Contents

Reports & Opinions

Reports

Dialogue

Features

Reviews

Copy Dates

C Vu 16.1: January 7th

C Vu 16.2: March 7th

Contact Information:

Editorial: James Dennett
914 24th Street,
San Diego
CA 92102, USA
cvu@accu.org

Advertising: Chris Lowe
ads@accu.org

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

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

Secretary: Alan Bellingham
01763 248259
secretary@accu.org

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

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

Membership fees and how to join:

Basic (C Vu only): £15

Full (C Vu and Overload): £25

Corporate: £80

Students: half normal rate

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

There are 6 issues of each journal produced every year.

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

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

Any questions - just email membership@accu.org

Reports & Opinions

Editorial

2003 in ACCU...

It's been another interesting year for ACCU. Another fine conference, the publication in affordable book form of the latest versions of the C and C++ Standards, a new chairman, new treasurer, a healthy membership including increasing numbers outside of the UK, continued activity in the various Mentored Developers projects, and I'm sure I'll owe apologies to many people for omitting their contributions. I'd like to offer my personal thanks to all those whose contributions in the past year have made ACCU such a worthwhile organisation. With a web site redesign underway and other changes planned or happening, 2004 looks set to be yet another interesting year.

Where Will C++'s Successor Live?

Like many of you reading this, my interest in programming languages extends beyond learning how to use them as tools to do the job of a software developer. Programming languages evolve just as natural languages do. Well, maybe not "just as" – the evolution is different, being more deliberate, but evolution it still is, and natural selection applies at numerous levels. In their early stages, languages evolve more quickly as experimental features are added, removed, changed, requested by users, and variously found to be flawed, helpful, or irrelevant. Some environments foster change, and others with more concern for the cost of change tend to stabilize languages.

Some years back I found the newsgroup `comp.lang.c++.moderated` and was pleased to find a community of people whose commitment to real understanding of one language was evident. From posts there I found `comp.std.c++`, which concerns itself with the standardization of C++ and the evolution of its standard, and after some while I volunteered as a moderator of that forum – I was reading it anyway, so it took little extra time. (If any among you dare to admit to occasionally enjoying a passionate argument about exactly what an rvalue is, or whether `#define` for `if(0);else` is really legal (and if that matters), I do recommend a visit to `comp.std.c++`. Or even if you just want to learn more about C++ than you would ever use in writing production code.)

People who are interested in making C++ better look to see how the future of C++ is being shaped. That means the "C++ Committee", actually a collection of national standards bodies gathered together under the umbrella organisation that is ISO (which is not an acronym, for a change). So, like me, many people find `comp.std.c++`, or come along to meetings of the BSI C++ Panel, convened by our own Lois Goldthwaite. Indeed the BSI C++ Panel overlaps so much in membership and interest with ACCU that at times it can take a conscious effort to remember that the charters and responsibilities are separate. After a while of looking at the standards process as defined by ISO and its member bodies such as BSI, many of us gain a great respect for all of the work that has gone into ISO standards, and the high quality of those standards when compared to many produced by more commercial processes. We also come to agree with Bjarne Stroupstrup's prediction that the foundation C++ built on in extending the C language would eventually become more of a liability than an asset. Some would agree that we have reached that point. C is a fine language, but trying to stay similar to C prevents C++ from moving forwards in a large number of ways.

The C++ committee is conservative, as befits an ISO standards committee... they're not going to axe the ugly pieces that C++ inherited from C. ISO committees are required to place great weight on respecting existing standards, and on considering the great burden that changes in a widely-adopted standards can place on users and implementers. Even if the C++ standard does mandate a change, some vendors will continue to support the older behaviour for many years to avoid angering their customers. Even writing a compiler that rejects such clangers as "`void main`" is likely to result in an increase in bug reports from users who don't know the language well enough to understand that the compiler is doing them a favour by telling them of errors in their code.

Various groups, knowing what C and C++ have achieved but mindful of the cost of too much backwards compatibility, have produced new languages based to various extents on C and C++. Examples include D, Java, C#, Managed C++ (and its replacement in a new "binding" of C++ to Microsoft's CLI). Most of these throw away more from C++ than I would, or add too much – the dividing line being that they discard the spirit of C++. Requiring garbage collection is a valid choice for a language, but disqualifies it from use in many environments. Disallowing access to raw memory is a valid choice. Disallowing full-blown multiple inheritance is a valid choice. These valid choices, however, are examples of decisions that (to me) violate the spirit of C++, and it would be possible to write a C++ successor that did not have to make such radical compromises. I would be prepared to argue that C compatibility is only a very minor part of the spirit of C++.

