their code. This is a fundamental no-no for network programming – you use the `getservbyname()` function which allows the user to define which ports they wish to use. This is network programming 101 – it makes me wonder what the quality of the rest of the code is like.

The other thing I would question is that instead of allowing the users to connect directly to the Jabber Server, the authors have an intervening web interface. Now, the web is one of the most clumsy ways of performing instant messaging that I know – I'd even prefer raw telnet to the agony of a web based instant messaging client. Apart from anything else it means that your intermediate server has to generate all the Jabber XML itself.

This way of doing things just doesn't make sense. One of the really nice things about Jabber is that it has excellent libraries for creating custom clients – and they are available in all the popular languages. Any competent programmer could knock up a nice custom client in a day – all the grunt work is done for you by the libraries. Why re-invent the wheel?

I'm sorry; I really can't recommend this book. If you are considering using a Jabber application, then I suggest you look at *Programming Jabber* by D.J. Adams, which will give you a much better understanding of how to use Jabber.

Maximum Apache Security 4ed by Anonymous (0-672-32459-8), SAMS, 945pp + CD @ £36-50 (1.37)
reviewed by Ian Bruntlett

The book lists numerous exploits that the average web master should be aware of. And it lists many online white papers that explain things in greater detail. It also discusses the server tools that ship as part of Apache. Apache's logging facilities are described.

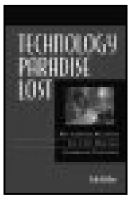
The internals of Apache 2.0 are discussed in detail. References to important sections of the Apache documentation are given.

Good practice to follow when writing server side applications is discussed. It recommends that you choose one language, learn it well and stay current on security issues. This may be a problem with some web masters who, like me, probably already know C/C++ and have dabbled with Perl. The book goes on to describe numerous problems – and interesting security and testing tools.

The book looks "under the hood" of Apache, identifying key C source files. It describes security as implemented by Netscape's SSL. It explains the use of firewalls with Apache. It also explains how, in certain circumstances, Apache can be used as a proxy server (for FTP, HTTP, HTTPS, SOCKS). The final chapter discusses the way to write expansion modules for Apache. This is very thorough but I feel the book should have also had an explanation of CGI and perhaps a better explanation of sockets programming and a discussion about ports.

Recommended despite some gaps (sockets, ports) and its age (2 years old).

Other



Technology Paradise Lost by Erik Keller (1-932394-13-3), Manning, 243pp @ $24.95
reviewed by Francis Glassborow

This book is aimed at managers and others trying to understand the shift in importance that IT has for commerce. By 2000 IT was the single largest expense for many companies (indeed it is claimed that it absorbed 50% of all corporate capital expenditure that year.)

Times have changed and much more happened than simply the inevitable dot-com collapse.

This book will add nothing to the peace of mind of the average software developer. Worse, Eric Keller predicts that the halcyon days of IT in the late 90's will never return. I think he is likely right. Yes we will continue to develop better tools, write larger and hopefully better software, have ever faster and more capacious computers but many companies realise that they need to focus on their core business rather than spending money chasing a technical IT fixes.

For the record I think the next great technical growth areas are far away from IT. I would speculate biology, genetic engineering and nanotechnology will increasingly feature in our futures.

Anyway, I would not recommend that you buy your own copy but if you want to better understand the employment environment for computer technologists (network managers, web-designers, software developers etc.) this book will help by providing a thoughtful view from an expert on IT in the commercial world.

Degunking Windows by Joli Ballew & Jeff Duntemann (1-932111-84-0), Paraglyph Press*, 310pp @ £18-99 (1.32)
reviewed by Francis Glassborow

The second author of this book has certainly been around for a long time (as long as I can remember). I am not familiar with Joli Ballew's other work but she has a considerable number of books to her name. The two authors complement each other and the resulting book is helpful and largely well written for the target readership (identified by the publishers as 'Novice to Intermediate').

If you want some help in keeping your Windows XP based machine working reasonably well this is a fair book for the non-expert. However one piece of advice that I would give most forcefully to anyone who wants to start cleaning his or her machine, tweaking software and editing the registry is to have a complete back-up before you do anything else. Hard drives are cheap enough these days so that there is no excuse for not cloning the drive before you start messing. If you do not do that I can promise you that even if you are an expert you will regret it.

Secrets & Lies (revised) by Bruce Schneier (0-471-25311-1), Wiley, 414pp @ £11-99 (1.63)
reviewed by Francis Glassborow

I reviewed the original edition of this book so I will start this review by quoting the opening of the Introduction to this updated version.

It's been over three years since the first edition of *Secrets and Lies* was published. Reading through it again after all this time, the most amazing thing is how little things have changed. Today, two years after 9/11 and in the middle of the worst spate of computer worms and viruses the world has ever seen, the book is just as relevant as it was when I wrote it.

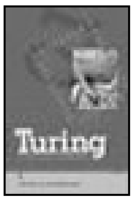
The attackers and attacks are the same. The targets and the risks are the same. The security tools to defend ourseleves are the same, and they're just as ineffective as they were three years ago. If anything, the problems have gotten worse. …

I can understand the author's frustration. As a world authority on cryptography he feels a sense of guilt and frustration that the promise of cryptography to address security problems has failed and no one seems to be taking any notice. If you did not read the first edition, please read this one and then try to ram the message home to your managers, your employers and your political representatives.

The real issues have little to do with whether you can or cannot read my email, and the real issues are not those of so called cyber-terrorism.

But the real issues can impact on every one of our lives. Only when we all understand why using high technology will not solve our problems will we spend time addressing the fundamentals. There is no value in having highly sophisticated identification technology (such as iris scans) if simple forgery can change the information to match the identity thief.

Please do not just sit back and say 'someone ought to do something.' You are someone, do something. The starting point is to stop being either complacent or fatalistic and make yourself better informed. Read this book.



Turing (A Novel about Computation) by Christos H. Papadimitriou (0-262-16218-0), MIT, 284pp @ £16-95 (1.47)
reviewed by Francis Glassborow

Novels and shorter fiction can be very effective tools for teaching. They can present technical information in a more informal and less threatening fashion (though not a novel, Proofs and Refutations: The Logic of Mathematical Discovery by Imre Lakatos – I am puzzled as to why Amazon only lists him as an editor when the work is substantially his Doctoral Thesis – is an excellent example of using a cast of fictional characters to explore and better understand of an aspect of mathematical philosophy.)

The trouble with fiction is that too often the background contains too many flaws to be relied upon. That is generally a pity because we assume otherwise. We know something is just a story but an author who places Bruges in, for example, France is doing a lot of unintended harm. Those that know better are irritated and those that do not subconsciously learn something that is untrue.

Now in this case we have a novel that is about computation. It is many other things as well including a good spicing of romance and a smidgeon of raw sex. You might wonder about the technical accuracy of the book (as regards computation rather than male/female relationships). This is one of those rare but very pleasing cases where a world class expert on mathematics and computing has written a book that is not just a fun read but which embodies a good deal of technical knowledge. If you are an expert on computation, read the book for fun, if you're not read it for fun and to learn a little too.

We could do with more books like this one that address our intellects at so many different levels simultaneously.