Contents

Reports & Opinions

Reports

Dialogue

Features

Python

Reviews

Copy Dates

C Vu 15.2: February 21st **(Note early copy date, to enable publication of the April issue before the Spring Conference)**

C Vu 15.3: May 1st

Contact Information:

Editorial: James Dennett
1261A 17th Avenue
San Francisco
CA 94112, USA
editor@accu.org

Advertising: Pete Goodliffe
Chris Lowe
ads@accu.org

Treasurer: Bryan Scattergood
19 Bayford Place
Cambridge, CB4 2UF
01223 475468 (home)
01223 692445 (work)
treasurer@accu.org

ACCU Chair: Alan Griffiths
alan@octopull.demon.co.uk
chair@accu.org

Secretary: Alan Bellingham
020 8998 6964
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): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21
There are 6 journals of each type 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

Professionalism For The Masses

Spring Conference

Firstly, a few words about the forthcoming ACCU Spring Conference. By now you should have seen the array of speakers. It truly is unmatched in its range of excellent speakers, at any price, and the price is one of the lowest you will find. It is also a great way to socialise with other software developers, and to drop the arguments about whether emacs or vi is the One True Editor in favour of more important discussions (such as which restaurant is best placed for handy access to a pub). More will follow from the conference organiser on why you will want to be there.

Student Code Critique

The quality (and number) of entries for this regular spot varies widely. This issue I'm pleased to see several really valuable responses; only one can win (I expect the suspense is killing you) but you will find it worthwhile to read all three. The variety they show gives some idea of how much scope there is for responses in this section. Think that these "easy" problems are below you? Maybe it is a chance to show how well you can pass on your wisdom to others.

How Professional is Professional?

So, you want to be live up to the ideals of "Professionalism". Sounds like a good thing. The next question is: what does that entail? (Apart from joining ACCU, obviously!)

Well, reading books is a good way to learn, and almost certainly necessary. Writing code, too, and reading code written by others. What else?

Those who want to keep up to date should look into reading one or more of the newsgroups. For C, try `comp.lang.c`. For C++, `comp.lang.c++.moderated` has heavy traffic in spite of (or maybe thanks to) the sterling efforts of its moderator, but a lot of interesting material is posted there, and it's used by many of the leading names in the C++ world. You might also want to read some platform-specific groups, as the "`lang`" groups focus fairly strictly on portable code. If the volume of the newsgroups is too much (or even if it isn't) then the ACCU mailing lists are a valuable resource and tend to have a better signal/noise ratio than unmoderated newsgroups. And of course there are the ACCU "Mentored Developers" projects where you can learn alongside others, either as a student or a mentor.

Naturally a professional will want to be armed with more than one programming language; it would hardly be admirable to choose a tool because it was the only one you had, better to choose it based on what is best for the job. So, you'll want to learn about other programming languages, and practise using them. Some reasonable books recommend you pick up one new language a year to broaden your thinking.

Now, programming in isolation isn't very common in the modern world. Programmers are expected to do design, analysis, write documentation, test code, estimate tasks, and sometimes communicate with customers, provide technical support, and coordinate the efforts of our peers. We're also asked to recommend technologies and tools, and to assess new processes. The list of what an ideal software engineer would know is virtually endless.

The aim of many ACCU members (and others) is to be the best they can. This is commendable - but is all of this necessary in order to be "professional"? I say that, while these are noble ambitions, it is not necessary to take things to these lengths in order to feel that one is professional. If you want to be among the best, you can indeed count on a need for an enormous commitment in time and money to keep learning. Then you might start to look disapprovingly on those who do not go to such lengths.

But that wouldn't be professional. Such attitudes can be found on the newsgroups: people who write that "if you don't understand that *<non-trivial item>* then you should not be attempting to do *<what the person needs to do>*." Yes, it's frustrating to see for the hundredth time someone asking a basic question that a FAQ or a websearch or a basic textbook would have answered, but remember that we all had to start somewhere. Aside: the first book I read on C++ was by one Herb Schildt. I wouldn't recommend it now, but I might have been demoralized at the time if I had mentioned it only to be told that I needed to unlearn it all and start over. We should try to ensure that the places where people can improve their skills are places which encourage them to do so. Preaching to the converted is easy, but there is a worrying gap in skill levels between the best and the majority. Our priority should not be to progress the start of the art so much as to improve the state of practice. The quality of output from our industry isn't good enough, and it's not going to be good enough if we simply establish cliques of high priests who view most working programmers as being part of the problem. Next time you're thinking of criticising someone for making an error that you consider appalling, instead of allowing it to reduce your respect for them, try politely to work out what it would take for them to be interested in how to improve on that error. Maybe they'll return the favour later!

With thanks to everyone who has ever helped me to fix a problem in my code...

James Dennett

From the Chair

Alan Griffiths `<chair@accu.org>`

How many of you have uttered the cry "why is there no documentation?" and how many of those don't look at the documentation that there is because "the documentation is useless" and/or "it is easier to ask someone"?

Just because the quantity of documentation is tangible there seems to be a strong tendency to focus on creating it. But what is too often missed is that documentation isn't the point. Communication is the point. Software development is largely about communication.

These thoughts have been sparked off by a project that I've been hearing of recently, where a firm of consultants is contracting out some development work for their client. When the contractor proposed a project plan it included an initial phase to "define the project scope" - the consultant's reaction: "we've already done that - here is the scope document". They are missing the point: their document, whether good or bad, can only support a dialogue between the contractor and the client - not replace it.

What has this got to do with ACCU?

Simple: we all have an interest in developing software and the most common software development problem I have to solve is that of communication. This could be between the customer and the development team, between members of the development team, between the programmer and the computer, or between the user and the program.

Communication is the point. Software development is largely about communication.

And where does this lead? You may not think that you have much you contribute, but you are wrong - every one of you has a unique experience in software development and knows something that will interest others. The ACCU provides a forum for you to practice communicating.

If you look at the journals you will recognise the same names occurring again and again, and you may think "I can't write as well as that". You may even be right, but how did these people get that good? That is right: practice.

The ACCU works because people participate, and writing in the journals, helping on the email lists and speaking at the conference are all ways of participating. And they are all ways to practice your communication skills.

There is a particular opportunity that I'd like to draw your attention to currently: some of the regular contributors to the journals have been focusing on practising other communications at the moment - they are preparing material for the conference. This means that the editors and readers of the journals will welcome the article that you could write even more than usual.

Don't try to convince yourself that you don't have anything worthwhile to say; those that haven't had them yet value insights and knowledge more easily. It is easy to undervalue what you know: "it's just recursion - everyone knows that". Well, everyone doesn't: if you can write an article that explains it to someone that doesn't then you and they have both gained something. And even someone that does understand it may gain something: a better appreciation of its use, or a way to explain it to colleagues who don't. If just explaining recursion is too easy for you, how about explaining its relationship to iteration, or to mathematical techniques like "proof by infinite descent" and "proof by induction".

[Editor's note: I would be interested in publishing more than one article on these themes. The comparison would help to highlight the most helpful aspects of each approach.]

There is one more thing to hide at the bottom of this report: I propose to make a small change to the constitution. At present paragraph one reads:

> Title 1.1 The name of the Association shall be the C Users' Group (UK) , to be known publicly as the Association of C & C++ Users.

Because of the interest in other programming languages, and things that extend beyond programming languages I think that we should drop the explicit reference to C and C++. I would like to substitute the following:

> Title 1.1 The name of the Association shall be the C Users' Group (UK), to be known publicly as the ACCU.

I don't think this is a contentious change, as it simply reflects the way the association has developed in the last decade. But feel free to comment - either by email or at the AGM.

Membership Report

David Hodge <membership@accu.org>

Facts and figures this time.

We have 1037 paid up members, so at mid January we are 73 down on the final total membership for last year. We will easily make this up before the end of June. 983 members have email, 137 use Standing Order to pay their subscription, 57 people bought Tshirts. There are 39 Corporate members, 72 Student members. We have members in 41 countries other than UK, the top two being 104 in the USA and 30 in Germany.

Don't forget to advise me if you change your email or postal address.

Conference Organiser's Report

Francis Glassborow
<francis@robinton.demon.co.uk>

Barring interference from circumstances outside our control, all seems set for a great conference this year. If you have not yet booked have a look at www.accuconference.co.uk. If you are not coming then you are missing out on the best gathering of programming talent on this planet this year.

As many of you know, I am hanging up my organiser's hat after seven years in the hot seat. We started these events the last time ISO's WG21 met in this country so it seems suitable that I pass the task on to someone else a couple of days before their next meeting here.

Unfortunately there has been a glitch in that Kevlin Henney who had been understudying me finds himself over-committed and unable to take on the job. So we need a volunteer. Whoever it is will get plenty of support from both me and Kevlin and an opportunity to build themselves a reputation with the world's best.

To be honest, there is not a large amount to do, it is mainly basic organisational things like getting invitations to speakers out early, liaising with our commercial organiser, jigsawing the programme together and nagging speakers to send in their biographies, synopses and presentation notes in a timely fashion. There is always the odd panic when a speaker has to drop out, but there is plenty of goodwill around and finding a substitute is easier than you may think (I am pretty sure we must have set some sort of record by getting Bertrand Meyer as substitute speaker a couple of years ago.)

If several of you volunteer, that will be no great problem because the work can always be shared. Just make sure that someone comes forward; most ACCU members could do the job.

So I will see you at the conference where you will be able to read my last report under this hat.

Wanted!

Volunteers to assist with the provision of networking for the ACCU Spring Conference and the WG14 and WG21 meetings. The hotel has all the infrastructure (e.g. 12 ISDN lines with access points in every conference room) but I need someone to help with setting up servers etc. I can even provide at least two suitable PCs, (operating system as desired) at least one wireless access point and two 10 way 10 base-T hubs but we would need to scavenge more) but what I cannot provide is the expertise required to set up the hardware and software.

Contact Francis Glassborow as soon as possible if you can assist:

francis@robinton.demon.co.uk

Standards Report

Lois Goldthwaite
<standards@accu.org>

The big event in the standards calendar this year, at least so far as the UK and ACCU are concerned, is that the C and C++ standards committees are holding their semi-annual meetings in Oxford this spring, hosted by ACCU and generously sponsored by Microsoft.

The C committee (WG14) meets from March 31 to April 4, while the C++ committee (WG21) convenes the following week, April 6 through 11. All meetings will take place at the Holiday Inn at the Pear Tree Roundabout in Oxford, just off the A34 and convenient to the Park and Ride service into the city centre.

If the place and dates seem to remind you of something, that is no accident. The ACCU Spring Conference takes place at the same hotel on April 2-5, and many of the speakers at the conference are drawn from members of the committees. This provides a rare opportunity to hear experts who seldom appear at conferences.

The last time the C++ committee met in the UK was July 1997. The C group was also here in 1997 and again in 1999. All of those meetings were held in London.

If any members of ACCU would like to observe some of the meetings of WG21, please write to standards@accu.org and I will make arrangements.

Dialogue

Francis' Scribbles

Francis Glassborow

Writing a Book

I have long promised to write a book about programming. Just after shipping my last column I settled down to do so. The target readership is people who have no programming background but want to learn enough to start writing their own programs. Note that that says nothing about being able to read and maintain anyone else's source code.

Of course you cannot learn to program without also learning about a programming language. However it is important to keep the objective clear. The language I use in my book is C++, but it is not a book about C++ and so I make absolutely no attempt to teach the language in its entirety.

The experience of writing is proving very interesting. My model student (i.e. the person who is using the first draft to learn to program) is a fifty-year-old housewife with no prior programming experience.

As I work through writing the various chapters certain features of C++ become issues. For example the student needs to be able to prompt for a filename and then open that file. At this stage I do not want to go anywhere near C-style strings. That type of arcane knowledge adds nothing to the process of learning to program.

C++ has a perfectly good string type that can be used to capture the filename from the user. But why do the filestream types lack members to allow a file to be opened with the filename in a string? Yes, of course I know how to use `c_str()` with a filestream constructor and the `open()` member function. But why does the design of C++ force users to use archaic mechanisms?

For the purposes of the book, I just provide a wrapper function in a utilities library but those that wonder why C++ is looked on as a poor first language need to recognise flaws like this one. The next release of C++ needs to ensure that programmers do not **have** to use C-style strings.

Another inconsistency is that the `getline()` member functions of `istream` types do not support `string`. There is a free function that does so. I wonder if this lack (of a member function in `istream` types) is an artefact of `basic_string` and `basic_istream` being templates.

I will have more to say about such issues in future columns.

Choosing Development Tools

One important consideration when tackling a new subject is to keep all the incidental details as simple as possible. Learning to use a word processor having all those hot keys active is unhelpful. The poor novice wishes to type a capital V and hits the ctrl key instead of the shift key. If they are lucky, nothing much happens.

It is particularly bad news to the novice that the ctrl key is so close to the shift key (and on older keyboards the alt key was right alongside). Hot keys are bad for novices.

Now move on to IDEs. These are clearly good things for experienced programmers and the more features the better. Class browsers, automatic generation of a source code framework for a class definition etc. are all great tools for the experienced but not for newcomers to programming.

Now there are some programmers who believe that the right place is to start with the command line. I think that is just as bad, if not worse. The newcomer is once again faced with a raft of things that have nothing to do with learning to program. Most of us find learning one new thing enough.

Fortunately I already knew of an appropriate IDE for newcomers because Al Stevens has written such an IDE for use by readers of his books. It is called Quincy and, in my opinion, provides just about what novices need.

It is greatly to Al's credit that he has placed it in the 'public domain' and allows other authors to use it. I first came across it when reviewing a respectable book on C++ for novices (*C++ by Example, Undersea Learning Edition*) written by Steve Donovan.

The latest version of Quincy works with MingW (a port of GCC/G++ for MS Windows environments) and uses gdb as its debugging tool.

Even better from my perspective, it has no problem with accessing MS Windows graphics support libraries in a console program. That is a requirement because my book leans heavily on the Playpen graphics library that was originally designed in 1996 for the ACCU. I will say more about that shortly.

The combination of Quincy, MingW and gdb provides a sound basis for a newcomer to programming (or just to C++) to work with. It is enough of an IDE to support their needs without having extras that just act as traps for the unwary newcomer.

If you are trying to learn C++ and are using an MS platform I think the combination of Quincy and MingW is the best around. Note that if you tried it earlier, build 8 of MingW removes a number of irritants. I am very grateful for Al Stevens' willingness to address irritants that would affect my readers while being of little significance to his readers. That is a hallmark of someone who takes a pride in his work.

All this is fine for people who use an MS Windows platform. However some of my potential readers will use Linux. I want to address their needs as well. The problem is that I cannot find a minimalist IDE for C++ on Linux. This is an example of the difficulty I have with Linux. If Linux is to be usable by ordinary non-expert members of the public the Linux community must understand the needs of such people.

I know that Linux is a great platform but it is not yet a platform that supports Joe Bloggs. Linux programmers, in common with far too many free software developers, assume a level of expertise and enthusiasm that is inappropriate for a general purpose OS. In a sense the Linux community is the opposite extreme to the Apple Macintosh one. The former assumes technical expertise whilst the latter assumes none.

We need operating systems that meet the reasonable needs of the whole population of computer users. To be honest, I doubt that the Apple computers will ever manage that but maybe that is indicative of my bias. Linux could certainly meet the needs if it came out from its ghetto of techno-nerds. Windows could also do so if it stopped trying to double guess the user's needs. In my experience Windows has shifted towards the Mac end of the spectrum with its XP releases. Its designers should stop and think again. Windows XP is great in many ways but it has some really nasty paternalistic behaviour that bites more experienced users when they are not looking.

So returning to my problem; I want my book to be usable in a Linux environment as well as in a Windows one. Getting Playpen to run in a Linux environment should not be hard because it was designed to port to other platforms. I stand no chance that Quincy can port to Linux because it makes serious use of VC++ wizards and MFC. Note that there is nothing wrong with that because the designer explicitly designed it for Windows programs (though it is mildly amusing that Quincy + MingW cannot compile the Quincy source code, or so Al Stevens says)

Does anyone know of a dead simple IDE for C++ development on Linux?

Something about Playpen

Playpen was written to a spec provided by a friend of mine who writes under the name of 'The Harpist'. The idea was to provide a minimalist graphics resource that could easily be implemented on common computer platforms.

When getting ready to write my book, I got it out and dusted it down. The low level code (MS Windows specific) looked as if it could do with some extra work. Garry Lancaster was one of those who volunteered to have a look.

He did an excellent job with the result that Playpen is written in three layers. The top layer provides what the user needs. The middle layer interfaces between the users needs and the platform specific implementation. And the bottom layer isolates the code that needs rewriting for each specific platform. I will send our webmaster the complete version when Garry has finished adding a couple of late enhancements (adding the ability to query the keyboard directly and query the mouse).

Playpen has very limited built in functionality. It allows you to create a single 512 by 512 graphics window. You can plot pixels in any one of 256 palette-mapped colours. You can query the colour of any pixel. You can modify the palette, change the origin, adjust the plotting mode and

change the scale. Recently Garry has added the ability to read and write the graphics area in PNG 256-colour palette format.

Everything else has to be built on that foundation. This provides a rich environment for the novice programmer who has to learn to do things for himself/herself. If you want to draw lines, circles etc. you have to write the code. Well actually I am not that cruel and provide some implementations to ship with Playpen.

Testers Wanted

While my target readership is the true novice to programming, I am wondering whether the book has anything to offer others as a first experience of C++. If any of you would like to volunteer as testers I would be happy to use the first few volunteers as long as they keep my principal objectives clearly in mind and are willing to make constructive criticism and observations as to how well it worked for them

I would also be interested in anyone who has to teach novices to program that has the time to look at my work and provide constructive criticism. I believe my approach is unique and places considerable demands on the reader whilst not overstressing them.

Of course if you know of a real novice who could act as a second test subject using you as a mentor I would be very happy to provide the current versions of the material. The problem with any test student is that they rapidly disqualify themselves by becoming programmers.

Feedback

The almost complete lack of response to my last column is a little disappointing. Perhaps you all assumed that someone else would write what you thought. Well it does not work that way. If you do not write, then no one does.

My column is in the Dialogue section of C Vu because it is not intended to be read and forgotten. It is supposed to provoke a certain amount of thought amongst its readers.

We cannot afford to remain passive spectators to the process of language development. There are people with enthusiasm who will pursue ideas that may damage our work. They do not do that deliberately but because they take the profound silence as an indication that all either agree with them or do not care.

I often put forward ideas as a mechanism to broaden the range of choice. However that requires others to explore that range of choice and to try to understand the implications.

Let me give you an example based on a couple of exchanges on `comp.lang.c++.moderated`.

The C++ programmers in ACCU generally know that it is not possible to initialise an array in the initialiser list of a constructor. The more knowledgeable may also know that it is also hard to initialise a POD member that way.

So now consider the following suggestion for extending the language:

```
struct X {
  int i;
  double d;
};

class example {
  X x;
public:
  example() : x({0, 0.0}){}
};
class witharray {
  X array[10];
public:
  witharray(int i, double d)
        : array({{i, d}}){}
};
```

In other words allow C style brace initialisation in C++ constructor initialiser lists.

What would be the implications of allowing such a syntax and are there any other features that would need to be added to make this proposal reasonable? Would the benefits justify the expenditure of effort? When thinking about this, consider if the technique could be used to provide initialisation of vectors.

Some of you must have opinions on this, but I will be very surprised if more than a couple of you make the effort to express them.

Problems

Restatement of problem 6

Consider the following definition of a simple function to draw a line across the screen. Assume that the resolution has been chosen so that successive pixels can be generated by incrementing an `int`.

```
void yline(int y, int startx, int endx) {
  for(int x = startx; x != endx; ++x)
    plot(x, y);
}
```

There is a flawed assumption in this code. How should it be fixed? There is also an issue of the line itself. Where does this line end? Is that a reasonable place for it to do so? In the computer world where pixels have finite size, how should we represent a line of zero length?