How would such a successor to C++ come to life? I don't know. Java and C# had the backing of huge companies. D has been created by a single (talented) man. C++ came from a small team who managed to flourish inside a large company, and was then opened up and implemented elsewhere. So, I write this with a sliver of a hope that someone reading this, or someone who they speak to, might have an answer and the passion to do something about it. I'd love to see a C++-killer of a language, but I've not seen it yet.

Here's One I Made Earlier*

Unlike the creations magically produced by Blue Peter presenters of yesteryear, however, the one I made earlier did not work out so well. In the last C Vu I wrote (quieten down, whoever said "ranted") at some length about the actions that the company Verisign has taken to break the domain name resolution system that is one of the most centralized systems on the Internet. Once again, the Internet shows that the print format is too slow to use to cover news stories. By the time that editorial had gone to print, Verisign had been threatened sufficiently sternly by ICANN, the body responsible for appointing Verisign, that they agreed to a ceasefire, and suspended their practice of returning incorrect results for all unregistered domains in the `.com` and `.net` top level domains (TLDs). So, was my time in writing that editorial wasted? (Quieten down again – that was rhetoric.) I think not. One thing that this editorial spot gives me is a chance to raise the profile of issues that matter to me, and that I think should matter to you. If a few more of you take time to check the ethics of a domain registrar before doing business with them, that's a positive effect. If a few people are motivated to act to protect the integrity of the protocols that underlie the current Internet, that's another. If one or two people join the EFF (Electronic Frontier Foundation, `http://www.eff.org/`, whose aim is to protect "digital rights") as a result of those thoughts, I'm pleased, and I might even be shamed into doing the same. If anyone's still reading at the end of a paragraph this long, it's little short of a miracle. Time for a break.

Do any of us still seek news from newspapers? Recently I realized that I do not. I still read newspapers for in-depth coverage, and sometimes because they condense a wide range of information more concisely than most other media, but for up to date news the first place to look is "online". Sometimes the television networks can be quick, but often the Internet is faster. News web sites are as fast as TV; weblogs are often faster, and realtime chat is faster still, if you can separate the signal from the noise. We have solved the problem of getting data quickly – the pressing problem now is to separate the useful information from the garbage. That applies in many (maybe most?) of the channels from which we might receive information online. Realtime chat is frequently banal, but can be a place to find diverse experts in minutes if you are lucky. Newsgroups are rarely worth the time to read unless they are moderated, though I know that the occasional rare individual who takes exception to the moderation policies of

* With apologies to readers outside of the UK, to whom this reference to a British children's TV program of my youth might be sadly opaque.

comp.std.c++ would disagree with me. 90% of my e-mail is now spam, though all of the e-mail clients I use can automatically classify the vast majority of that. And the Web? An invaluable source of information, though "surfing" from site to site is now rarely a viable way to find good material, and navigation by search engine is a more common mode of navigation among experienced web users. Technologies are adapting to helping us to find the information we want, now that we have instant access to petabytes of data containing terabytes of useful material.

And Finally...

Finally for this year's C Vu editorials, anyway, I would like to wish you an enjoyable end to 2003, with whatever December may mean to you. I hope each and every one of you can spend it doing enjoyable things with fine people, as I intend to. I'll also be raising a glass to absent friends, some of whom will be reading this. Here's to a good end for 2003 and a great 2004 to follow it.

James

From The Chair

Ewan Milne <chair@accu.org>

When I became ACCU Chair earlier this year, there were two pressing issues for me to attend to, both relating to the upcoming conference. Now that we have a new organiser in place and a team supporting me in organising the programme, these matters are well in hand. This gives me the breathing space to start thinking a little about the future. In fact, in the New Year, and leading up to conference time in April, I'd like to get us all thinking about the direction we'd like the association to take in the future. Many of the features that now greatly add to the strength of the association started as initiatives from the membership: the mentored developers scheme is a perfect example. It is through this sort of organic growth that we can continue to enjoy the lively and enriching environment that the ACCU offers.

So, expect me to start stirring up a little bit of debate in the next few months. In the meantime, I hope you all have an enjoyable Christmas.

Membership Report

David Hodge <membership@accu.org>

The membership now stands at 922 with around 80% of the membership having renewed (mid-November). Final reminders are now being sent out to persuade the stragglers to renew. If you feel you would like to make life easier for yourself next August why not start paying by standing order, just email me for a form or details of our bank account.

There have been a number of requests for the web renewal page to come up with your current address details. We prefer to keep all the membership details in one place (my computer). However we could have a fast track system for those members renewing where their details are the same – maybe just enter Surname, Membership number and amount. I will look into it for next year.

Publications Officer's Report

Tom Hughes <tom@accu.org>

Following a suggestion made by one of our members, the committee has recently decided in principle to submit Overload to an online database of journals used in libraries and academic institutions around the world.

The service in question is that provided by EBSCO (www.ebsco.com) and we hope that this action will raise our profile by making material more widely available to researchers and others who might not currently be aware of our existence.