Another way to represent a line is by providing its start point and its length. What problems might result from such a choice?

Commentary

The easy part is that the body of the function assumes that `startx` is smaller than `endx`. However before you can deal with that assumption you must look carefully at the code and what it does.

It treats a line as made up of a finite number of pixels. Note that a pixel is not a mathematical point. We have to consider whether it is correct to omit plotting the end pixel. If you think of the end as a point (without magnitude) then the answer is no. But that is not what we have.

Should we plot a pixel for a line of zero length? As a pixel has finite magnitude, I think not. The above function works quite happily for zero length lines, it plots nothing.

If that does not convince you, what should you get if you execute the following two statements:

```
yline(5, 0, 10);
yline(5, 10, 20);
```

Surely you should get a line twice as long as either of the two parts. And what about when you use exclusive or plotting? Surely you should not get a line with a pixel missing?

The function that I have given meets normal expectations, two equal length lines back to back produce a line that is exactly twice as long and even with xor plotting there will be no missing pixel.

I believe that those that think the above function stops too early are confused by the difference between mathematical points and real world pixels. It is not that we can only approximate a point in the real world, a pixel is not and never can be even an approximation to a point. That means that the mathematical concept is inappropriate.

The above implementation for drawing a line on the screen considers a line as a container of pixels and, as such, is entirely consistent with the way C++ handles containers. However this view does not come for free. It now matters whether we draw a line from `(startx, y)` to `(endx, y)` or from `(endx, y)` to `(startx, y);` the omitted point will be different. Once we are willing to accept that we can rewrite the code above as:

```
void yline(int y, int startx, int endx) {
  if(startx < endx) {
    for(int x = startx; x != endx; ++x)
      plot(x, y);
  }
  else {
    for(int x = startx; x != endx; --x)
      plot(x, y);
  }
}
```

I think that the surprise that this causes many programmers is the same one that newcomers to the STL have when they first encounter the fact that the `end()` iterator does not refer to an item in the container. With experience you come to accept that as correct. You also come to understand why you cannot go backwards through an STL container by starting at `end()` and finishing at `begin()`.

Yet despite their familiarity with the STL, almost every programmer I have shown `yline()` to expresses the opinion that it is wrong because it stops a pixel too early. None of them stopped and asked what the function was for.

[concluded at foot of page 8]

Student Code Critique Competition

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.

Student Code Critique 19

The Code

I'm trying to write a class to represent a card that can used to create a pack of cards. I'm thinking an array of pointer to card in the back of my head.

```
class Card {
public:
  Card():itsNumber(0) {}
  Card(int Number):itsNumber(Number) {}
  virtual ~Card(){};
  void SetNumber(int val){ itsNumber = val; }
  int GetNumber() const { return itsNumber; }
  virtual void Display() const =0;
private:
  int itsNumber;
};
const int RegularPack = 54;

int main() {
  int i;
  Card* pack[Regularpack];
  Card* pCard;
  for (i = 0; i < RegularDeck; i++) {
    Card:pack[i]->SetNumber(i);
  }
  cout << pack[50]->GetNumber() << '\n';
  for (int i = 0; i <= 53; i++) {
    cout << i << endl;;
    cin >> *pack[i]->SetNumber(i);
  }
  cout << 'The CARD is: ' << Card << endl;
}
```

The entries

From Ivan Uemlianin <i.a.uemlianin@celtic.co.uk>

The student introduced their code saying, 'I'm thinking of an array of pointer to card in the back of my head.' My approach here is that the route from back-of-head to code has been too direct: front-of-head could probably have helped out a bit.

I'll deal with design issues first, and then idiom and syntax together. Design issues are more fundamental and (hopefully) being clear at the design stage will help avoid later mix-ups.

My first advice is to do things one at a time. Multitasking is good in an operating system, but when people do it, there is generally a price to pay. The code submitted is not in any particular programming language, is half designed and half hacked together in `main()`. Focus on the task itself, then the design (i.e. how you'll tackle the task), and then the implementation.

Design

My second advice is to go and work in a room without a computer. Stop thinking about programming computers altogether, and start thinking about playing cards. What things or ideas are in your head when you're playing cards? Get a pack and deal yourself a few hands if that would help. Write down every thing or action that occurs to you. After a short while, start thinking about which parts of all this – what real-world concepts – you want to model, and in how much depth. It's important at this point to be clear about why you're working on this ('Because my teacher told me to,' doesn't count); for sake of argument, let's say you're building a library that can be used by card-game-playing applications.

You'll probably have quite a list: as well as 'playing card', you might have 'pack of cards', 'suit', 'rank' (i.e. 2-10, J, Q, K, A), and maybe more; shuffle, deal, and so on. Start to link up objects with each other and with attributes and operations. For example, suit & rank go with card, but shuffle and deal go with pack of cards. Also, the kind of pack might determine what suits and ranks are available.

Now you can start to sketch all this out – in pseudocode: thinking about implementation details too soon can do more harm than good. Err towards your native human language, rather than your favourite programming language. Here's how some of it might look:

Card:
- *suit (e.g. Hearts, Diamonds, Spades, Clubs)*
- *rank (e.g. 2-10, J, Q, K, A)*
- *show() (turn face up; let the other players see)*

PackOfCards:
- *cards (container of Cards)*
- *cardType (something to determine what suits & ranks are allowed)*
- *dealCard() (take from top of pack)*
- *returnCard() (put back to bottom of pack)*
- *shuffle()*

Up to now, there's been no need to limit yourself to a single programming language. All the languages you know have been giving you ideas to think around the problem. Now you decide, 'Let's do it in C++!' Get out your favourite C++ books and put them in a prominent place on the table (you're still in your computer-less room, I hope).

Implementing the above in C++ we might get something like this:

```
#include <iostream>
#include <queue>

class Card {
public:
  Card(Rank&, Suit&); // will most often be
                      // generated as part of a PackOfCards
  ~Card(); // does this need to be virtual?
  Suit getSuit();
  Rank getRank();
  void show(); // turn face up
  void hide(); // turn face down
  void display(); //generate on-screen representation
private:
  Suit suit;
  Rank rank;
  bool face_up;
};
```

[continued from page 7]

Sometimes using familiar terminology actually leads us astray. I think that most seeing that function for the first time think in terms of mathematical lines and plotting points even though they know that you cannot draw a mathematical line by plotting points.

So what do you think now?

Problem 7

Look at the following code that is designed to draw an approximate circle by drawing a polygon with a sufficiently large number of sides. The function that draws a side is based on the same design principles as `yline()`. The supplied software uses the closest pixel to each vertex as the start/end point of a side.

The programmer has determined that a 100-sided polygon is indistinguishable from a circle when drawn at a resolution of 800 by 600. However he is very surprised when he draws a set of nested circles to create a filled in disc because the result is a doughnut shape with a central disk in background colour. Why? How should he change the circle drawing function?

```
void drawcircle(double radius, point2d centre) {
  drawpolygon(radius, 100, centre);
}
```

```
class PackOfCards {
public:
  PackOfCards(); // generate one Card of each
                 // Suit/Rank combination
  PackOfCards(CardType); // in case we want to
                         // support exotic packs
  ~PackOfCards(); // does this need to be virtual?
  void shuffle();
  Card dealCard(); // take card from top of pack
  void returnCard(Card); //put card to bottom of pack
  bool empty(); // have we reached the bottom?
private:
  PackType type;
  queue<Card> cards; // queue models how we'll
                     // use the pack
};

class Suit { /* ... */ };
// e.g. Hearts, Diamonds, Spades, Clubs
class Rank { /* ... */ };
// e.g. 2-10, K, J, Q, A
class PackType { /* ... */ };
// e.g. Tarot, European, etc.?

int main() {
  Suit* spades = Suit('s');
  // make sure Suits, Ranks and Cards
  Rank* queen = Rank('q');
  // can be declared like this,
  Card* queenOfSpades = Card(queen, spades);
  // or provide better functions
  p = PackOfCards();
  p.shuffle();
  while(!p.empty()) {
    Card c = p.dealCard();
    c.display();
    if(c == queenOfSpades)
    // how else might I compare Cards?
      cout << 'Argh!' << endl;
  }
};
```

I've left some things out: for example, are Aces high? How do you compare suits? Sometimes it only matters whether a suit is red or black, sometimes suits are ranked (e.g. Spades beat Clubs), sometimes there are trump suits. Should this be modelled in `Suit` or `PackType` or another class `Game`? But you have to stop somewhere. You might decide it's not necessary to model suits and ranks like this, and just have `string`s and `int`s (with King = 13?).

The code is not complete. As you can see, I've just done the declarations and a test frame. You should leave out the definitions – even the easy one-liners – till the declarations have settled down a bit. Consequently, the comments are important. They're not for 'someone else' to read (and suffer) for no apparent reason: they are for you to read when you come back to fill in the rest of the code.

From now, it might be safe to start hacking.

Syntax and Idiom

I think most of your syntax errors, at least in `class Card { ... };` are down to not really committing to C++: there is another language interfering (I admit I have no idea what it is. Something from Microsoft?). The errors in `main()` are more to do with programming-as-you-go-along, and generally result in inconsistency and/or incoherence.

(a) `int i` and the for loops

`int i` is declared on line 14 and used as a count variable in the two for loops at lines 17 and 21. In the latter for loop it is re-declared, an error. Comments:

- I would tend to declare a for loop's count variable within the loop, based on a general logical principle of declaring things close to where they're used. If `i` is not used outside the loops, and if it doesn't need to preserve state between loops, it doesn't need to exist between loops.
- I would probably rename `i` as something more meaningful like '`count`'.
- The books tend to prefix the increment operator in for loops. I think the difference is mainly semantic in this case (i.e. you want to return the new value).
- In both cases you're looping through a pack of cards. Why describe it differently each time? Using a meaningful variable name for your sentinel is better than using a meaningless number. Go easy on your future self.
- So, lines 17 and 21 could both be written as:

```
for(int count=0; count < regularPack; ++count) {
```

(b) `RegularPack`

- I assume `RegularPack`, `Regularpack` and `RegularDeck` are all meant to refer to the same `const int`. If so, they should have the same name.
- If you're following the idiom of class names with an upper-case letter, and variable names with a lower-case letter, then '`regularPack`' would be better.
- As noted above I assume the '`<= 53`' in line 21 really means '`< regularPack`'.

(c) line 16: `Card* pCard;`

This isn't used anywhere! If you're intending to come back and do something with this you should have a comment saying what it's for. Don't rely on understanding your own code when you come back to it.

Summary

The phrase, 'back to the drawing board,' sums up this critique. I imagine almost all of the errors in the submitted code – the lack of design, the hybrid language, the incoherence – were caused by hacking without thinking, a kind of unconscious programming, going straight from the back of the head to the screen. Don't rush to hack. Tarry awhile at the front of the head.

From Simon Sebright

<simonsebright@brightsoftware.freeserve.co.uk>

Well, fresh from the success of last time's competition, I'm back at the desk. Not so much homebrew tonight, so the coherence may be different. This time, I'm facing unemployment due to redundancy. Hrmmmph. But, I find this a good way to hone myself for potential interviews. Needs must when the wife drives. Anyway, about cards...

Having just written the following critique, I'll give the order of importance of the main points I found:

- There's no Pack class
- A distinct lack of comments
- The difference between a pointer and what it points at
- Using the standard container classes where appropriate
- Using constants, not hard-coded values

Something I noticed after the fact was a lack of comments. Being so focused on the code, its mistakes, etc. meant I didn't keep my eye on the ball that is the larger picture. Of course, comments should only be there to add value, so if the student felt that the code was self-explanatory, then so be it. And I am a Dutch uncle. I've asked interviewees about self-documenting code in the past and usually got blank looks. I like the ideas behind it, and it has often lead me to suggest code snippets that are more verbose than need be, much to the delight of the vultures on `accu-general` ;-)

This time, we have a (vague) requirement. The top-level requirement would appear to be to be able to represent a pack of cards. And what better way to do it than with some cards. More on the classic language problem later, but I think it's more important to discuss what we're trying to do, rather than what's wrong with a first attempt.

The most obvious point is that there is no class called Pack, Deck, whatever you like, just Card. There should be. My thoughts on Packs led me to go up, down and around as to what a card is. The first observation on the proto `Card` class is that its data member is an `int`. That is, each card is represented by a number. This would probably seem odd to most people, who would identify a card with two parameters: a suit and a number (let's assume King, Queen, Jack, Ace are numbers for now). This guy's only got one. More later.

Well, what is a pack of cards? It depends why we need it, which we don't know. The most obvious example is to develop a card game program, in which the pack represents the available cards to play with. So, 52 cards, then. Well, some games have two jokers, so perhaps we can extend the pack to 54. Now, what happens to the suit, value characterisation of a card? It disappears. Suddenly, our student's `int` doesn't seem that daft. Jokers are cards 53 and 54. Now, suppose you are writing software for a factory that produces packs of cards. You'll need at least one card with the instructions for Bridge on it. Enter cards 55, ...

So, the simple idea is not really all that simple. Do you have an abstract class, `Card`, and sub classes for `PlayingCard`, `Joker`, `ExtraCard`, etc.? Exploring that idea, we might have `GetNumber()` as an abstract function on `Card`. Playing cards have a suit and value, so can calculate a number (putting the suits in a given order). Jokers can perhaps be red or blue, and they know whether they're 53 or 54 accordingly. `ExtraCards` can be initialised with a number and a description.

Or, we can have the number in the base class as above, but functions on the `PlayingCard` class to calculate a suit and value from it. Don't really like that, though, although it does have the number as the consistent thing between cards. Just consider that the constructor of the class `PlayingCard` would take an `int` rather than value and suit pair. Ugghh!

Now, as we've got different concrete classes, we can't simply have an array of types, whether it's a C-style array or not. `vector<>` only takes one type for things it stores. Either we can have the `Pack` keep separate arrays for the different types, or we have to keep an array of pointers, and thus we can store them all as `Card*` s. At this point, I waffled, deleted and thought what kind of access to the `Pack`'s `Card`s do we want? An `operator []` returning a `Card*` of a given number? An index function taking a suit and value returning a `PlayingCard*`? It depends on what the user of `Pack` wants to do. If we don't expose the latter, though, then the user will be potentially be downcasting to get `PlayingCard` pointers from `Card` pointers.

Let's roll with a `vector` of `Card` pointers. Similar to the given code, so I can discuss the problem with it. Why use a `vector` as opposed to C-style array, though, first? Why not? It's the standard thing, you know (s in stl, in fact). You get the same semantics and more (it grows, for example).

OK, but, our `vector` contains pointers, not real things, so what do the pointers point at? What do your pointers point at? Nothing! That's right, if you have a pointer to something, there are two things to worry about, the something and the pointer to it. You've only got the pointers, not the somethings. They need something real to point at. You need to assign values to your pointers, those values being the addresses of the somethings to point at. Our `Pack` class owns the `Card`s it's got, so it should also govern the lifetime of the cards, i.e. create and destroy them. When? In the constructor and destructor would be my recommendation. What's a `Pack` without some of its cards? How much less there would be to learn if we didn't have disparate types to worry about. In that case, and in the case of the student's code, an array of actual objects would have been much more suitable. No messy cleaning up to do.

At this point, I've just reread the code for some more ideas and I see the addition of a virtual function! `Display()`. Aha! What's going on here, then? What derived classes were you thinking of? One for each value/suit pair and the joker? One for a playing card, one for a joker? Hmm. Why put the `itsNumber` member in, then? It's all getting rather messy and without a bit of clarification, could lead to me being here all night discussing the issue.

Of course, the code given won't compile. The constant `RegularPack` is used with two other names. The first, `Regularpack`, suggests a familiarity with case-insensitive languages. C, C++ are case-sensitive. The second, `RegularDeck`, suggests a trip to America.

There's an interesting `Card:` prefix to accessing the array. Not sure what language that's imitating, but it's not correct. The compiler knows that an element of the pack array is a `Card*`, because that's how you declared it.

No idea what the `cin` line is trying to do in the second loop. Are we trying to allow the user to set a number of a card, or choose a card? It looks like gibberish to me, so I'll give the student the benefit of the doubt and assume his girlfriend rang in the middle of the exercise. I hope it won't compile because the compiler is expecting to be able to take an input value and assign it so the expression on the right, which is not an l-value. (Confusion might arise here, l for left, usually you have `int myLValue = thatRValue;`)

The last `cout` line seems to want to output to the user the class definition `Card`! This isn't allowed, the `<<` operator must take an instance of a class or built-in, not the class definition. Not sure what was intended.

Given the compilation problems, there are non-compilation observations – an extra semicolon at the end of one line. Not a disaster, just a null statement, but untidy.

Having defined a constant for the number of cards in a pack, you then use the magic number '`53`'. '`i < RegularDeck`' means you can change it once (if you have three jokers, perhaps).

Given the gibberish nature of the `cin` line, it might get taken out altogether, but I'm not overly happy with the precedence of the values bits of the expression. There seems to be an extra `*` at the start, yes, that's it. You either do `*p.Member()` or `p->Member()`, but the former must (I think off the top of my head) be written as `(*p).Member()` to indicate that you are taking the address of `p` rather than the address of `p.Member()`. The compiler should help here, at the very least by giving you a cryptic error.

I've now just gone back in time to write the summary. I feel a bit mean in not supplying much of an alternative. I hope my discussion above indicates the various possibilities, and given the absence of any more information, I'll err on the better side of discretion.

From Brett Fishburne <William.fishburne@verizon.net>

[Someone suggested a while back that these critiques be written more akin to the Bridge game critiques. Here is my lame and long attempt at such an effort.]

Maurice Math and Ollie Object had finally created a joint venture, Math & Object, LLC. As the opened the doors for business, Crafty Card came in describing his desire to have a street card deck that would allow him to run some street gambling operations with very good percentages. Maurice and Ollie were only too happy to oblige.

Once they wrote their initial effort, as usual, there were a few compilation bugs to be resolved. The first and easiest was to refer to the constant name `RegularPack` instead of `Regularpack` in the array declaration. Then there was a similar syntax error using `RegularDeck` instead of `RegularPack` in the first for loop. Another mistake in variable naming showed up on the last line of the program where `Card` had been used inadvertently instead of `pCard`.

The ugly issue of namespaces came up immediately thereafter as Ollie had warned, and `std::` was prepended to references to `cout` and `cin` and `#include <iostream>` was inserted at the top of the file. The reference to `Card:pack` had been slipped in by Ollie after Maurice had successfully argued that no namespace was *really* needed for such a simple class, so it, too, was summarily removed.

One compiler error remained which had Maurice shaking his head in frustration:

```
void value not ignored as it ought to be
```

This error occurred on the only line that used `cin`. Ollie was quick to identify that `SetNumber` returned `void`, and the two quickly realized that the number from `cin` was never actually being inserted the way the line was phrased. Instead, they added another line to `cout` and then accepted the number from `cin` into a temporary variable declared within the loop and sent that number to `SetNumber` (while making these adjustments, they also corrected the direct reference to '`53`' and referred to `RegularPack` instead):

```
for (int i=0; i <= RegularPack; i++) {
  int TempCardNo;
  std::cout << i
            << ': Enter a new number for this card: ';
  std::cin >> TempCardNo;
  std::cout << std::endl;
  *pack[i]->SetNumber(TempCardNo);
}
```

Unfortunately, this did not resolve the error! So, there was more going on then met the eye. Maurice and Ollie noodled this one together until they finally realized that they had accidentally dereferenced the `pack` variable in the for loop. After removing the '`*`', the program compiled successfully. The program was then delivered to the customer, Mr. Card (listing 1).

Mr. Card, thrilled with his new application, took it immediately to the street and took bets. He invoked the program and was shocked to find only '`Memory Fault`' as the program response. Embarrassed, Mr. Card covered the bets he had erroneously taken and returned the program to Math & Object, demanding satisfaction. Math & Object cited their long history with Microsoft and assured Mr. Card that this type of problem was typical in any application and that they would be on it right away!

Ollie Object immediately leapt on the use of a pointer for the `Card` variable as he was well aware that most memory faults were from bad pointers. He insisted that the array be made up of `Card`s and not pointers to `Card`s. Maurice conceded and the change was made. Once more they compiled the program only to face a huge new collection of errors. The errors seemed to be related to the use of a virtual function. Ollie blushed immediately realizing that in the early design phase he had planned for `Card` to be an abstract class and that when Maurice had insisted things were too complicated, the abstraction concept had been removed. Ollie quickly commented out the virtual before the destructor and removed the `Display` function that, while designed, had never been implemented.

```
#include <iostream>

class Card {
public:
  Card() : itsNumber(0) {}
  Card(int Number) : itsNumber(Number) {}
  virtual ~Card() {};
  void SetNumber(int val) { itsNumber=val; }
  int GetNumber() const { return itsNumber; }
  virtual void Display() const = 0;
private:
  int itsNumber;
};

const int RegularPack = 54;

int main() {
  int i;
  Card* pack[RegularPack];
  Card* pCard;
  for (i=0; i < RegularPack; i++)
    pack[i]->SetNumber(i);
  std::cout << pack[50]->GetNumber() << '\n';
  for (int i=0; i <=53; i++) {
    int TempCardNo;
    std::cout << i
              << ': Enter a new number for this card: ';
    std::cin >> TempCardNo;
    pack[i]->SetNumber(TempCardNo);
  }
  std::cout << 'The CARD is: ' << pCard << '\n';
}
```

Listing 1

The two programmers compiled once more only to find that the slew of errors largely remained with new errors showing up because of the use of '`->`' to dereference instead of '`.`'. These last were quickly resolved. Once more the pair compiled and once more their console filled with errors related to `ostream`. At last Ollie realized that instead of printing the card number for `pCard` on the last line, they had, instead, printed the object itself. Of course, `ostream` couldn't handle this without befriending the `>>` operator for `ostream`. Given that Ollie frowned upon the use of `friend`, he resolved to print out the card using the previously written `GetNumber` function and adjusted the line accordingly. The program successfully compiled once more!

Triumphant, Ollie and Maurice called upon Crafty Card and delivered the newly fixed program (listing 2) declaring that it would now stand up to the rigors of the street and identified 'magnetic interference' as the problem advising that Mr. Card not set up his card game too close to the subway.

Mr. Card, thrilled to be back in business once more, promptly went out to use his new program. He took bets once more and demonstrated that by setting all of the cards to the value 0, no matter which card was selected, the return from the program was 0. And, indeed, this was the case. He then said he would set the cards all to the value 2 and asked a young bettor to, Select a card, any card and guess the value returned. The bettor selected the 15th card and suggested that the value would be '4 of clubs.' Crafty carefully explained that the cards were really only numbers (which confused numerous bettors) and finally the young bettor said that the return would be 2 reasoning that if Crafty had set all the cards to 2 then he couldn't lose. Crafty set all the cards to 2 and the card selected was returned as 0. Crafty collected his money and the bets continued, but the atmosphere soon grew sour. The bettors began guessing that 0 would be returned no matter how Crafty set the cards and the money poured from Crafty's pocket until he had to make a quick retreat to the establishment of Math & Object.

Maurice and Ollie grew faint while Crafty Card described exactly how one walked with broken knees and they turned to the matriarch of their craft, A.C. Cu, for help. While Ms. Cu had much to do, she was ever eager to help young programmers find their way to both competence and prosperity. After looking over the malformed program, A.C. began the following monologue:

'How is it possible to have a deck of cards without suits, nay, without even an indicator to distinguish the royalty from the commoners? Do you even know what a card is?'

Maurice, stunned, attempted to reply (he had decided that the use of a pure number was so much more serene then the jumble of letters and numbers and <gasp> suits), but was whisked to silence by a wave of Ms. Cu's hand indicating the question had been but rhetorical—

'If you are going to build a card, then build a card, not a simple string of numbers. Having done so, allow the cards to be organized into a deck, allow the deck to be shuffled, and, for the love of God, allow one to select an individual card from that deck. Make use of the STL to simplify your life. Finally, but most importantly, build a test harness that tests your code and do so thoroughly before offering Mr. Crafty Card another partial solution.'

She then produced two of the coveted C Vu beanies with the bright yellow propellers and having set such upon their brows, returned them to their tasks.

The two returned, enlightened, and ready to work. Maurice Math quickly calculated the following:

1) There are but four suits (Clubs, Diamonds, Hearts, and Spades).
2) Each suit has exactly thirteen cards.
3) The first card of each suit is an Ace (A) and the last three are denoted Jack (J), Queen (Q), King (K) with the intervening cards numbered from 2 to 10.
4) The total number of cards in a deck is 52.

Ollie Object, meanwhile, cracked his copy of *The C++ Standard Library* (Nicolai M. Josuttis – shameless plug for same) and considered the options. There was no question that a container was needed. That this container should allow some sort of searching based upon the position of the card in the deck. In addition, it will be necessary for the position of cards in the deck to be shuffled. Ollie, however, is leaning back towards using a standard array when he stumbles across the array wrapper (p. 219, see also Stroustrup, *The C++ Programming Language, 3rd Edition*) and he is off. It felt good and natural for Ollie to reuse code from another source in his new application.

The deck structure having been determined, the card itself needed some clarification. Each card is actually defined by two elements, the value (Ace-King) and the suit (Clubs-Spades). The linking of these two elements had to be more than coincidence and a quick look at Josuttis made it clear that a `pair` was what was needed.

Ollie had created enumerated types (`Face` and `Suit`). The `Face` types started with 1 (the traditional value of an ace) rather than the typical 0 as this might be of value in future applications although it made little difference for this program. Thus, a given card was defined as a `pair` of `Face` and `Suit`:

```
typedef std::pair<Face, Suit> Card;
```

```
#include <iostream>

class Card {
public:
  Card() : itsNumber(0) {}
  Card(int Number) : itsNumber(Number) {}
  ~Card() {};
  void SetNumber(int val) { itsNumber=val; }
  int GetNumber() const { return itsNumber; }
private:
  int itsNumber;
};

const int RegularPack = 54;

int main() {
  int i;
  Card pack[RegularPack];
  Card pCard;
  for(i=0; i < RegularPack; i++)
    pack[i].SetNumber(i);
  std::cout << pack[50].GetNumber() << '\n';
  for(int i=0; i <=53; i++) {
    int TempCardNo;
    std::cout << i
              << ': Enter a new number for this card: ';
    std::cin >> TempCardNo;
    std::cout << std::endl;
    pack[i].SetNumber(TempCardNo);
  }
  std::cout << 'The CARD is: '
            << pCard.GetNumber() << std::endl;
}
```

Listing 2

```
#include <cstddef>
#include <iostream>
#include <pair.h>
#include <stdlib.h>
#include <time.h>
//#define TEST_ALL
#ifdef TEST_ALL
#define TEST_INITIALIZATION
#define TEST_SHUFFLE
#endif

template<class T, size_t thesize>
class carray {
public:
// type definitions
  typedef T          value_type;
  typedef T*         iterator;
  typedef const T*   const_iterator;
  typedef T&         reference;
  typedef const T&   const_reference;
  typedef size_t     size_type;
  typedef ptrdiff_t difference_type;
// iterator support
  iterator begin() { return v; }
  const_iterator begin() const { return v; }
  iterator end() { return v+thesize; }
  const_iterator end() const { return v+thesize; }
// direct element access
  reference operator[](size_t i) { return v[i]; }
  const_reference operator[](size_t i) const
               { return v[i]; }  // size is constant
  size_type size() const { return thesize; }
  size_type max_size() const { return thesize; }
// conversion to ordinary array
  T* as_array() { return v; }
private:
  T v[thesize];//Fixed-size array of elements of type T
};

enum Suit { Clubs, Diamonds, Hearts, Spades };
enum Face { Ace=1, Two, Three, Four, Five, Six, Seven,
              Eight, Nine, Ten, Jack, Queen, King};

typedef std::pair<Face, Suit> Card;

ostream &operator<<(ostream &out, Card aCard) {
  switch(aCard.first) {
    case Ace:  out << 'Ace '; break;
    case Two:  out << 'Two '; break;
    // <ten similar lines snipped>
    case King: out << 'King '; break;  }
  switch(aCard.second) {
    case Clubs: out << 'of Clubs'; break;
    /* <similar lines snipped> */ }
  return out;
}
const size_t RegularPack = 52;

int main() {
  carray<Card, RegularPack> deck;
  // Initialize the deck to ascending face within ascending
suit
  {
    int deckPosition=0;
    for(Suit aSuit=Clubs;aSuit<=Spades;++((int)aSuit)) {
      for(Face aFace=Ace;aFace<=King; ++((int)aFace)) {
        Card aCard(aFace, aSuit);
        deck[deckPosition++] = aCard;
      }
    }
  }

#ifdef TEST_INITIALIZATION
  for (carray<Card, RegularPack>::const_iterator
          pos=deck.begin(); pos != deck.end(); ++pos)
{
    std::cout << *pos << std::endl;
  }
#endif

  // set the seed for the random number generator
  srand((int)time(NULL));
  for(int deckPosition=0;
        deckPosition < RegularPack;
        ++deckPosition) {
    Card aCard=deck[deckPosition];
    int swapSpot=(rand() % (RegularPack-1)) + 1;
    deck[deckPosition] = deck[swapSpot];
    deck[swapSpot] = aCard;
  }

#ifdef TEST_SHUFFLE
  for (carray<Card, RegularPack>::const_iterator
        pos=deck.begin(); pos != deck.end(); ++pos) {
    std::cout << *pos << std::endl;
  }
#endif

  // Ask the user which card is desired
  {
    int deckPosition=0;
    while((deckPosition<1)||(deckPosition>RegularPack)){
      std::cout << 'Enter a card position: ';
      std::cin >> deckPosition;
      if((deckPosition<1)||(deckPosition>RegularPack)) {
        std::cout << 'Please enter a deck position '
        << 'between 1 and ' << RegularPack << std::endl;
      }
    }
    std::cout << 'The CARD is: '
              << deck[deckPosition-1] << std::endl;
  }
}
```

Listing 3

Maurice pointed out that since `Card` was no longer an integer, it would be necessary to overload the `ostream` operator `<<` so that the `Card` could be printed out. This was done with a `switch` for each pair member.

Maurice then coded the main program that initialized the deck to a default configuration (ascending face within ascending suit) using nested for loops. Ollie and Maurice remembered the need to test as they went, so they then printed out the initialized deck and found that all was well. They contained this code within a `#ifdef` that let them turn their test on and off.

Maurice went and researched many different ways to shuffle cards, but settled upon a simple mechanism that worked reasonably well, but wouldn't have done for a true `Casino`. Stepping through the deck from front to back, a random number is generated for each card and the current card is swapped with the card at the position generated. Again, a test harness for this function was generated and many tests were run to confirm that all the cards were present and that the order changed with every call to shuffle.

Ollie then added the user interface back allowing for Crafty Card to put in the bettor's selection. After a little more testing he realized that it would be necessary to make sure that the number entered was within the range of a deck of cards. Thus, the request for a card number was put into a while loop that checked the range conditions. This too was tested and then the entire program was tested.

Finally, Ollie and Maurice were ready to deliver their code (Listing 3 - see next page). After some use, Crafty Card was so pleased that he decided to let the Math & Object founders keep their knees.

The Winner of SCC 19

The editor's choice is: Brett Fishburne

Please email `francis.glassborow@ntlworld.com` to arrange for your prize.

Student Code Critique 20 can be found on page 14

The Wall

Letters to the Editor

C Vu 14.6 Comments

James,

This is probably a little too late for 15.1. One of the penalties of living in the boonies (San Francisco Bay Area 8-)) is that C Vu rarely arrives more than a day before the copy date.

PDF files

My current employer used to use PostScript for electronic papers, but we have almost completely shifted to Acrobat now. As an academic institution we try to restrict ourselves to Acrobat 3 format documents unless we need some feature only present in 5. Our main problem is ensuring that people embed all fonts. Mathematical formulae are very hard to read after font substitution has removed all the symbols and Greek letters.

We are also noticing an increasing number of PDF files that crash older PostScript printers. It seems to be a memory problem rather than PostScript version.

I have also used Ghostscript to generate PDF files. It may not be the best around, but it is about as cheap as one could wish!

Python

I have no problems with a regular Python section. If people are willing to write it, I will read it. Given that my current system administration role has virtually no requirement for separately compiled languages I am more interested in Python than C++ these days.

Linux

I came across a good implementation of a CD-bootable Linux from `www.knoppix.com` recently. It is not suitable for running a full-time server, but it is useful to be able to boot a CD that does not need any hard drive space at all. Any customisation can be done by floppy disc. The boot image can read most Windows partitions, and includes a number of system and network tools for emergencies.

The image runs from a RAM disc and the compressed images on the CD, so it helps to have at least 128MB of memory on the machine. It is surprisingly tolerant of the hardware - even my venerable Compaq Armada 1500 laptop will run it, though anything more than 640x480 is unstable.

Conference

The program looks excellent - certainly on a par with the best I have seen offered in this part of California. Unfortunately there is no way I can justify the combined flight/fees/accommodation this year.

Regards,

Graham A Patterson

Thanks for writing, and for the PDF and Linux tips. I'm glad to hear that you are one of those happy to see Python material in C Vu - there's more this issue, and more to follow. It is just so hard to make myself practice using Python when I can fall back on Perl, but the formality of Python makes me think that it's a better match for C and C++ in general.

As for conference costs: yes, for those travelling from the US the price of admission is only part of the story. However, even including flights and accommodation the price compares favourably to most commercial conferences, and few if any can compete in terms of what is on offer. *James*

Changing stance?

James,

Thanks for your C Vu editorial last month, most interesting - do I detect a change of editorial stance in C Vu? :)

I think all ACCU'ers are aware of the skills v. theory argument by now, usually appearing under the "Why don't Universities teach useful skills?" heading, in these discussions the theoretical stuff usually gets hammered. But as you point out Turing machines, the halting problem and such has a place in the curriculum. After all, Universities are academic institutions so why do we criticise them for teaching abstract academic ideas?

This debate has been much on my mind in the last few months as I have absented myself from the code-face this year to return to academic study. I increasingly found my software skills were not enough in the complex world. So far my course has been hard work but rewarding and has given me some interesting insights into this debate.

Computer people are not the only ones to have this debate about vocational learning v. academic teaching. I read today that the British Government are seeking to re-address this balance ("Students to be offered vocational degree programmes", FT, 14 January 2002). People are seeing renewed merit in the old polytechnics with their vocational bias.

However, I doubt that formal classroom learning can solve the problem, even if our Universities take to teaching (modern) C++, Perl, Apache, etc. I would like to suggest that it is just not possible train programmers as we might imagine they could, that is, we are wrong to expect someone to leave school with a diploma in C++ programming and be instantly usable in our code shop.

I base this statement on two observations. First, much of the knowledge experienced programmers use is not explicit knowledge, it is in fact tacit knowledge which defies codification, we can only learn by being in the environment, observing and doing. Intuitively we know this: when I come afresh to a programming project I am able to draw on my experience of programs to *feel* how the new one works. Over time I develop an empathy for the program such that I have a feel for the bugs and the flow of the data around the system.

This brings me to my second point. If we accept the work of John Selly Brown and Paul Duguid ("Organisational learning and communities of practice", Organizational Science, February 1991, also at `http://www2.parc.com/ops/members/brown/papers/orglearning.html`) much of what we learn we learn through enculturement. By immersing ourselves in the programming culture we learn how to use our tools, how to use our chosen language, how to interact with our operating systems. There are even different cultures of programming, so one group believes "`delete this`" to be an evil statement never to be used, while others accept it as quite a useful technique.

Each company has its own culture, we learn about our organisations through enculturement, it just so happens that programmers have a sub-culture that we carry with us. Firms which have a programmer friendly culture maximise their return on programmers' talent, those which fail to recognise programmer culture, say by making us work to ISO 9001 standards, will not deny themselves our best work.

Therefore, it is actually impossible for schools to teach us the kind of skills many people call for. These skills can only be learned "on the job."

So where does this view leave us? Well, firstly it lends support to the craftsman view of software engineering. Apprenticeships represent a means of culture based learning which has served us well in the past.

Beyond this there is good news and bad news. The bad news is while we can look to Universities and schools to improve their courses we shouldn't expect them to resolve the issue. The good news is that the role of the ACCU is clearer than ever - the ACCU is part of the learning culture.

Before we dismiss schools completely I think they can find a role for themselves. Yes, the entry level courses can be improved, they must be broadened and teach people the importance of continual learning. Beyond this there is a model we can look to, that of Business Schools which specifically target experienced people for some courses. This is not to say software engineering should become a Business School discipline but there may be a role for academic institutions to pay in offering advanced programmers to experienced software engineers.

Now if you will forgive me I have an exam in a few minutes....

Allan Kelly

A change in editorial stance? Inasmuch as I am not the same editor as Francis, that is inevitable. I agree with Francis on a broad range of issues, but we should remember that editorial viewpoints are the opinions of one person. I am not letting Universities off the hook; we cannot expect them to turn out industry-ready software engineers, but we can at least expect that what they teach should not encourage bad practice. You might be interested to know that I plan, in a future editorial, to talk about the need for better bridges between software engineering and other groups... *James*

Features

A Programmer's View of Book Writing

Silas S Brown

At present I am writing my PhD thesis. It's the largest single work I've yet written and it can probably be regarded as a book, although there are some differences between this and writing a "real" book for commercial publication. However, strangely enough I'm not really thinking of my thesis as a book; it bears more resemblance to a programming project.

Consider for example my directory structure. I'm not writing everything in a single big file (that could get unwieldy); I have "include" files. I have a separate subdirectory for each chapter, to contain all of the files that pertain to that chapter (not just the text but also figures etc, which I store as separate files because that way they're easier to change). Sometimes I have a different file for each section within a chapter; small files are more manageable, and it's generally easier to jump between two separate files than between two places in a large file. (I know some IDEs have features to make it easier to jump around a program, but I think that keeping a good directory structure is at least as good and is more portable.)

I am using LaTeX as my main language; writing in LaTeX is more like programming than writing in Word; the typesetting stage is a little like compiling. Besides generating good, optimised typesetting that reads better, LaTeX also takes care of all cross-referencing, numbering and indexing, generates the bibliography automatically (with the aid of BibTeX), and virtually forces you to use style sheets (as opposed to Word where you have to discipline yourself to do these things); this means you can make far-reaching global changes to the document without having to worry about its consistency in this respect.

I have recursive Makefiles and I can use "make" to generate a PDF of any chapter or of the whole work. Some sections are written in LyX (a front-end that uses LaTeX) but these are exported to LaTeX before being included in the rest; this is a little like including a small amount of "visual" programming in a larger project (but it's not quite so awkward). There are other languages involved as well; the figures are typeset in various different programs (such as Lout, music programs, a graph visualiser, and a ray tracer) under the control of Python scripts, and at least one section is generated by applying a stylesheet transformation to a database.

At one point I had the thesis Makefiles depending on the prototype of the system I was describing; after all, if I fix some bug in that program, then this means that the test cases should be re-run, and since the output of some of these is included as figures in the thesis, then the thesis has to be re-typeset as well. One nice thing about properly-constructed Makefiles is that they won't let you forget any necessary commands to ensure that all your files are up-to-date with each other, but at the same time they can let you try out a small change without needlessly re-processing what hasn't changed. (Make has some nice documentation in the GNU "info" pages, accessible via the "info" command or from within Emacs.)

My method of working also feels more like software development than writing. I find bugs and fix them. I do "systems integration" between different sections. I have releases and revisions. I use the development features of XEmacs to navigate around my source (apart from the LyX files), and I work on whichever part I fancy without unnecessary resistance from the physical structure of the final document. I use a private Wiki to keep track of my notes, ideas and documentation that is separate from the work in progress. And so on. I'm always at least one step away from the final output; I somehow feel safer behind a compiler because I know that this will do at least some amount of sorting out for me.

Perhaps there is something about large creative projects of any kind (programming or otherwise) that lends itself to the methods we use for software development. Or perhaps I'm a programmer gone mad. At any rate, I think this shows that software development and technical writing projects do have some things to lend to each other.

I must stop now and write some thesis. I wonder if any members who are reading this would like to write another article that takes the idea further.

Silas S Brown

Student Code Critique 20

The problem this time is to help this student within the terms he specifies. However you should take the opportunity to correct errors and misconceptions.

Solutions by ***March 1st*** *(sorry about the tight deadline but the next issue of C Vu must be ready for distribution during the ACCU Spring Conference.*

I am trying to write a function that dynamically allocates an array of integers. The function should accept an integer argument indicating the number of elements to allocate. The function should perform necessary error-checking to determine if the memory was successfully allocated. if the memory was allocated the function should return a pointer to it. Otherwise it should return a null pointer.

This is 'Homework' so I do not wish to have the program written for me. My problem at this point is that I am having trouble coming up with the concepts in my head. Any help would be appreciated. This is what I have so far.

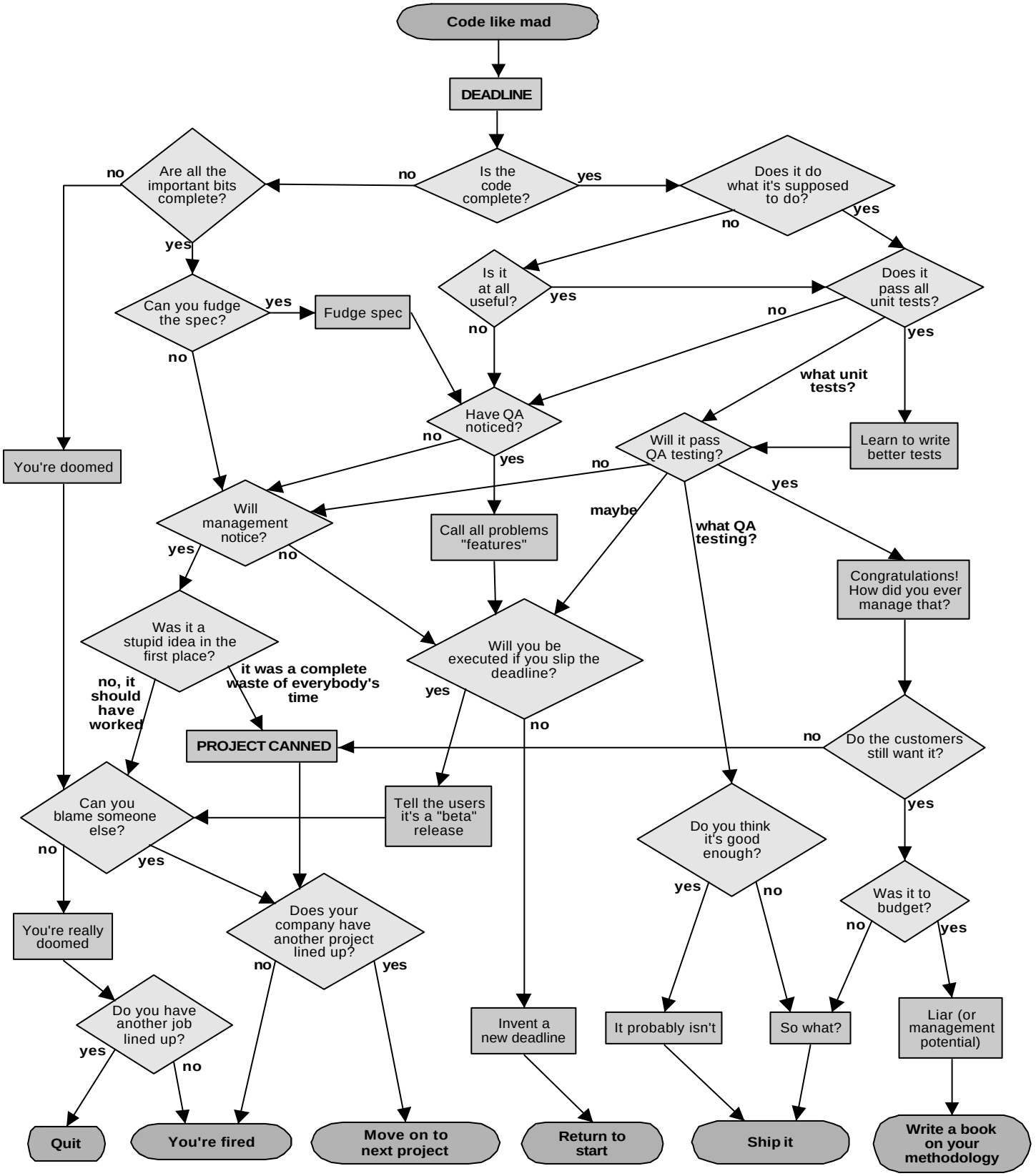
```
#include <iostream>
using namespace std;

//function prototypes
int *allocatemem(int);
int *sortNums(int);

void main(void) {
  int NoGrades;
  int *grades;
  int *test;
  cout << 'Number of grades to enter: ';
  cin >> NoGrades;
  grades = allocatemem(NoGrades);
  for(int i=0; i<NoGrades; i++); {
    cout<<'What is test score # '
        << (i+1) <<' ?';
    cin>> *(test +i);
    if(*(test +i) < 0) {
      cout << 'Must be positive \n';
      cout << 'Please enter Test #'
           << (i+1) <<' correctly: \n';
      i–;
    }
  }
  sortNums(NoGrades);
}
int allocatemem (int amount) {
  int *memory;
  memory = new int[amount];
  if(memory != 0) {
    cout << 'We have memory \n';
    cout << 'going back to main \n';
    return memory;
  }
  cout << 'We do not have enough memory for'
          'this task \n';
    cout << 'going back to main \n';
    return memory;
}
int sortNums(int scores) {
  for(int j=0; int temp=0; i<numberOfScores; j++)
    temp= *(test +1);
    if(*(test + 1) < * test +1 + 1) {
      *(test +1)= *(test +i +1)
      *(test + i + 1) = temp
    }
  for(int a = 0; a<numberOFScores; a++)
    cout<<*(test + 1)<<' ';
}
```

Professionalism in Programming #18

Engineering a release

by Pete Goodliffe <pete@cthree.org>

Relentless software lifecycle problems getting you down? Consult this handy reference to make your software development life simpler.



An Introduction to Optimising Programs

Roger Orr

Writing programs is hard, and writing programs that are fast is harder still. In my experience there are too few overviews of the optimising process, The overviews there are tend to be fairly specific to a target language and/or platform.

I hope this article will provide some general guidelines for programmers who are faced with a program that is 'too slow'. I'll look at trying to decide how much optimising is needed, some of the alternative techniques available, measuring the improvement and briefly talk about a common measure of algorithm speed.

Why do you need to optimise?

This should be the first question you ask yourself when you have a program which seems to be too slow. It helps you to focus on the motivation for making the program faster.

In any programming task there are a large number of possible goals; common ones, other than performance, include:

- Reliability
- Short development time
- Maintainability
- Rich in features
- Memory footprint
- Portability

As the old joke about specifying software goals puts it: "Fast, on-time, reliable and feature-rich. Pick any one." The joke reflects the natures of trade-offs in computing. So don't forget that the effort spent optimising a program is likely to reduce the success in reaching other goals you may have. Typically the optimisation process produces more obscure code, which may be both harder to modify and more likely to contain bugs.

But programmers the world over seem convinced, often against the evidence, that they can and should spend time optimising programs without justifying this decision.

Sometimes though you don't need to make the program faster - it is simply a matter of user perception! People complain, reasonably enough, that programs are slow when they have to wait for them. There are at least three things you may be able to do to improve this.

Firstly, you may be able to make the slow part happen asynchronously, perhaps by executing it as a background process, which allows the user to carry on working while the activity completes.

Secondly, doing the tasks in a different order may help. For example, moving the slowest part to the end of the chain may mean the user can leave the program to complete while they get on with another activity.

Thirdly, giving the user an indication of progress. As anyone who has waited for a train late at night knows, doing nothing makes time drag. If the user gets an indication of progress (percentage complete and/or time to finish) the delay is more manageable.

When you do need to optimise a program you need to decide on the priority of performance compared with other goals for the program. Setting this priority can be hard. Sometimes targets for performance are provided up front when the initial specification is written, but I find that usually performance is not specified at all - it is just assumed that the program will be 'fast enough'. (Even when specified the actual phrasing is sometimes subject to interpretation. As an example, suppose a specification states "the communications module must be able to process an average of 35 incoming requests per second". In a system where message lengths could range from 32 to 8192 bytes there could be a lot of arguing over whether or not the specification had been met!)

If you are trying to set the priority you must use your judgement. In some cases performance is vital - if you're too slow you fail. This is typically true for many parts of 'real time' programs where failing to keep up with incoming events can produce catastrophic failure - or if an interactive game is perceived as 'too slow' the users may simply reject it.

When the situation is less clear cut a common approach is to let the users set the priority - you might say 'I have one week of development time, what would you like me to concentrate on?' Their top requirement might be fixing a bug, adding a feature, etc. rather than making it faster.

An approximate measurement that can help is the combined performance of both development and deployment. What do I mean by this?

Optimising code costs time. How much time will be saved by the speed up? A rough calculation of this figure can help quantify how much time it is worth spending making the program faster.

Based on the expected use of the program I can estimate how much time would be saved (and for whom) if I optimised it. Then I can balance this time saving against the time lost by optimising - include both the development time and any effect on the completion date.

Spending half a day optimising an end of month report, run by one person, to take four minutes rather than five is likely to be a net waste of time!

Investigate alternative ways to achieve performance gains

We'll assume you have quantified your need for increased performance. Before you simply start changing existing code think about the problem and see whether another approach might be productive.

Firstly, perhaps just improving the platform! Check the machine has enough memory. I once had a program which ran several hundred times slower when moved from a 50MHz 486 machine to a 90MHz Pentium one; simply because the Pentium machine had less memory and spent most of its time swapping data between main memory and the hard disk. "Moore's Law" states that chip density, and hence performance, doubles every 18 months. It is really an observation rather than a law, but it has been true for a while and looks likely to remain so for some years to come. So upgrading your hardware (or adding more memory) might be a good start. There's even an equivalent software 'law' - "Proebsting's Law" [1] - which states that compiler optimising technology doubles every 18 years (!) So getting a newer compiler, and checking optimisation is enabled, may help too.

Then there could be lots of different ways to change the way the program runs to reduce the impact of it being too slow. For example:

- Share the output of one report among a number of users
- Look at the task which is causing the complaints - perhaps you can restructure the business process to remove or simplify this task, or to batch up several tasks into one.
- Sometimes many users are doing very similar analysis of the same data in the same way. Perhaps some analysis, or even partial reports, could be generated and saved beforehand.
- Specialised hardware could be used if you're doing a lot of intensive numeric computations.
- Depending on the languages you are using in the project perhaps the slow part of the program could be re-written in a faster language.
- Split the task between several machines. You might be able to make use of another machine to run a slow running report offline, or overnight.
- Make the program multi-threaded. Be careful though - making a program multi-threaded late in the development cycle can cause more problems than it solves.
- Enable unattended operation, perhaps by allowing data to be provided in a file as well as interactively, so the user can start something running and then go off and do other tasks.

Be creative! Sometimes you can find a way to drastically improve the effective performance of the overall process without making big changes to the internals of the program.

Measure, measure, measure!

Once you've decided you need to make your code execute faster it is vital to take some measurements. Do not rely on 'gut feelings', since even for experienced programmers these are notoriously wrong. You can all too easily waste effort in optimising code only to find it doesn't really impact the overall performance. The "80/20" rule applies here - over 80% of the processing occurs in less than 20% of the code (and it is often even more extreme than this). Your first task is to identify this 20% and your second task is to make it faster. Both tasks may be harder than they look.

Before you start you need to decide two things.

Firstly define the actual task you are trying to optimise - this includes the sizes of data sets, the target hardware, etc.

It may be worth creating a test environment, possibly including additional programs that can drive the one under test. For example, when trying to measure the performance of a Web server it is common to use a load generator to simulate a lot of users accessing your pages. The hard part of any test environment is trying to ensure that the test data and load is as similar as possible to the real one. In some cases this can be achieved by capturing data from the live system and replaying it for testing purposes. It is a good idea, where you can, to have several different data sets or you

may end up accidentally optimising your code to match specific features of your test environment not often occurring 'in the wild'.

Remember one golden rule, to optimise the actual code you are going to deploy. Many environments follow the debug/release build model where most of the development cycle is performed on a 'debug' build (with debugging symbols, extra trace statements and assert statements) but the actual released code is built differently (optimised code, trace statements optimised away, etc). This is not the article to talk about the pros and cons of this common practice, just be aware that you will probably waste some (or even all) of your time if you optimise the debug build and ship the release build!

Secondly you must decide how big an improvement you need. This measurement provides you with a target so you know when your optimisation task has finished. The so called 'Speed-up' theorem from the theory of computability states, among other things, that for nearly all functions there is no 'best' program - you can always find another algorithm which is faster. So without a target to aim for you may find you never stop optimising!

Now you are ready to identify which parts of the task actually take the time.

There are a number of techniques to measure this. The most comprehensive is to use a tool designed to provide performance measures, such as Rational's Quantify or Compuware's TrueTime. These tools are powerful and when all works well they provide you with various ways to view the collected data.

The most useful report for optimisation purposes is probably the 'top 10' of functions by execution time. This gives you a hit list to start with. Keep the data from this initial analysis for comparison purposes against the optimisations you make.

The big advantage of the tools is that they are written by experienced people who avoid the large number of traps for the unwary. They are designed to be easy to use but also contain powerful features which can help in particular cases. They are often expensive - but don't forget so is your time!

However sometimes you can't use tools like these. They may not be available for your target platform, or cost too much, or their impact on the program being measured may be excessive. I would still strongly encourage you to try and measure before you optimise. Depending on the size of the job this could include:

a) **Writing a profiler for your chosen platform.** If you want to do this, just lie down till the feeling goes away! This is usually far too costly, but it might be worthwhile for some sorts of embedded systems development.

b) **Modifying the code to calculate elapsed time at critical points.** Many operating environments provide ways of getting the time with a high degree of precision, although this is typically not very portable (for example, Microsoft Windows has the QueryPerformanceCounter API). The easiest way to calculate elapsed time is to obtain the real time clock value before and after some action and print the difference. One common trap to avoid is measuring the data collection code itself. Suppose you instrument the call to a function `a()`, which in turn instruments the calls it makes to functions `b()` and `c()`. A naïve approach to measuring elapsed time would include the cost of instrumenting the calls to `b()` and `c()` in the overall total for `a()`. This can seriously skew the results (and is example of a good reason to prefer using a 3rd party tool!)

c) **Counting the number of times each function is called.** Some compilers provide a switch which calls a user-written 'hook' for procedure entry/exit. Another approach is to make use (in C/C++ anyway) of procedure entry macros - typically turned on by a pre-processor setting. A simple syntax parser and/or coding standard can even automate the insertion of these into your source file. One well-known example is Fred Fish's debug library.[2]

d) **Sampling the program counter.** If you interrupt the process periodically and see where the program counter is that can provide a statistical measurement of the execution profile. In an ideal world the interruptions would be truly random and the less this is achieved the less accurate the results may be. However, this is often the simplest measurement to take - use a debugger and just 'break' the program a number of times when it is busy to see where it is!

e) **Make a function slower.** One less well-known approach which can be of benefit in measuring speed is, paradoxically, to make things slower. Let me explain. Suppose you suspect one routine is the bottleneck in the processing but you cannot easily measure this directly. Simply modify the code to call this routine twice, and measure the resultant slowdown in program execution. To a first approximation if the program runs 10% slower the un-modified routine takes 10% of the execution time. This can be useful in deciding whether a given optimisation is worth attempting.

Using one of these mechanisms you assemble a short list of 'hot spots' in your program. You are now ready to start looking at some code, and deciding how the slowest functions could be improved. There are many possible direction this can take!

Perhaps you can reduce the number of times 'high scoring' functions are called, by caching common values. You might find the disk is a bottleneck and look at changing the data format, or the hardware. In a distributed world the network is often a bottleneck and you may be able to 'batch up' packets to produce a smaller number of bigger messages.

One common result is the whole program is limited solely by CPU speed, so I'll now focus on processor bound functions.

The 'complexity' notation

Many times programs work fast enough on small data sets and too slowly on big ones. This is often because the algorithms picked do not scale well from small to large data sets. Complexity notation describes the approximate way the time taken for the function changes with an increasing number of items, and knowing this can help you to pick the best tool for the job.

You may have come across this notation already - for example the C++ standard defines the complexity of operations for vectors, maps, etc. There is a more formal notation, called the 'Big-Oh order notation', for measuring complexity but this is outside the scope this article.

The complexity notation usually given is the 'worst case' figure. For example sorting is typically much faster when the input is already partly sorted data but the complexity measure is for randomly ordered data.

'Constant time' means the function is approximately as fast for a small or large number of data points. An example of a constant time operation is accessing an element of an array (in most programming languages) - it doesn't really matter how big the array is since it is simply a matter of calculating an offset.

'Linear time' means that the time for the function is roughly proportional to the number of points. An example of a linear time algorithm is finding a random element in a linked list since on average half the points must be tested.

There are other complexity values but both of these scale fairly intuitively - people expect programs to run more slowly if they're working on more pieces of data. If you double the number of items the program can run anything up to twice as slowly without serious complaints.

The real problem is functions that increase in time more than linearly. One common example is 'quadratic time' for functions where the time increases in proportion to the square of the number of items. As the size of the data increases eventually these functions 'take over' the performance of the overall program.

Suppose you have used several linear time functions and one quadratic time function in your program. Perhaps the quadratic function is only 1 millisecond out of an overall time of 100 milliseconds when you have only 10 data points. Consider what happens when you have 10 times as many data points. The quadratic function increases to about 100 milliseconds and the rest of the program increases to about 1 second. So the single quadratic function now takes 10% of the overall program time. What if you have 10,000 data points? This one function will dwarf the contribution of the rest of the program.

No matter how fast the code is, with enough data points the higher complexity functions will eventually 'take over' the performance of your whole program.

This is why it is so important, as I said earlier, to measure performance of the program as closely as possible to the way it is used in practice - including the number of data points - because otherwise you might fail to detect such functions.

What can you do when you find a function that is of too high an order? Try to search for another algorithm that is of lower order - such as linear time. This is likely to have much more impact on the performance of your program than if you leave the algorithm unchanged and simply try to optimise the individual lines of code.

The order of many standard algorithms, such as sorting, and operations on common data structures - such as arrays, maps and lists - is well documented and it should be relatively easy to see whether a better algorithm exists.

Armed with this information you can often transform higher order functions into something better.

[concluded at foot of page 18]

Eclipse, A New Integrated Development Environment

Silvia de Beer <silvia@juggler.net>

The slogan of Eclipse is "Eclipse, an IDE for anything, and for nothing in particular". You can use the IDE to develop programs in any language. However, the strongest point of Eclipse is to use it for Java development.

Eclipse starts its history as a product from Object Technology International. IBM takes over Object Technology International, and donates the source code in November 2001 to the `eclipse.org` open-source consortium. IBM continues to base its WebSphere IDE on Eclipse, and is phasing out VisualAge for Java.

The architecture of Eclipse is based on a platform runtime. All functionality of Eclipse is added in the form of plug-ins. When you download the basic Eclipse installation, it comes together with the plug-ins for Java development. For C or C++ you have to download an extra plug-in, the C/C++ Development Tools, (CDT). The first release of the CDT was on the 11th November 2002. The CDT is using the GNU tools at the moment: make files, gcc and its debugger is based on gdb.

Eclipse is completely written in Java, and does not use Swing for the graphical parts. Eclipse has developed its own graphical libraries, SWT and JFace. SWT/Jface is analogous to AWT/Swing, but is different in that it uses a rich set of native widgets. The portable SWT/JFace API is implemented on different platforms using a combination of Java code and JNI natives specific to each platform. For this reason, the implementation always follows the native look and feel, and is relatively fast.

The Eclipse workbench consists of an editor and a number of views. The workbench introduces the concept of perspectives to perform different tasks. There is the plain resource perspective, which offers file browsing and editing. The Java Development perspective offers class hierarchy browsing. The debug perspective offers all tools for debugging your code. The CVS perspective allows access to your repository. In the first week you might be a bit overwhelmed by the many possibilities to configure your IDE, but you quickly get used to it. For Java development I experienced a productivity increase by using Eclipse, compared to just using a plain editor and command line tools for compilation. The productivity increase is mainly due to the automatic code completion and re-factoring support. By using the re-factoring support, there are no more tedious find and replaces, but a simple indication that you want to change a name, want to move a function elsewhere, and all is automatically taken care of, with a preview of the changes which are going to be made. Eclipse Java development compiles by default every time you save, so you immediately discover whether you break something when you edit your code. JTest is integrated, so there is no reason anymore why you should not develop your unit tests when you are writing your code. The editor offers handy features like automatic import statement organisation, generation of various parts of your code, e.g. class bodies, getter and setter functions, overriding methods, addition of constructors from the parent class, automatic code formatting.

The idea of offering the Eclipse platform as an open source project is that tool providers can easily offer plug-ins, either as an open source product, or as a commercial product. For example, a plug-in could be offered for UML diagram editing, integrated with automatic code generation. Rational Rose already offers a commercial plug-in for Eclipse, but if Eclipse becomes a success I am sure that other open source plug-ins will become available. There is already a number of plug-ins, but sometimes the functionality is still limited. I investigated XML editor plug-ins, but found none of them really satisfying.

The Eclipse framework is set up in a way that developing a plug-in is relatively easy. The framework defines a number of extension points, and documents the classes that can be sub-classed. By developing plug-ins, developers can add specific functionality to Eclipse. This can range from a few extra buttons and menu items, to full blown graphical editors, editors for other programming languages or support for writing embedded programs. Deployment of a plug-in simply consists of unzipping the zip file in the plugins directory.

For those who want to find out more: `http://www.eclipse.org`

Silvia de Beer

[continued from page 17]

As a simple example, take this pseudo-code:

```
int N = collection.size();
for(int index = 0; index < N; index++) {
  element e = collection.elementAt(index);
  ...
}
```

How is `elementAt` implemented? If the collection is implemented as a linked list then typically `elementAt` will be linear time and so the whole code fragment is quadratic.

What can you do? Many things - but complexity notation can help you analyse them quickly.

As an example you might change the collection from a list into an array. Then the whole iteration process becomes linear time - but you need to investigate the effect this might have on inserting items into the collection in the first place.

What next?

In many cases analysis like that shown above will help you to identify one or two particularly slow functions which can be changed to use a more efficient algorithm.

Modify, or rewrite, the slow functions. Test the changed functions and then measure the speed of the new code to see whether you have made the improvement you hope for. If the change has not improved the speed simply back it out (you are using source code control software aren't you?) and re-examine both the performance data and your analysis of it.

This is a particular example of refactoring, where the function is being changed solely for performance. A couple of the general principles of refactoring are particularly relevant here.[3]

Firstly, it is almost always a good idea to ensure you keep this refactoring separate from any other changes you may wish to make to the code; this will help to isolate the changes solely made for performance and ensure you are comparing like with like when measuring the effect on program speed.

Secondly, ensure you have a test harness or test code to verify that the new function has the same results as the old one. Rewriting a working, but slow, function into a blindingly fast one that gives the wrong answer is not usually useful!

One way to achieve the testing is to initially leave in the old function and execute both it and the new function. Then add a verification check to compare the outputs and ensure that your optimised function does in fact produce the same answer as the original code. Then switch off the old code and the verification check just before measuring the performance improvement. I can't recall how many bugs this simple technique has saved me from over the years!

Summary

This introduction to optimising has stressed the importance of measuring things. It is important to try and quantify how slow the current program is and what magnitude of improvement is required. This provides a sensible basis for prioritising any work done on optimising. During optimisation measuring the speed of the components of the program under test enables you to focus your effort where it will be most effective - and also lets you quantify the improvement you make.

Finally the apocryphal 'Jackson's rules of program optimisation' state:

Rule 1. Don't.

Rule 2. If you must, don't do it now.

So to comply with this rule, I hope to cover more of the specifics of optimising algorithms and improving existing code in a future article!

(Many thanks to Mark Green and Richard Bridgman who reviewed drafts of this article!)

Roger Orr

References

[1] On Proebsting's Law - `http://www.cs.virginia.edu/~jks6b/on_proebstings_law.pdf`

[2] Fred Fish's debug library - `http://sourceforge.net/projects/dbug/`

[3] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley.

It Could Happen to You

Brett Fishburne

So, I was asked to develop a very simple application that really shouldn't take any time at all.

"You know how you walk into a deli and you get a number?" asked the user.

"Yes," I replied.

"I want a program that gives me such a number."

"That's all?"

"Yes."

Confident that I could handle such a simple assignment, I wrote a counter. It read a file to get the current number, incremented that number by one, wrote the new number to the file (overwriting) and spit out the new number. I initialized the counter to 0. Thus began a ridiculous two-year odyssey around a little program that "...really shouldn't take any time at all."

It turns out that the user had not considered that more than one person might go for the number at the same time, so I modified the program with a semaphore to prevent two people from getting the same number.

It also became quickly clear that the program had to start with the number 1923. Correction duly made.

Then I was told that the numbers are actually associated with months in the year. There were only 100 numbers per month. In addition to returning the number, I needed to return the month and year to which the number corresponded. Now the program got more complex, but I knew that each month began with "00," so this became the month delineator. As I rolled this modification out, it turned out that sometimes someone wanted to request the next number available in a given month and year.

Now the original data file (which had only one number) had to have a number for each month and year combination and I had to come up with some sort of marker for indicating that no numbers had come from a given month and year. For my own personal reasons, I picked "1555" as such an indicator. The input file was slightly more complex now and the data therein had to be read into an array (month and year indexes, of course), then the proper number was chosen. The logic got more complicated as users incremented outside of a given month/year combination as well.

Suddenly users were making mistakes and wanted a warning (like DOS gives) to make sure that the user really wanted the number. In addition, sometimes the users wanted to know what the next available number was without actually taking a number. More coding.

Now users wanted to "reserve" a set of numbers, which is a quasi-form of taking numbers out. The file got a little more complicated...space was made for up to four ranges of "reservations" per month. This drastically complicated the underlying data structure (hey, arrays don't work any more). A dramatic rewrite was necessary due to the dependence on arrays. In addition, the logic was much more complicated, now users could request a reservation that spanned months; what happened when two overlapping ranges were requested; how were multiple ranges linked together so that the four possible ranges would be sufficient.

Just as the range functions became available, it was necessary to implement an "automated" mode that didn't print out erroneous things (like the DOS double-check question). This led to more rewrites and global variables to track state.

Suddenly, the old rule of 100 numbers per month was changed to 150 numbers per month! I was told that this was a malleable parameter now and that this could change any number of times. Horrific code rewrite! Remember the presumption that "00" ended a month? Well, it didn't always any more! Now a new data structure was required which "remembered" the amount of numbers in a month and which had to be queried to determine how many numbers there should be in any particular month. Virtually every function was affected.

The "simple" program was now 323 lines of very convoluted PERL. ALL of the original data structures have been replaced. The program remains fragile in another way (I was told that handling numbers under 10,000 was fine, but it isn't - we're almost to 7,000 now and anticipate hitting 10,000 around December) and probably many more of which I am not aware (the four range limit has never been truly exercised, the largest number of ranges ever requested is two in a given month, but this is certainly another area to consider). On the other hand, this program is used by dozens of end-users, every single day. It is often hailed as the "glue" which holds our effort together and work literally stops when it fails (less and less often).

Lessons Learned

1. Most users don't know everything they want right away. Iterating with a program they can use in its natural setting helps them define what is missing.
2. It is critical to insulate logic from data structures even on "simple" programs.
3. Many users are used to being treated like idiots by their operating system (Windows) and expect the same treatment (i.e. find it comforting) from all applications.
4. Assumptions (even assumptions you have been told are valid) kill you. Most programmers in the United States make the following erroneous assumptions about Social Security Number (a government issued identification number):

a. That it can only be numeric (wrong!)

b. That it is unique (wrong! The numbers are recycled after death)

c. That it will stay the same size (probably wrong)

5. Code that functions quickly and easily will get incorporated into many other efforts expanding both the original functionality and the need for the program to behave in expected ways in multiple environments.

Brett Fishburne

Write for ACCU!

If you would not be forgotten,
As soon as you are dead and rotten,
Either write things worth reading,
Or do things worth the writing.

Benjamin Franklin

What to write?

Here is a small selection of suggested titles. These have been *specifically* asked for by ACCU members. Please look at the list and consider if you can write something on a topic.

- **The preprocessor**
  What does it do? What can I do with it?
- **Working with strings**
  How do they differ in C and C++?
- **Which loop?**
  How do I choose between `for`, `while`, and friends?

Don't let this list constrain what you write! What are you doing right now? What do you know about? Please write something about this for the ACCU journals.

How to submit

You can send submissions by email to `editor@accu.org`. Plain text is perfectly acceptable; there is a Word document template you may wish to use if you want to retain formatting. That's all there is to it – *please write something.*

Pete Goodliffe

Python Section

ACCU Mentored Developers Python Project: Practical Uses for Python in C++ Series: Part 1

Generating Strings for C++ in Python

Paul Grenyer

In this article I explore and provide examples of using Python[1] to generate constant strings for C++. There are of course lots of ways of defining and using constant strings as well as other variables, but for the purpose of this article I am going to stick with constant strings similar to example below:

```
const std::string
        myConstantString("A Constant String");
```

Why do I need to use Python to generate something like this, I hear you ask? Well, the simple answer is that you don't. Python becomes useful when you have much larger strings to generate, such as software license agreements (e.g. GNU General Public License[2]). These are often displayed in application about boxes and therefore need to be transformed from the flat text file that the customer has supplied the license to you in (or that you have created from the Microsoft Word Document they supplied), into a form that can easily be pasted into something like an edit box.

To create a string such as the one in the example above from the GNU General Public License there are three things that must be done to each line:

1. Each line must be enclosed in its own pair of inverted commas. For example:

   ```
   "GNU GENERAL PUBLIC LICENSE\n"
   "Version 2, June 1991\n"
   ```

2. All inverted commas embedded in the line must be changed to the correct control character. For example:

   ```
   "Each licensee is addressed as "you".\n"
   ```
   becomes
   ```
   "Each licensee is addressed as \"you\".\n"
   ```

3. The original new line character must be replaced with the correct new line character for the platform. For example on Windows:

   ```
   "GNU GENERAL PUBLIC LICENSE\n\r"
   "Version 2, June 1991\n\r"
   ```

It is also a good idea to have a template C++ header file with a *specific comment*, at the point where the license needs to go, that can be detected by the Python script. This means that if at any time you want to change the C++, you don't have to fiddle with the Python script, just the template. For example:

```
// License.h
#ifndef LICENSE_H
#define LICENSE_H
#include <string>

namespace MyApp {
  namespace Licenses {
    const std::string gnuGPL(
// License Here
); // gnuGPL
  } // Licenses
} // MyApp

#endif // LICENSE_H
```

We are now ready to write a Python script. Assuming that we have the license in a flat text file called '`gnugpl.txt`' and the C++ template in a file called '`LicenseTemplate.h`', the first thing we want the script to do is take the two input files and an output file as command line arguments:

```
import sys
import string

if len(sys.argv) < 4:
  print "License.py requires three filenames "
  "as command line parameters: "
  "License, Template, OutputFile"
else:
  print "License Filename: " + sys.argv[1]
  print "Template Filename: " + sys.argv[2]
  print "Output Filename: " + sys.argv[3]
```

Now that we've successfully identified each file we need to load the two input files into a container a line at a time. There is no need to check that the files are opened successfully as Python will give an error and abort the script if it can't open the files.

```
LicenseFile = open(sys.argv[1],"r")
LicenseStore = LicenseFile.readlines()
LicenseFile.close

TemplateFile = open(sys.argv[2],"r")
TemplateStore = TemplateFile.readlines()
TemplateFile.close
```

Next we want to write the C++ output file and identify the *specific comment*. The *specific comment* should then be replaced with each, correctly formatted line of the license.

```
OutputFile = open(sys.argv[3],"w")

for templateline in TemplateStore:
  if templateline == "// License Here\n":
    for line in LicenseStore :
      # Change " to \"
      line = string.replace(line,'"','\\" ')
      # Write License line enclosed in " and
      # with Windows new line characters
      OutputFile.write('\t\t"' + \
        line[:len(line)-1] + '\\r\\n"\n\n')
    else:
      OutputFile.write(templateline)

OutputFile.close()
```

This completes the Python Script. To create the C++ header file (`License.h`) containing the license const string, simply run the Python script with the required command line arguments:

```
license.py gnugpl.txt LicenseTemplate.h
                                    License.h
```

The full `License.py` script and a Microsoft Visual C++ 6.0 test application can be downloaded from my website[3].

Paul Grenyer

References

[1] Python: `http:\\www.python.org`

[2] GNU General Public License:
`http://www.gnu.org/copyleft/gpl.html`

[3] Full `License.py` script and a Microsoft Visual C++ 6.0 test application: `http://www.paulgrenyer.co.uk/download/python_license.zip`

Reviews

Bookcase

Collated by Michael Minihane
<michaelm@pobox.co.uk>

Francis Glassborow writes:

I guess our Production Editor may be chopping out the bodies of quite a few reviews. Remember that the complete versions are always available on the web. Please also remember to write an alternative review if you strongly disagree with the one we publish. That is essential in the interests of fair play.

I need volunteers to review C and C++ books for novices. Until now I have covered most of those myself but as I am well on the way to getting my own book published I consider it inappropriate that I should continue to handle these reviews, Particularly as almost all the books aimed at this market are, in my opinion, badly flawed. They are usually written by authors who know too little about the programming language they are using. Sometimes they are written by authors who have little idea as to the needs of their readership. If you feel able to be confidently critical of these books please volunteer to review such books.

If you want to join our reviewers, getting started is very simple, just go to our website, navigate to the mailing lists section and then subscribe to `accu-books`. You will then get periodic updates as to what books are available for review (about 250 at the moment). We make a small (£5) admin charge per book (which is credited to you if you get a real lemon). You normally have about two to three months to complete the review, after which the book becomes yours to do with as you wish.

Talking about reviewing, reminds me that I was recently browsing through Amazon.com's website when I came across a list of their top fifty reviewers. I am highly criticized in some quarters for having dared to review about 900 books in the last 12 years. Yes that is a pretty substantial number of books. But can someone tell me how anyone can review over 4000 books for Amazon.com. That is about a decade's reading for me and most people think I am a pretty prolific reader.

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`
The PC Bookshop (020 7831 0022)
`orders@pcbooks.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 1.4 or better as appropriate (in a context where the true rate hovers around 1.5). I consider any rate below 1.3 as being sufficiently poor to merit complaint to the publisher.

C & C++

C++ Templates: The Complete Guide by David Vandevoorde & Nicolai Josuttis (0 201 73484 2), Addison-Wesley, 552pp @ $54.99 (UK price unknown)
reviewed by Josh Walker

Unless you have been hiding under a very large pile of code lately, you will have noticed that templates are quite in vogue in C++ at the moment. Some of this is due to hype, but mostly it is justified because templates are an extremely powerful and genuinely useful tool. As a result, there are several recent books that make sophisticated use of template techniques, e.g., *Modern C++ Design* and *Generative Programming*. Furthermore, any book on the STL will clearly touch on templates. And even general books such as *More Exceptional C++* can't avoid mentioning templates. So why do we need another book on templates? Because despite all of the above, there is a gaping hole in the literature, which the authors of *C++ Templates* describe in Chapter 1:

> Yet we have found that most existing books and articles are at best superficial in their treatment of the theory and application of C++ templates. Even those few books that do an excellent job of surveying various template-based techniques fail to describe accurately how these techniques are supported by the language. As a result, beginning and advanced C++ programmers alike are finding themselves wrestling with templates, attempting to decide why their code is handled unexpectedly. *This observation was one of the main motivations for us to write this book.*

This book succeeds phenomenally at filling the hole.

You can rest assured that the authors know what they're talking about. Nicolai Josuttis should already be well known to you as the author of *The C++ Standard Library*, another must-have book. David Vandevoorde may be less familiar. He is an engineer at EDG, which produces a C++ compiler front end widely acknowledged to be the most standards-conforming available. He was also the main implementor of EDG's support for exported templates, the only implementation of this (controversial) feature on the market. In *C++ Templates*, Josuttis and Vandevoorde live up to their excellent reputations by presenting accurate and useful information in a clear, easy-to-read manner.

Let me list some of the high points so you can see why this book is so useful. Part I of IV is titled *The Basics*. It begins with a brisk tutorial/review of function templates in Chapter 2, and class templates in Chapter 3. This material is probably familiar, but reading these chapters will clear up any lingering confusion you may have. Already after Chapter 3 we enter territory where many C++ programmers will struggle. The remaining chapters in Part I are *Nontype Template Parameters* (Chapter 4), *Tricky Basics* (Chapter 5), which among other things discusses using the keywords `typename` and `template` for dependent names, *Using Templates in Practice* (Chapter 6), which covers compilation models and debugging techniques, and *Basic Template Terminology* (Chapter 7).

Part II is *Templates in Depth*. All but the most battle-hardened template coder will learn something here. With topics such as name lookup, SFINAE, points of instantiation, argument deduction ... this part alone is worth the price of the entire book. Part III, *Templates in Design* and Part IV, *Advanced Applications* both discuss more concrete details of applying templates to real problems. These also contain a wealth of sample code.

While the information in this book will vastly improve the quality of your template code, another equally valuable contribution is specifying the vocabulary with which we talk about templates. Clarifying the conventional meaning of terms such as traits, policies, template arguments, template parameters, etc. will aid in more precise communication between programmers. As an example, I have already seen the term SFINAE, which the authors have coined, popping up in newsgroups and mailing lists. SFINAE stands for Substitution Failure Is Not An Error, and is a principle that enables remarkable compile-time techniques such as checking for the existence of a particular method in a class.

Now, to keep this review from being entirely one-sided, here are my three complaints. During my review, I was able to contribute several typos to the online errata; however, these are all minor annoyances that shouldn't cause real confusion. Second, I felt that Chapter 13 on *Future Directions*, which discusses possible additions and changes to templates in C++0x would have been better left as an appendix. Although quite interesting and worth reading, this chapter is necessarily more speculative and may become dated relatively soon. Finally, my biggest complaint is that the **`SArray`** (simple array) template class presented at the beginning of chapter 18 on expression templates holds some dangers for naïve readers. The copy constructor is not exception-safe; it will leak memory if the contained type's assignment operator throws. Second, the use of assert statements for error checking is inconsistent (assigning two differently-sized arrays will trigger an assertion, but adding them together may silently read past the end of the smaller one). Although these issues are not strictly relevant in the context of expression templates, I felt that this class failed to meet the high standard established by the rest of the book.

But don't let these few complaints dissuade you. Overall this is an excellent book. If you only buy one new C++ book this year, this should be it.

C++ Network Programming vol 2 by Douglas C. Schmidt & Stephen D. Huston (0-201-79525-6), Addison-Wesley, 350pp @ £30-99 (1.29)
reviewed by Francis Glassborow

There are four major places to look for software libraries/components before you resort to writing your own. The first place to look is the C++ Standard Library. The components provided by that are excellent if not perfect and have the advantage of being highly portable. As long as they do what you need with adequate performance (both speed and code footprint) then there is little reason to look elsewhere.

The second place to look is `www.boost.org` where you will find components that are as close to the C++ Standard as you can get without actually being in it. This should surprise no one because they are designed and developed by those responsible for the Standard Library + a good number of other experts including domain experts who ensure that the designs meet real needs and are correct for those needs.

The fourth place to look is at commercial libraries. Unfortunately the quality of those is highly variable because even the best have relatively small design teams coupled with too little oversight from other designers, domain experts and users. There are outstanding exceptions to this represented by such companies as NAG (Numerical Algorithms Group) and their maths libraries that have been under continuous development for more than two decades.

So what is in third place? Open Source software. This category of software ensures that many eyes have looked at it and added their observations. Bugs and design faults get spotted and corrected. There are two main problems with Open Source components. The first is that they tend to be based on the lowest common denominator of a range of available compilers and older idioms. There is nothing wrong in that except that language enthusiasts can often point to ways in which it could have been done better if designed for the latest systems. That also requires that the designs often require that the user learn a somewhat different interface style to that they find in the C++ Standard Library and in the Boost libraries. The second problem is that documentation is often neglected.

C++ Network Programming Volume 1 set out to address the second issue as regards ACE (the Adaptive Communication Environment). This second volume looks at various important aspects of ACE that are featured by its multiple frameworks. Each of the main frameworks is described in terms of its design and the patterns used.

This book helps you understand how ACE is designed to work. However it achieves more than this because it provides excellent working examples of various patterns that are described elsewhere (for example in *Pattern-Oriented Software Development*, which is co-authored by one of the co-authors of this book). That means that as well as being a book about the higher level architectural features of ACE it is also a book showing such high level design in practice.

There are a multitude of things about ACE that irritate me (for example its naming conventions) but that does not detract from the fact that it is a well designed framework library that provides the components needed by those writing middleware for networking. If you have a need to write software in this domain then you should know about ACE and, in order to make best use of it, understand its design. This book will help you achieve that latter objective while educating you about general framework design.

I could envisage using it as one of several textbooks for a course on software design, as well as a text for those who use or wish to use ACE in their professional work. I think it would also provide an excellent resource for those studying the use of patterns in software design. There is a sense in which this volume is the practical that goes alongside the theory presented in *Pattern-Oriented Software Development.*

Highly recommended for those in its target readership and recommended for those studying patterns and framework design.

Special Edition Visual C++.Net (details not provided)
reviewed by Paul Grenyer

Special Edition Visual C++.Net is the latest edition of Kate Gregory's Special Edition Using Visual C++ book. As you would expect it is an update of the last edition of the book, Special Edition Using Visual C++ 6.0, for the Microsoft .Net framework.

Although there are a number of additional chapters, this edition of the book is slightly smaller, in terms of pages, than the previous edition. This is chiefly because a lot of the material from the previous edition has been dropped or abbreviated.

The first additional chapter is chapter 1, '*.Net Background*'. I've read a number of descriptions of the .Net framework and this is one of the better ones. Most importantly it reassures veteran Visual C++ programmers that there is no need to abandon C++ or COM in favour of C#, VB.Net or any of the other .Net languages. These older Visual C++ technologies can be used (for the most part) in exactly the same way with Visual C++.Net.

Chapters 2 to 14 inclusive are the condensed and updated for MFC 7, ATL 7, etc. chapters from the previous edition. These are written in much the same style and use many of the same examples. COM+ has been added to the ATL chapter.

The remaining chapters look at some of the new features that the .Net framework provides, including The Common Language Runtime, Integrating with C#, ADO.Net and most importantly Managed and Unmanaged C++.

The Common Language Runtime chapter gives a good overview of some of the key objects and takes a look at the new .Net Framework types in comparison to the C++ built-in types, including the new String object.

The managed and unmanaged C++ chapter contains a good explanation of what is managed code and what is unmanaged code and how to convert between the two in the same application. Garbage collection is concisely, but well explained and includes a description of when it is likely to be invoked by the framework and what happens to the pointer afterwards. It also explains how to invoke garbage collection explicitly.

Overall not a bad book considering it comes from QUE publishing. There is no in-depth material, but it does give a good overview of what is available from Microsoft Visual C++.Net. It's a long way from ideal for the serious programmer, but does show how to accomplish many common Microsoft Windows programming tasks easily.

GTK+ Programming in C by Syd Logan (0 13 014264 6), Prentice Hall, 830pp @ £35-99 (1.25)
reviewed by Tony Houghton

I was quite surprised at the size of this book, considering that it only covers GTK+ and not also Gnome. Not only that, but despite the cover blurb's, 'Leverage the full power of Gtk+ 1.2, GLIB and GDK', the latter two libraries get little more than a passing mention. I consider omitting full coverage to be the worst fault with this book, because GLib provides most of the types, structures and utility functions that are not concerned explicitly with GUI building and GDK provides low level graphical routines. A good knowledge of GDK is necessary to display graphics and text in ways beyond the widgets provided by GTK+.

The good news is that the coverage of GTK+ is detailed, very well researched and accurate as far as I can judge. However, it isn't just detailed, it's positively verbose and usually gave me the impression that I was wading through waffle before getting to the point. No wonder the book's fat for its scope. This makes it rather awkward as a reference, but also ponderous as a tutorial and the order of presentation of some material and lack of exercises, also work against it as a tutorial.

With the publication date, one might have expected the author to have access to at least a beta of GTK+ 2. Now that version 2 has been released, books on GTK+ 1.x could be seen as obsolete, but the bulk of the API has not changed significantly.

There are plenty of code listings, but unlike the prose, they're brief and to the point. They're well laid out and labelled with line numbers to make them easier to reference from the text. I generally like listings, provided they're done sensibly, as in this case, because it means the book doesn't have to be read in front of a computer. However, errors always seem to creep in and this book is no exception. One listing that was supposed to demonstrate debugging had even had its deliberate bug accidentally fixed.

I would have given GTK+ Programming in C a 'Highly Recommended' award if it covered GLib and GDK and was written more concisely, but I think its flaws are too serious to be overlooked.

Java etc.

Instant Messaging in Java by Iain Shigeoka (1 930110 46 4), Manning, 378pp @ $39.95 (UK price unknown)
reviewed by Emma Willis

Earlier this year I was researching the possibilities

for implementation of a portable java instant messaging system; I had little technological experience, or funding. I soon discovered the joys of the Jabber XML messaging protocols that seemed to fit the bill. There were only 3 books in print on this topic at the time and I surely must be one of the first people to read this, Shigeoka's offering on the subject.

Instant Messaging in Java appears ideal – it is a step-by-step guide to coding your own Jabber server and client application. What it fails to mention is that in 90% of situations, you will never need to write your own Jabber server; Jabber is an open source project with a series of daemon server implementations for a number of different platforms – should you ever need a more robust solution, Jabber's commercial arm are there to help.

Where this book rises above others on the subject is in the clarity with which it communicates the concepts behind the Jabber protocols. XML and Jabber in particular, can be complicated for newcomers by intricate DTDs or namespace definitions. Great use is made of sequence diagrams and XML to successfully remove the mystery of XML and present Jabber as the common-sense framework that it is.

The central part of the book concentrates on the implementation of the client and server; slowly adding more functionality as more Jabber concepts are introduced. Towards the end of the book, if you have been following the code examples, you should have a working Jabber system that supports group protocols, invites, authentication and more.

Instant Messaging in Java was my first step in this area and I required a lot of assistance with my project. In addition to the numerous Jabber community resources, I was grateful for the interactive elements supplied with this book. Manning provide a chat website to post your questions and discuss relevant topics - Iain, the author, was always helpful and prompt in his responses. The site, as you would expect, is also the best place to track book errata and code downloads.

Once the project is complete, Iain introduces the reader to the wider world of Instant Messaging - letting you know of the alternative and complementary IM tools available. I feel however that the book represents more of a personal project than a practical guide for reader implementations. There are so many IM tools out there and a wealth of web-based Jabber resources; in fact, after much research I abandoned Iain's book and explored a number of open-source Java APIs that achieved the same results, with less effort. I think of this book as a good introduction to Jabber, to IM and to a logical framework for building a Jabber server and client but I feel it is limited in its teaching abilities with regards to the implementation code.

Java Development with Ant by Erik Hatcher & Steve Loughran (1 930110 58 8), Manning, 634pp @ £40-50

reviewed by James Gordon

I've been using NetBeans for Java development for a while and have been hearing about `Ant` all that time. I have kept away from it, but when his book came up I jumped at the chance to read up on Ant.

This book is an excellent book on an excellent tool. The book walks through a simple, but complete, Ant script and then launches into every facet of the tool. With so many things you can do it is quite a task, but one that the book succeeds very well at.

The layout is excellent and there is a comprehensive contents and index with an appendix of all the Ant Tasks also at the back.

One thing I like when trying to understand something is example output. I, probably like others, find it easier to step through code where you can see the output at specific steps.

The book is for Ant 1.5 so it is very up to date. £40 is a lot but it is a very good book. It would be an excellent book if it were a bit cheaper.

Java Management Extensions by J Steven Perry (0 596 00245 9), O'Reilly, 300pp @ £24-95 (1.11)

reviewed by Tim Penhey

The Java Management Extensions (JMX) is the Java standard for management of applications. In real terms this means providing an interface to your java applications so that they can be modified and monitored in a standard way.

If you have used JBoss or Tomcat 4.1.x you may have noticed that many of the functions of these applications are handled through managed beans (MBeans). Reading this book has been very beneficial to my understanding of how this all operates.

The book covers the different types of MBeans, what they are, what needs to be done to make a bean managed and how the MBean server operates. The book's examples are all executed against version 1.0 of the reference implementation (which I believe came out between the end of 1999 and mid 2000).

The book gives very clear examples of how to create managed beans using each of the increasingly complicated methods. Many books suffer from lack of relevant sample code, this book is not one of those. If there is something that you want to do with a managed bean then chances are there is an example somewhere in the book that shows how it is done.

The book is easy to read but occasionally repetitive. However it is hard not to be at least a little repetitive when you are implementing the same code again and again only differing in the manner in which a management interface is exposed.

I would recommend this book to any Java programmer wanting to know how to implement managed beans. Once again O'Reilly have given us another excellent Java book.

HAVi by Roger Lea et al. (0 13 060035 0), Prentice Hall, 450pp @ £35-99 (1.25)

reviewed by Jan Vroonhof

This title discusses the Java parts of the HAVi standard for home entertainment devices, based around IEEE1394/Firewire. It begins with an enthusiastic and compelling argument of why HAVi would be a good thing to have and an up-beat story of how it was created. As stated in the foreword, its adoption in the marketplace is still unsure.

Then the book explains how camcorders, videos, etc. interconnect with an overview of the technology involved and the various acronyms used and an explanation on how this is modelled abstractly by HAVi. Then it discusses the classes involved by giving example programs using the relevant methods and explanations of them.

Unfortunately as it progresses it gets weaker. The first chapters have small code fragments with lots of explanatory text. Toward the complete example programs given in the last chapters it is just pages and pages of code with no markup, comments or text. At times it mixes patronising explanations of core Java concepts with basic syntax errors. Much of the back half is taken up with a listing of class declarations, which would have been much more useful with a good index that listed class names.

Very unfortunately the authors chose **not** to discuss the Level2 UI defined in HAVi 1.1. However that part is the only part of HAVi that ended up in digital TV platforms such as MHP.

This book could have been executed a lot better. However, even a weaker book that explains the ideas and concepts of the designers beyond the dry standards is so useful that it is worth having, should you need to implement or program a HAVi based device.

Perl & Python

Writing Perl Modules for CPAN by Sam Tregar (1 59059 018 X), Apress, 288pp @ $34.95 (UK price unknown)

reviewed by Joe Mc Cool

The Comprehensive Perl Archive Network is an extraordinary institution. I know of nothing else quite like it in any other programming language. Larry Wall, the creator of Perl claims that his language makes easy programming tasks simple and the difficult tasks possible. It is the availability of freely downloadable modules from CPAN that contributes the most to these objectives.

My understanding of a Perl module parallels that of a C library. Just like a library file, a Perl module is made up of shrink-wrapped code that I can call from my own project. Once I know how it works, I needn't concern myself with its internals. Just like a black box, I can pass it parameters and data for processing. I need have no fear of me corrupting its name space, nor it corrupting mine.

Perl programmers avoid re-inventing wheels by sharing their work. Their generosity has made available a whole plethora of high-quality (mostly), self-documenting code covering areas as diverse as emailing, database, parsing engines, date manipulation, networking, space exploration, etc. It is really is difficult to think of a programming problem which has not been addressed in some fashion. Time and time again on the comp.lang.perl.misc news forum I see the experts' response to a problem; 'go to CPAN and download module ...' (see `www.cpan.org`).

I suspect that most Perl programmers have the ambition to become the contributor of a well-used module to CPAN. It is at these would-be contributors that Tregar has pitched his text. Down loading a CPAN module is almost trivial. Contributing a module is only a

little more complex; one submits a request for comments on an idea, then a CPAN Author ID and once that's in place the code can be uploaded. Typical of Perl, the whole process keeps formality to a minimum.

So, it if is all so simple, why is there a need for this book?

Among other things it springs from a need for portability and standards in code, proper documentation and code maintenance. Tregar also goes to some length to point out that Perl modules need not be written in Perl! Indeed three whole chapters are devoted to writing modules in C (both Inline::C and XS)! He suggests that this is often necessary when interfaces are needed to important C libraries. He claims that many of the more popular CPAN modules are in fact written in C.

Is it worth buying? Yes, if you're interested in the C part. I know of no other documentation. However, if all you are interested in is writing conventional modules, then I suggest its usefulness is doubtful. Yes, if one is to write modules that are to go up on the public domain on CPAN then the code needs to be good, but writing good code should be your objective anyway and there are better sources out there to help with that (see `www.perl.com`).

Note that nowhere in the text does the author suggest 'use strict;' or 'use warnings;'. The source code examples all have file extensions of 'txt'! This smacks of sloppiness, echoed by the book's impoverished page layout and design.

Graphics Programming with Perl by Martien Verbruggen (1 930110 02 2), Manning, 303pp @ £35-99 (1.11)
reviewed by Joe McCool

At first sight, one would not associate Perl with graphics programming. Perl is a 'glue' language, extraordinarily useful for text processing, but it is text progressing ability that has equipped Perl so effectively for the web revolution. The web revolution in turn has demanded graphics.

When I first noticed the author's name, I was immediately excited. Anyone who has spent as much time as I have lurking around `comp.lang.perl.misc`, could not help but be excited. Martien is one of the more frequent and more articulate contributors to that forum.

Here he has written an excellent book, well written and very clear. It might have a limited readership – not many of us are motivated enough to get to grips with the internals of Gimp, GIF or JPEG file formats, but many more will be interested in the mechanics of displaying data on a simple line graph or histogram.

Using the example code from this text one can pull data down from the Web and display it graphically with ease. It does mean down loading a few modules from CPAN (for example PGPLOT, GD and Image::Magick), but this is relatively straightforward.

My copy lacks colour images, so some of the delicacies of shading are missing (for example water marks). Also the book's web site lacks some of the source examples – but the author assures me that his recent upload should fix this. It had been hoped to produce a full colour electronic version that could be downloaded from the web site at a price (`www.manning.com/verbruggen`), but this has proved impractical. Be warned that the electronic version available there lacks cross-links in the table of contents and index.

Highly recommended.

Python Cookbook by Edited by Alex Martelli & David Ascher (0 596 00167 3), O'Reilly, 574pp @ £28-50 (1.40)
reviewed by Tim Penhey

I was inspired to learn Python after reading *The Cathedral and the Bazaar*. One of the major problems of trying to teach yourself another programming language just from books is that you often miss out on the idioms the fluent users of the language tend to use without even noticing. The *Python Cookbook* is an excellent source of Python idioms.

Unlike some of the other O'Reilly cookbooks, the *Python Cookbook* is not so much filled with recipes, but with methods of cooking. Many of the entries are very simple and often show another way to approach a problem. There are well over 100 different contributors to the book.

The book is split up into sections of related 'recipes'. Each recipe has a brief introduction in the form of defining the problem; a solution is then shown, followed by a discussion of the solution. Often the discussions show tweaks to the solutions that are only available in the later versions of Python.

At least a rudimentary knowledge of Python is needed for this book. The book does not attempt to teach Python as a whole, but more specific tricks or idioms to do with the language.

One of my favourites is also due to first learning Perl. Declaring dictionaries (or hashes or maps or whatever you call them) can be a little verbose in Python and there is a nice two-line function that makes it a little more terse.

I find Python an interesting language and I think that the *Python Cookbook* would be a great addition for anyone looking to learn the language. I have no reservations recommending this book.

Other Languages

XML in a Nutshell by Elliotte Rusty Harold & W Scott Means (0 596 00292 0), O'Reilly, 613pp @ £28-50 (1.40)
reviewed by Huw Lloyd

From the outset, this book gives XML centre stage. The scope is clearly delineated and supported by a comprehensive narration of the core XML technologies.

This book helps the reader make sense of a bewildering plethora of XML vocabulary. It depicts the various orthogonal aspects of XML across easily managed chapters. The strengths and weaknesses of technologies are often discussed along with consideration of future and on going requirements for developers.

This book purports to be a reference and readers accustomed to other O'Reilly 'Nutshell' books might be surprised at the degree of verbosity the text offers. Half of the book is an introductory description to the core XML technologies and even the reference section is descriptive. That said the descriptions are usually apt and useful, although I felt a concise author may have brought more clarity to some of the difficult areas of XML, for example, the namespace and DTD chapters were rather muted and lacking in direction.

The reference section is mostly descriptive, it is concise and reads well, perhaps better than corresponding sections of the introduction. The use of production rules and diagrams to depict language constructs in the reference section were particularly helpful.

I would have liked to see some tables of incompatible technology, e.g., browsers and what they support and perhaps less repetition in the introductory sections. For a second edition, the text could certainly have been more concise. Overall this is a reasonable book that should prove useful.

Mastering XML by Chuck White et al. (0 7821 2847 5), Sybex, 1154pp + CD @ £36.99 (1.35)
reviewed by Aaron Ridout

On the whole the book has both breadth and sufficient depth, the authors all know their subject areas and always point you to where to get more depth.

The CD has a nice interface (auto-starts on the PC without installing any fancy viewers) and includes a good HTML based interface for cross platform access.

I can only make two minor criticisms:

1) Missing from the CD is a complete list of web links given in the book, so you have to type them in manually.

2) There are several large tables that cover several pages! Personally I'd prefer them to be available on the CD (there is only 440Mbytes used) in XML of course.

If you need one book on XML that covers a large proportion of current XML applications, then this book would get my vote. Recommended.

XSLT by Johan Hjelm & Peter Stark (0 471 40603 1), Wiley, 310pp + CD @ £37-50 (1.33)
reviewed by Aaron Ridout

While the authors clearly know their subject, unfortunately the layout and print editing is so poor that I cannot recommend this book. The technical details are in the book; you just can't find them.

This book is aimed at the professional programmer who wants to learn this new technology. The back cover starts with a tautology like the sub-title and goes on to extol the virtues of this book as having 'numerous clear examples', numerous perhaps, but not clear.

Most of the examples are incomplete fragments; the exceptions are the 'complete' examples towards the end of the book.

There is no 'How To' cross-reference or index of examples, which is a must for XSLT

because it is not an ordinary procedural language, but neither is it declarative nor functional. So once the beginner has read the book and wants to come back for more details, one could spend hours looking for the solution to a problem, like 'how do I change the value of a variable?' (Answer: you can't, but what you do instead is...)

The companion CD has so little on it (51MB) it was not worth the CD it was printed on! The section in the book devoted to the CD consists of 3 sentences. The first is false; all of the code examples are not on the CD. The second is untrue in relation to the quantity of tools on the CD. The last is nearly true, they are all freeware or demos, but the time-limited tool had already expired and so will not load, nor tell you where to download an upgrade!

The CD does not have an index either in the book nor on the CD, the information on the tools is plain text and without references to where the tools were downloaded from originally nor where to find updates.

The book shows and describes several tools, which are not on the CD, nor are there references to them included in the text, for instance I'd like to know where they got their R380 simulator.

The 12-page index is a mess, the number of typos is large, and the formatting of useful reference material such as function descriptions is poor.

Several examples are so unclear I have taken hours to understand them.

The index doubles as a one-line glossary, but several acronyms used in the body are missing.

For a book which extols the virtues of the separation of content and presentation, this book is a potent example of why this is still an idea in its infancy, as both the book and the CD suffer from the lack of co-ordination that must be rigorously applied to ensure that the automated formatting of data is always correct. Not recommended.

Professional PHP4 XML by Luis Argerich et al (1 86100 721 3), WROX*, 850pp @ $49.99 (UK price unknown)
reviewed by Christopher Hill

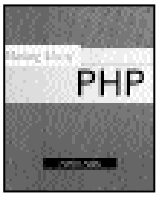
This is another brick of a book, with the usual string of bald and hairy authors on the front of a very red cover.

The book has 17 chapters and 7 Appendixes. With the exception of the opening chapter, each chapter could (should?) be a book in its own right. PHP is 'covered' in one chapter, XSLT in another SVG (Scaleable Vector Graphics) in a third, etc.

Each chapter does give you a feel for the topic of the chapter, enough to have a brief play, but you will be quickly reaching for a full book on the topic.

For the topics that I know, or have actually played with, they seem to be reliable and give a clear presentation. The authors drop a bit of PHP in most chapters, but a number of these appearances are gratuitous.

If you want a book on PHP, I suggest you look elsewhere. The book gives a basic introduction to most of the current XML technologies - useful for a better understanding of the range of XML technologies and maybe for choosing the technology to use, but not a book to which you will often return.



Making Use of PHP by Ashock Appu (0 471 21973 8), Wiley, 345pp @ £25-95 (1.35)
reviewed by Christopher Hill

Over the years I have grown to expect a high standard from this publisher. Sadly this book falls short of my expectations, on both technical and teaching levels.

The book is aimed at basic (with a small b) programmers and as such should present valid code and begin to instil good practice. There is a discussion of string handling in PHP which a) shows that no one ran the code examples and b) shows that Appu does not understand how to handle nested quote marks. A couple of pages on and the `if` construct is described. Again the syntax is incorrect (missing brackets) and the equality operator is given as '= ='. The whole book is riddled with such errors.

The code examples are hung around the premise of building a web site for a CD Music store. Each problem is stated, a task list drawn up, the snippets of code discussed and then the solution is presented. The issue I have is that most of the tasks that Appu selected to code in PHP are best done on a calculator – for example '*Mega Music Mart want to calculate its profit or loss for September. It earned $6,000 dollars* (sic) *in the month and had expenses of $4,000. The development team need to write a code* (sic) *to do the calculation.*' Even allowing for this appearing very early in the book, this is a very poor example and the quality of the tasks does not improve.

Appu does not address defensive programming – validating user input and/or ensuring user input is not going to harm you (the classic example of an input field being applied directly to an SQL where clause). The final straw was when Appu asks the student to open up the permissions on the home directory. He says that this is BAD THING to do, but does it to demonstrate directory functions.

Truly an awful book - strongly **not** recommended.

Creating Applications with Mozilla by David Boswell et al. (0-596-00052-9), O'Reilly, 454pp @ £28-50 (1.40)
reviewed by Joe McCool

Often I choose books to review because they seem relevant to the work I am doing at the time. Only recently I downloaded and compiled a copy of the Mozilla browser (from `www.mozilla.org`). I was prompted by pure curiosity and by the desire for something more modern than my old Netscape 4.7. Some web sites strike me as being unfriendly to Netscape. Its 4.7 version, at least, doesn't contain some 'essential' new features. I'd hoped that Mozilla would ease this difficulty. I do use it, but only occasionally. The text based browser lynx still remains my favourite.

I assumed then that this book was concerned with extending Mozilla (qua browser) in some way. I thought I could add some new features - perhaps make it simpler. In a sense I was correct, but making those changes is complex and not for the faint hearted. Mozilla is much more than a mere browser. It is a complete GUI application development platform. According to the authors it enables programmers to develop cross platform applications that will run anywhere that Mozilla itself will run. (Strange, all the examples quoted are either browsers or chat programs!) It uses a front end called XPFE, itself making use of web standards like CSS, JavaScript and XML. '*JavaScript creates the functionality of a Mozilla-based application, Cascading Style Sheets format the look and feel and XUL creates the application's structure.*' Readers need to be versed in all of these.

In other words, we write the code in XUL. Here is an hello.xul example:

```
<?xml version="1.0"?>
<!— Sample XUL file —>
<window xmlns="http://www.
  mozilla.org/keymaster/
  gatekeeper/there.is.
  only.xul">
<box align="center">
  <button label="hello xFly"
    onclick="alert('Hello
    World');" />
</box>
</window>
```

In Mozilla I can click on this code and it magically executes. Good fun this. The text has lots of such examples. XPFE is similar to DHTML, but has the ability to develop stand-alone applications.

The authors claim that the XPFE learning curve is a lot less steep than the alternatives; C, C++ and Java.

On the one hand this book will have a limited readership. This is new and possibly volatile technology, which may very well be a flop. W3C still haven't approved some of the standards. Mozilla is the open source face of Netscape, which has been taken over by AOL.

On the other hand, there is little else out there. (All I could find was; *Netscape Mozilla Source Code Guide* – William R. Stanek.) Geeks will pull the text and sources down from the O'Reilly web site (`www.oreilly.com/catalog/mozilla`). For anyone else: there's little choice. So if you're interested in Mozilla and XPFE, then this is probably a good buy. It is written and produced to the usual O'Reilly standards.

As a unix purist my suspicions of this technology (not the book) are that this is another addition to the bloatware movement. We're building castles in the air!

Other Programming

Wireless Internet Enterprise Applications by Chetan Sharma (0 471 39382 7), Wiley, 236pp @ £21-50 (1.39)
reviewed by Aaron Ridout

From cover to cover this book says what you'd expect from the title and it is written

authoritatively and concisely but from a mainly USA point of view.

This book is an excellent overview of wireless applications via the Internet or m-commerce. If you need to know what technologies are around now and in the near future, then this book will give a great overview of the technologies and most of the pitfalls best avoided.

This book is mainly for business and technical consultants rather than developers, but that is the stated audience.

The author has obviously tried to include international aspects and has done a good job, but unless you've done it... To be fair the only things left out were Data Protection and Currency conversion.

In 236 pages I found 15 pages with anything I could remotely comment upon and only three of those could be considered detrimental to the reader:

1) The bandwidths for 2.5G and 3G quoted are the original optimistic ones, as reality gets closer all the network operators have down-sized their data-rate figures.

2) EU law, in relation to data protection and privacy, is not mentioned under Internationalisation and Privacy section.

3) One screen image is incorrect and the text verses tables are not on the same pages occasionally.

There are only a few topics that I'd have added:

a) PKI is not mentioned directly but most aspects of it are,

b) The Biometrics section does not talk about repudiation and assumes that anyone can get a class-A certificate 'I am whom I say I am' - Why is it so difficult to get a passport?

c) Currency conversion is not included (assumes that US$ are the international currency)

d) The European Union and presumably other jurisdictions have different Data Protection laws, which may have to be designed into a web-based application.

Highly recommended, but slight USA bias.

ATM Signalling Protocols and Practice by Brandt & Hapke (0 471 63282 2), Wiley, 252pp @ £50-00 (?)
reviewed by Mark Easterbrook

Although the book starts with an introduction and overview of ATM, these are more refreshers - this is not a book for someone who is not already familiar with the complexities of ATM.

The book's focus is very much the UNI and PNNI layers and it concentrates on these and the interfaces to the SAAL. The detail is very much at the bits and bytes level with numerous traces of message flows scattered throughout the text.

This book would be of interest to protocol engineer debugging interfaces between ATM equipment or ATM networks at the interface level. It does not cover implementation and therefore a protocol stack programmer would need to look elsewhere.

I was disappointed to find only cursory references to the MTP3b layer or other SS7 over ATM details after the book starts with a telephony signalling (as opposed to a datacomms) bias.

A cover price of £50 makes it impossible to recommend this book.

Pocket Guides and References

Overview by Francis Glassborow

O'Reilly is turning these out in considerable numbers so I guess that they must meet some real needs of enough people to make them commercially viable. The format is small enough to fit in an anorak pocket, but a tad large for your suit pockets. I remain unconvinced as to their utility to the majority. The format and the page count necessarily means that very limited information is provided. At my normal place of work I would expect to have good and complete references and so would have no need of books form this series. There can be exceptions to this general principle, which I mention in the individual comments I make on the ten books I am currently looking at.

I have withheld the cover images from this block of books because I do not think they warrant the extra space.

C Pocket Reference by Peter Prinz & Ulla Kirch-Prinz (0-596-00436-2), O'Reilly, 134 @ £10.50 (1.42)

This book states on the cover that it covers C99. So I tested it to see if it provided any useful information on that area. Here is what it says on two aspects of C that were added in the latest standard:

> In ANSI C99, any integer expression with a positive value can be used to specify the length of a non-static array with block scope. This is also referred to as a variable-length array.
>
> The type-generic macros defined in header file `tgmath.h` unified names that can be used to call the different mathematical functions for specific real and complex floating types.

The purpose of those quotes is to highlight the problems I have with this book. If I do not already know about the subject, this book will not help me. And if I do know but need to look something up, I will probably need more detail than this book can give.

Digital Photography Pocket Guide by Derrick Story (0-596-00454-0), O'Reilly, 114pp @ £10-50 (1.42)

This is OK as far as it goes, but I would really want to read a decent book on the subject. Every camera model is different, so the only interesting stuff is the general coverage of subjects such as compression, white balance etc. Those are not really properly covered in a short pocket guide.

Essential System Administration Pocket Reference by AEleen Frisch (0-596-00449-4), O'Reilly, 137pp @ £10-50 (1.42)

Let me be blunt, a system administrator needs to know much more than this book covers and should be working in an environment where in depth reference books are available.

Macintosh Troubleshooting Pocket Guide by David Lerner & Aaron Freimark, Tekserve Corpo (0-596-00443-5), O'Reilly, 72pp @ £8-95 (1.45)

Now this time I think the format has something to offer. The ordinary user needs something that covers the areas they can handle before they seek expensive professional help. One advantage of the Mac is that it comes in a very limited range of models and so many of the problems can be pinned down relatively easily.

MacOSX Pocket Guide by Chuck Toporek (1-596-00458-3), O'Reilly, 141pp @ £10-50 (1.42)

I can imagine that a Mac user who was already fluent in using Mac OS 9 might find a small book like this one useful, but I still think they would be better off with a more substantial book. The exception would be if you were using a laptop where having a copy of this book tucked into the carry case could be useful.

Objective-C Pocket Reference by Andrew M. Duncan (0-596-00423-0), O'Reilly, 122pp @ £8-95 (1.45)

I find the existence of this book a real curiosity. I know that GCC will handle Objective C but how many people ever use it in that mode? That is a serious question because ACCU has always been open to including material on this language and no one has ever provided any.

Unlike the other books in this series, it is just possible that this one could enable someone to experiment a little with Objective C with GCC (this book covers 3.1) before deciding if they want to know more.

PHP Pocket Reference by Rasmus Lerdorf (0-596-00402-8), O'Reilly, 132pp @ £10-50 (1.42)

Most of this book (98 pages out of 132) simply lists 1404 of the 2750 functions that ship with PHP 4.3. It gives the version where each was introduced together with a one-sentence summary. I think the author would have been better advised to go for broke an even with the higher page count have covered all 2750 functions.

Because PHP is something that consultants sometimes have to fix on site this book could be useful, but could have been more so, even in this format.

VB.NET Language Pocket Reference by Steven Roman, Ron Petrusha, & Paul Lomax (0-596-00428-1), O'Reilly, 142pp @ £9.95 (1.50)

If you are going to program in VB.NET you need a full reference text so I would give this book a miss.

Windows XP Pocket Reference by David A. Karp (0-596-00425-7), O'Reilly, 181pp @ £8-95 (1.45)

I personally would take a copy of Windows XP Annoyances with me when visiting a client on site because its more comprehensive coverage is more likely to provide the clues I need when trying to fix a problem. However, if I had a laptop using Windows XP (mine uses Windows 98) I think that I might well have a copy of this book in its carry case. There is enough in it to

be helpful and even at 180 pages (note that, definitely still an anorak pocket book even though 50 pages longer than the PHP volume.) it is light enough to add to the essential gear for a laptop.

Word Pocket Guide by Walter Glenn (0-596-00445-1), O'Reilly, 143pp @ £8-95 (1.45)
This is another book that I might add to my tools to take onsite with me. At home I would want something more substantial but most of the problems my clients get into can probably be handled via reminders from this book. I would view it more as an aid to troubleshooting than a guide to Word. It covers version 97, 2000 and 2002.

Unix

The Unix Programming Environment by Kernighan and Pike (Prentice Hall, 1984). reviewed by Ian Bruntlett
Essential reading for Unix users, albeit a little dated – by now shell and sed scripts have been replaced with Perl scripts and K&R C has been replaced with ISO C. It contains gems of background information that will improve your understanding of the Unix system. Indeed, the aim of the book is to promote greater understanding and promote effective use of Unix, not by cataloguing details but by explaining the system and the relationships between its component parts. This book isn't just for Unix users –if you are using Windows then this book will help you come to terms with the Cygwin Unix utilities for Windows.

The book consists of ten chapters. The first of these, Unix for Beginners, gives an overview to get you started using a Unix system. It covers basics (logging in/out, elementary commands, terminal use), day to day use (files, file system, printing files, directories) and use of the command line interpreter / shell. Unfortunately it makes reference to the Unix Programmer's Manual, something that most people won't have a copy of. It has a few quirks, such as using ed as an editor instead of vi or emacs or referring to a now non-existent filename length limit of 14 characters.

The next chapter, The File System, covers everything from the directory layout to the meaning of inodes. Again, some of the material is archaic - using octal instead of hex. Some of the explanations are a bit terse but on the whole they seem ok. The description of inodes (index nodes), file permissions and the security model is good and thorough. It would have been nice to see mentioned that the 'ls' command is an abbreviation of 'list sorted'.

Using the Shell covers meta-characters, quoting, creating new commands, passing arguments to them, the use of shell variables and some elementary control-flow. After CP/M, DOS and Windows, the handling of shell variables seems arcane but with the tutorial provided it is possible to see the point. Unix shell programming seems so powerful that it's a pity to use COMMAND.COM or CMD.EXE.

Filters covers the most commonly used filters - programs that take stdin, performs a simple transformation on it and write the results to stdout. It covers sed scripts which seem to be an acquired taste - especially with their regular expressions and escape characters to stop key characters from being interpreted by the shell. A tutorial for the AWK programming language is presented and is used to provide more filters (wordfreq, fold, field, addup, and calendar).

Shell Programming shows the basics of shell programming (Unix batch files), illustrated through the evolution of some useful shell programs (which, watchfor, watchwho, checkmail, nohup, overwrite, replace, zap, pick, news, get, put). It's a very productive chapter and shows the great potential of the shell.

Programming with Standard I/O deals with the C standard I/O library, again illustrated with the use of examples (vis, p, pick, zap, idiff) - all written in old-style K&R C. Some of the idioms are unusual but they can be reasonably easily understood.

Unix System Calls deals with I/O, directories, inodes, signals and interrupts. As usual a mixed bag of example utilities is provided (waitslow, cp, spell checking filenames, sv, waitfile, timeout).

Program Development covers the development of an interpreter using yacc, make and lex. It isn't a substitute for the yacc or lex manuals but it does have a good set of examples to work through - I didn't have time to work my way through it.

Document Preparation deals with document preparation using troff. I skipped this chapter as too much hard work, preferring to stick with a word processor or HTML.

The final chapter, Epilogue, concludes the book. It also lists the Unix approach or philosophy;

1. Let the machine do the work. Use programs like grep, wc and awk to mechanise tasks that you might do by hand on other systems.

2. Let other people do the work. Use existing programs glued together. Write a small program to interface to an existing one that does the real work.

3. Do the job in stages.

4. Build tools.

In conclusion, for all its archaisms this is a really good book that challenges the reader and provides useful, meaningful exercises. Highly recommended.

Methodologies etc.

UML Weekend Crash Course by Thomas A Pender (0 7645 4910 3), Wiley, 355pp + CD @ £18-99 (1.57) reviewed by David Nash
The idea of this series of books is to provide an introduction to each chosen subject in a sequence of lessons, designed to be followed over the course of a weekend. Although their idea of a weekend is slightly generous, lasting from Friday afternoon, through Saturday evening, until Sunday afternoon.

I must admit up front that I did not spend a weekend reading this book. Rather, I read most of it over a period of two or three weeks. I am a little sceptical about the value of such a 'cram' style course and I think it is more realistic to read the book over a longer period of time, as I did for the review.

Setting aside minor quibbles regarding the format of the 'course' I have to say that the content is very good. It is written in a clear manner and covers all the features of UML that I have come across in the past, although not being an expert I couldn't say whether it is completely comprehensive. It certainly contains enough to be useful though.

The book goes through all the usual stuff: Use cases, class diagrams, activity diagrams and sequence diagrams and so on. It also separates the different models into the Static View, the Dynamic View and the Functional View, representing different aspects of the system being modelled.

One criticism I would make is that it gives no coverage to the subject of choosing your classes, or designing them. Although, to be fair, it does not try to be a book about design. However, the omission is rather obvious when applying the methodology to the case-study used throughout the book and the author suddenly jumps from having described class and object diagrams, to 'now identify your classes' - without offering a great deal of help in how that should be done.

When the book reaches the 'Sunday Afternoon' section (session 27), it suddenly switches from the previous case study (an order fulfilment/dispatch system) to a new one, involving Java-based web services. This involves a quick tutorial on such systems, which to my mind is unnecessary padding that is not at all relevant to the subject being taught. It gives the impression of trying to be a bit trendier, by involving HTML, XML and Java server pages, than the other case-study, which is of a more traditional nature.

The book also includes a CD-ROM, with the full text of the book, plus supplementary material and the Object Management Group UML standard (all as PDFs) and for those running Windows (NT4 or greater in one case) a couple of sample UML generating tools. Curiously enough the Windows-specific files on the CD were not visible to my Linux PC, although the rest were and the files were all there when I rebooted into Windows. I was (pleasantly) surprised to see that the tools don't get much coverage (besides a short chapter) in the book, which prefers to concentrate on the models themselves rather than the tools used to generate them.

Overall a well-written book and I would recommend it, although perhaps not to be used in a single weekend sitting.

Extreme Programming Perspectives by Michelle Marchesi et al (0 201 77005 9), Addison-Wesley, 623pp @ £37-99 (1.33) reviewed by Rob Hughes
This book is a collection of short papers by diverse authors on a wide range of XP subjects and Agile Methods (AM) in general. This style of book can prove difficult, with a lack of continuity and no clear writing style to aid flow. This book manages to avoid these problems; each paper is short, often less than

ten pages and it is clear that significant effort has been put into ensuring writing styles are similar. I wonder if this is just a fortuitous and/or clever decision by the authors, or perhaps reflects a general mindset of those deeply involved with AMs.

The ideas covered are extremely broad. Some of the papers are technical and describe particular aspects of XP, others advise how to go about switching to an AM, yet others describe real-world experiences of using AMs. I found it very easy to read, enjoyed being able to dip into articles that took my interest and found that complex areas were well covered despite the brevity of the articles.

My only real concern with the book is with regard to its intended audience, which would seem quite limited. This is an excellent way to get an overview of AMs if you don't already know much about them; to anyone who is interested in getting into the meat of XP, I suspect that other books are a better bet. Unfortunately I also suspect the already limited audience may also find the price off-putting; I don't see many people splashing out the kind of money asked for just out of general interest and for a nice read.

Mastering UML with Rational Rose 2002 by Wendy & Michael Boggs (0 7821 4017 3), Sybex, 675pp + CD @ £45-99 (1.30)
reviewed by Rick Stones

Firstly I must say I am not a member of the 'UML is the solution to all modelling problems' camp. It's good, but not a panacea for all problems. Also the CD in the back of the book, as stated on the book cover, has some of the models used on the book, not (unfortunately) any trial type edition of Rose. I'm not sure why they didn't just make it downloadable. Now we have those out of the way, the book review.

This book seems to be aimed at a fairly specific market. It doesn't try and teach you UML from the ground up and people who are already UML experts will probably not need this kind of book to get started with Rose. The book assumes some basic knowledge of UML and an installed copy of Rose. It starts with the very basics of using Rose, even telling you that the magnifying glass is a zoom tool; using numbered bullets for stepping through fairly obvious menu trees and screen shots for the font selection window did seem rather silly to me.

After the first couple of chapters, the level of detail changes to a much more sensible one and the book explains things with a nice blend of problem to model, the kind of UML approach that you use and then how to actually do it in Rose. Along the way there are a few handy hints and tips to help you use Rose effectively. Most of the last quarter of the book is devoted to code generation (C++/Java and VB), which I thought was a little excessive in terms of space devoted to it, though the DTD generation was interesting. That may just be my prejudice against generated code. It would have been nice to see a completed worked example in the book, but maybe they felt that was more the province of books teaching UML.

Overall, after a bit of a slow start, I found the book quite helpful. It is a bit verbose in places and could easily have been shortened by 100 pages. If you already know the basics of UML and need some help translating that into the practicalities of using Rose, then you should find this book useful.

Software Requirements: Styles and Techniques by Soren Lauesen (0 201 74570 4), Addison-Wesley, 591pp @ £35-99 (1.11)
reviewed by Silvia de Beer

The title gives a correct summary of this book; it deals with styles and techniques to write software requirements. It is not a cookbook of how to write your software requirements but a summarisation of styles and techniques. The first time I paged through this book I noticed that from page 439 onwards the book solely consists of examples of requirement specifications, all on a grey background. The examples only seem to be there to increase the page count. For someone who uses this book as self-study, the example requirements specifications do not seem to be very useful. However, this book is set up so that it can be used in courses. The exercises that are based on those requirements specifications might actually be quite useful. Another visual impression was the many white odd pages, because the editor chose to start the sections on an even page.

The first half of the book is interesting for people who are actually writing requirements, e.g. people who write Requests for Proposals (RFP), proposals and functional requirements as a first step during a development process. The book concentrates on writing requirements for simple business processes and it does not cover very technical systems.

During the second half of the book I lost my interest somewhat. The author repeats himself a few times and fills many pages with trivial explanations. The most important observations that the author has to offer are in the first four long chapters. He explains that you can write goal-level, domain-level or design-level requirements and that requirement roles can differ depending on the project type; in-house development, Commercial Off The Shelf deployment, subcontract, etc. The author has a fairly traditional view of requirements specifications, probably because of his background and prefers the traditional E/R model above UML. Other traditional diagramming techniques are also explained in his book.

Management & Leadership

Software Project Management in Practice
reviewed by Greg Billington

This book is a good read for a project manager who is working in an organisation with a focus on process improvement, particularly those going down the SEI CMM route.

It covers a lot of material and each chapter has a consistent style; overview and explanation of the topic, a real life example from Infosys (a company with high process maturity), then a summary. The summary is typically a few short bullet points on half a page, which contains the nuggets of wisdom. You could read the twelve summaries, but these become clearer having read the previous chapter.

I found the topic explanation straight forward but not eye opening, the real life examples though bring it all together and show detailed, clear and concise document extracts from a sensible (well I could relate to it!) project, e.g. 9 people doing 500 person-days software project. The fact it was not a trivially small or overtly large project allowed me to relate to it and I was refreshed that there was no CD in the cover, so the examples showed the simplicity of information required without the need for numerous Microsoft Word templates.

The final chapter is the shortest and covers project closure, 5 pages of explanation and 10 pages of example. I found this interesting but it seemed too short to cover such an important topic.

Software Leadership by Murray Cantor (0 201 70044 1), Addison-Wesley, 193pp @ £22-99 (1.30)
reviewed by Rob Hughes

Murray Cantor writes with an eloquence and brevity that many would do well to take note of. There is no needless repetition of largely irrelevant points here and this book should be recommended simply because it is a pleasure to read, but we should consider the material it presents first.

The book addresses two major issues; the nature and frequent failure of software development and maximising the chances of successful software development.

The first of these issues is handled in a simple and comprehensible way. Difficult conceptual issues are explained in an approachable fashion, allowing the least technical of people to understand the issues involved. For example, the author elegantly explains why the heavily controlled processes of the past have failed; trying to force a linear model onto a non-linear process is doomed to fail.

The presentation of alternatives to traditional software management is not quite as good. The general concepts are well handled, but there is a little too much pressure towards the Rational Unified Process. Perhaps understandable given the author's role at Rational, but nonetheless, the book gives the impression that the RUP is the only reasonable approach to use, which I felt was perhaps unfair. Specialists might see through this commercial pushing, but managers without a significant IT background – who really should be amongst the audience for this book – may not pick up on this and might see the RUP as the next silver bullet. That is not to say that the RUP is not a good solution, but people shouldn't

be guided into thinking it is the only solution.

I have chosen a quote to give as an example of how Cantor shows a deep understanding of not just software development processes, but software development people as well.

> The fact that programmers can control the unforgiving machine makes them different and creates a sense of separation from those who cannot. [...]. If you cannot write a program, you simply do not get it. If you do not get it, how can you possibly control what the programmers do?

This book is well recommended to everyone from developer to tester to manager. But for the periodic pushes towards the RUP, a highly recommended rating might be justified, but that aspect just dragged it down slightly for me.

Testing & Security

Test-Driven Development by Kent Beck (0-321-14653-0), Addison-Wesley, 220pp @ £22-99 (1.30)
reviewed by Francis Glassborow

The author's credentials for writing on this subject should be well known to you. Just in case you have missed it, he is one of the principal architects of the Extreme Programming member of the Agile Programming Alliance. Once you realise this you may also, correctly, conclude that the code will not be C++. It maybe that I have missed something while reading this book, but I do not think the author actually tells us what language his sample code is written in. I am pretty certain that the code in Part 1 has been extracted from Java. I say extracted because the code is all implementation level detail.

When we move on to part 2 the author does tell his readers that he is now going to use Python. I think the code reverts to Java for part 3 where the author tackles 'Patterns for Test-Driven Development.'

The first two parts of the book very much support the 'by Example' of the title. In part one he uses the development of currency support and in part 2 walks you through the design and implementation of a testing framework.

The book is written in many small chapters which is an example of text emulating method because one of the authors continued themes is that we should proceed by small steps. In other words he advocates a form of minimalism, do not do more than you have to and do not do more than one thing at a time. The result is that we have a book of 32 chapters packed into 205 pages. As he always starts a new chapter (and part) on an odd numbered page, you will find quite a few (21) completely blank pages plus an average of half a blank page per chapter. That makes the average chapter between five and six pages. But that is exactly the right message for this book.

The second message of this book is that you should write your tests to fail. By that I mean you should imagine what could go wrong and then test for that. Let me give you an example from some code I was writing recently where I needed to find the Polar co-ordinates of a point that was provided in Cartesian form. What can go wrong? Well the y co-ordinate might be zero. How can we tackle that? Well the very first thing is to write a test that tests exactly that case. The next thing is to realise that using an arc tan for the special case must be a common problem, so look at the Standard Library and find `arctan2()`. That is not the end of the problem, because that only deals with division by zero (by side –stepping it). We also need to test for the case when both x and y are zero. That test will quickly force us to revise our code for calculating an argument by handling the special case.

This book will have little to offer if you already take a minimalist approach to your code writing (or as I describe it, never do anything till you have to) coupled with writing tests to detect assumptions (How I wish teachers and instructors would write such tests and apply them to their students work – good students would pretty quickly catch on to writing their own tests to ensure their assignments passed. Actually those with a degree of entrepreneurial sense might charge a small fee for pre-testing the work of the idler students). The rest could benefit from considering this approach and reading this book could help.

Introducing Software Testing by Louise Tamres (0 201 71974 6), Addison Wesley, 281pp @ £29-99 (1.50)
reviewed by Francis Glassborow

By comparison with *Test-Driven Development by Example* this book tackles the other end of testing; that associated with accepting a product. The book does give some coverage of unit testing but is more concerned with the other aspects of testing.

The book is aimed at those who are new to testing and those who are responsible for mentoring new testers. This is a valuable objective. It is easy to say that software should be tested and simply assume that everyone knows what that means. But in practice it is very easy to write tests that actually assume things about data that may not be true. It is easy to test that software does what it is supposed to do, but what about testing that it handles cases for which it was not designed? How careful has the programmer been with assumptions? For example, it is a common assumption in the US that a social security number uniquely identifies a person. The hidden assumption in that statement is that you mean 'living person'. But how do you test that the writer of some software has not made that assumption? Yes complete testing is hard. Testing multi-threaded code is hard. Testing software that must run in a multi-tasking environment is hard. Testing software that reads in files is hard (remember that the file might be corrupted, a point completely ignored by MS Word 6 which would pull everything down if provided a `.doc` file that had got truncated or otherwise corrupted).

The point that I am trying to make is that understand what and how to test is not some innate knowledge that we acquire by osmosis, it is something we have to learn along with learning to write software and to write documentation. How many Computer Science courses teach software testing alongside programming? Some do, but most do not (just as many teacher training courses never teach how to test pupils.)

Maybe you are a programmer and just wish that the test department would go away and stop bugging you. You are wrong to think that way. You should want those responsible for doing the testing to test properly and so point to the failure points in your work. At the same time those in the test department should appreciate being told how they can better test software (yours included). That way the product you are jointly responsible for will better meet the needs of the customer.

This book is not a casual read. It is the kind of book that, in my opinion, is best read a few pages at a time followed by giving careful thought to what you have just read. It is important to understand what testing is about and the responsibilities the tester has to both the writers of software and the consumers of the results. It is the job of a tester to help the software writer to produce a better result. It is also the job of the tester to ensure that the result meets the needs of the client. I think the author of this book tackles these twin responsibilities effectively.

Whether you are about to transfer to the testing side or you want your work better tested you need to put in some time studying how to do it and do it better. This book is an excellent place to start your studies.

Writing Secure Code 2ed by Michael Howard & David LeBlanc (0 7356 1722 8), Microsoft Press*, 768pp @ £36.99 (1.35)
reviewed by Francis Glassborow

Somehow I missed the first edition of this book, and when I look at the contents I get a feeling that quite a few people in Microsoft also missed it. Before you read any further you should know that this book is very much specific to those writing for Microsoft OSs. No doubt those writing for Linux, System X etc. could gain some useful insights into the general principles of writing secure code, but much of the text is aimed at addressing the problem on MS based systems that would considerably reduce its value to such readers. The problems would be largely the same but the solutions could be very different.

The book is divided into four parts followed a set of short appendices. In part 1 the authors introduce the current position and highlight why it is different from the past. For example, right at the start of chapter 1 we have:

> As the Internet grows in importance, applications are becoming highly interconnected. In the "good old days," computers were usually islands of functionality, with little, if any, interconnectivity. In those days, it didn't matter if your application was insecure – The worst you could do was attack yourself – and so long as an application performed its task successfully, most people didn't care about security.

This sets the motivation for the whole book and in a sense provides a justification for the very poor security that many Microsoft

products suffer from. In the days when viruses where transmitted by floppy disks, good discipline by the computer owner was largely sufficient. The developing theme in this section is that times have changed and developers need to change with them. The concluding chapter of this section highlights the degree of change necessary by tackling the issue of threat modelling.

In part 2 the authors cover the more straightforward aspects of secure coding. Inevitably they start with the problem of buffer overruns. Here it is interesting to note that they repeatedly emphasise that such tools as the .NET /GS (similar to Crispin Cowan's *StackGuard* for GCC produced apps.) option are not a replacement for responsible programming. The need for responsibility and understanding is a continuing thread in this book. In affect the authors are saying that producing and distributing an application without understanding potential abuse and protecting against such is unacceptable. I agree with them, but it will take many years to remove all the dangerous legacy code and applications that are in regular use round the world. Just imagine the reaction if Microsoft stated that it was withdrawing the licence to use all versions of Windows prior to some new release (let us call it SXP) on the grounds that they could be exploited to damage other users (which, of course, they can via such things as denial of service attacks). That raises an issue that is not covered in this book, the need for responsibility and understanding amongst computer users (for example, all users of always-on Internet connections have a responsibility to ensure that their systems are kept secure).

The chapter I particularly enjoyed in part 2 is the one titled *All Input is Evil.* This is something that needs to be drummed into programmers from day one. This chapter starts with:

> If someone you didn't know came to your door and offered you something to eat, would you eat it? No, of course you wouldn't. So why do so many applications accept data from strangers without first evaluating it? It's safe to say most security exploits involve the target application incorrectly checking incoming data or in some cases not at all.

The rot starts with books written for newcomers to programming. Pick up any one of them aimed at introducing C or C++ and look at the sample code. I will lay very heavy odds that input is via `scanf()` or `gets()` in C, and `operator >>` in C++. The student is taught to trust input by default. This must be wrong.

In part 3 the authors deal with several higher-level security issues such as socket security. While the book is generally very much targeted at developers for Microsoft OSs, this section is very strongly oriented this way.

The final part of the book is titled *Special Topics* and is concerned with such things as testing, code reviews and documentation from a specifically security viewpoint. This part includes an excellent chapter on *General Good Practices*. Those of you who feel complacent about the security of your equipment might note:

> Because of these design features, any service that opens a window on the interactive desktop is exposing itself to applications executed by the logged-on user. If the service attempts to use window messages to control its functionality, the logged-on user can disrupt functionality by using malicious messages.

I guess many of you are going to feel irritated by my conclusion to this review but if you write software to run on a PC using a Microsoft Windows OS you should read this book and be familiar with what the authors have to say. And if you feel complacent because you use some other operating system such as Linux, it is time to wake up and recognise that the world has moved on and security is everyone's responsibility. Defensive programming these days isn't just a matter of asking what can go wrong, but requires that you ask yourself how your work can be abused. Of course, if you are just writing a small application for your own personal use which does not rely on any widely available library then you can probably still adopt the attitude that only you will pay for your mistakes.

This is not a book that I recommend because it will improve your ability to write good code, it is a book that I consider to be essential reading by any responsible developer (the only people I will exempt are those that are already working in teams that routinely require such things as threat analysis on their work) so that their code does not damage others. ACCU does not have a recommendation classification for books like this one. Perhaps this should be the first book under a new heading: **Essential Reading for Developers for MS Windows**.

Network Security with OpenSSL by John Viega et al (0 596 00270 X), O'Reilly, 367pp @ £28-50 (1.40)

reviewed by Christoph Ludwig

All programs that use network resources are potential objects of snooping, tampering, or other attacks. Even if you are familiar with cryptology, chances are your programs won't withstand sophisticated attacks unless you rely on established protocols like SSL and its successor TLS. OpenSSL is a highly regarded C implementation of these protocols. Unfortunately, there are plenty of subtle traps when using OpenSSL, giving you a false sense of security. *Network Security with OpenSS*' aims to show you how to avoid such pitfalls.

It presumes that the reader already has a rough understanding of modern cryptology. The explanations given are too vague to serve as more than a reminder. The authors explicitly avoid the details since they do not have an immediate impact on the use of OpenSSL. They may have a valid point here, but I miss a comprehensive commented bibliography that allows the reader to look up details or further information.

The book does not explain the design of OpenSSL or give a complete reference of all available C functions. It's rather like a cookbook; it shows how to perform certain tasks like setting up an SSL connection with proper authentication of client and server or how to correctly use message authentication codes. However, simply adapting examples when developing security relevant parts of a program leaves an uneasy feeling. A more complete discussion of all possible options and how different parts of OpenSSL interact would have been helpful.

There are more or less complete OpenSSL bindings to other languages like Perl, Python and PHP. The book dedicated one chapter to these bindings, but I doubt whether 24 pages suffice to cover everything that you need to know in order to make efficient use of OpenSSL from other languages.

In summary, if you are going to use OpenSSL, then I recommend you have a look into this book. It will give you a good starting point, at least. However, it won't serve as your only reference, you will have to dig through the documentation shipping with OpenSSL and some textbook on cryptology anyway.

Other Computing

Essential SNMP by Douglas Mauro & Kevin Schmidt (0 596 00020 0), O'Reilly, 313pp @ £28-50 (1.40)

reviewed by Rick Stones

Having recently bumped into a need to know a little about SNMP at short notice, deciding to review this book was well timed. For those who don't know, SNMP is a network management protocol for managing and monitoring devices on IP networks. It can also be used for monitoring things such as server load and performance.

The book assumes some very basic networking knowledge, but no SNMP knowledge and pretty much jumps in with how management information for networking is defined by SNMP, how OIDs are allocated and the SNMP data types. It then uses the free Net-SNMP client package to illustrate the theory, usually using a Cisco router as the device to monitor, although you don't need to know much about Cisco routers to understand what is going on. Everything is referenced back to the relevant RFCs, if you want to look for the more formal base definitions. Where programming is needed things are kept very simple and written in easy to read (honestly!) Perl. This section of the book I found well written and authoritative, though more explanation might have been helpful in places, at least for this reader.

The next section of the book moves on to the more practical aspects of setting up network management tools to manage and monitor your network, looking mainly at commercial tools, such as HP Open View, although other free tools are used where they exist. The book is full of practical information, different ways you could structure the monitoring, when to use polling and when to rely on traps, etc. You do get the feeling that the authors have very much 'been there and done it for real'.

If you know the basics of TCP/IP networking and need, or want, to learn about SNMP from the ground up, this title has pretty much all the information you need along with a lot of helpful advice. Recommended.