Negotiations are still in progress, but it is likely that we will start submitting the contents of Overload beginning with the next issue of the journal, the first issue of 2004.

If anybody does have any comments to make about this plan then now would be a good time to communicate them to me in order that I can pass them on to the committee.

Standards Report

Lois Goldthwaite
<standards@accu.org>

Christmas came early this year, when my copy of the newly-published C++ Standard arrived in the post. ACCU and its support for standards activities get a nice plug in the introduction (erm, written by me, but it's still there). Although BSI held off permission for Wiley to print the document until the updated standard had received approval in a vote of national standards bodies, the hard-bound edition got into print several weeks faster than ISO itself managed to make it available. However, you can now order ISO 14882:2003 from the ISO web site, priced at CHF 364 (or £164). As I write, the BSI on-line store doesn't have the new edition yet, but the 1998 version costs £110 from them (plus supplying your own binder). Or you can get a copy from your favourite bookseller for around £35, bound so it will stay open when lying on your desk. Not a hard decision, is it?

Words like "unconscionable delay" and "overpriced" just naturally seem to turn up in the same sentence with "ISO" far too often. Ecma, on the other hand, prides itself on being able to produce standards on a fast-paced schedule and then gives them away free. However, Ecma achieves these goals by restricting participation in standards development to the small group of companies who are willing to pay its high membership fees. According to its web page, it is "driven by industry to meet the needs of industry ", unfortunately encouraging little or no input from the wider community. For something like a standard for video card drivers, such an approach can be justified – the number of people seriously interested in the technical aspects is dwarfed by the number of people who could benefit from a standard being developed and adopted quickly.

But when it comes to a standard for a programming language, however, the list of stakeholders is much broader than just compiler vendors, and many necessary tradeoffs must be thoughtfully deliberated.

Recently Ecma has launched a new project to develop a C++ "binding" for the Common Language Infrastructure underpinning the .Net environment. (CLI is already an Ecma standard.) The announced intent is not to change anything in standard C++ but to add some extensions which make C++ a first-class language for developing programs in that managed environment (Managed C++ was a rough draft). Amazingly, no thought was given to inviting WG21, the international ISO C++ committee, to collaborate in this effort, even though the ultimate aim of the Ecma project is to have the document adopted as an ISO standard through the fast-track process.

Ecma is the inventor and main practitioner of fast-tracking, having steered almost 200 standards through ISO/IEC since 1987. However, some people are starting to think that Ecma is abusing its liaison status with all these fast-track standards, that it is deliberately bypassing the consensus-based process which makes ISO standards so highly respected.

But because the C++ committee is so large and so active within the ISO standards world, perhaps this is an opportunity to change things for the better. Some of us are vigorously campaigning for at least a liaison arrangement to be set up between WG21 and Ecma's TG5, the new group. The objective is to blend the nimbleness of the Ecma process with the broader consensus model of ISO standardization, thus producing a document that all parties can happily endorse and avoiding any appearance that a few vendors are able to dominate its evolution.

If you would like to express an opinion on this issue, please write to standards@accu.org. I will see that it gets passed on to the people involved.

The press release announcing the Ecma C++ project is at:

http://www.ecma-international.org/news/ecma-TG5-PR.htm

Dialogue

Student Code Critique Competition

Set and collated by Francis Glassborow

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 24

The problem

Curious what novices set out to do but that should not stop us from trying to help them achieve their objectives. I suspect that the writer of this is trying to grab the console output and paste it into a file. Now we know what he should be doing, but while helping him try to raise the code quality.

I want to copy my C++ program output to an ASCII text file

I store my data in arrays and then use `cout` to output the data in DOS... as I generate data in the order of 1000's ...I am having a problem with physically copy pasting the data into notepad (an MS Windows pure text editor).

Here is my code...please suggest what changes are necessary. My coding is not really that efficient. Please bear with me

```
# include<iostream.h>
# include<conio.h>
# include<math.h>
# include<iomanip.h>

void main(){
  void clrscr();
  int i,r,j,m,x;
  static Sum[5000][5000];
  Sum[1][1]=25000;
  Sum[2][1]=35000;
// input the data into arrays
  for(j=2; j<=11; j++)
    for(i=1; i<=pow(2,j); i++) {
      if(i==1) {
        Sum[i][j] = Sum[i][j-1] + 25000;
        Sum[i+1][j] = Sum[i][j-1] + 35000;
      }
      if(i>1) {
        Sum[i][j] = Sum[r][j-1] + 25000;
        r=r+1;
        Sum[i+1][j] =
             Sum[(i+1)/2][j-1] + 35000;
      }
      i=i+1;
      r=2;
    }
// output the data
  for(m=2; m<=11; m++) {
    for(x=1;x <= pow(2,m); x++) {
      for(i=1; i<=2; i++) {
        cout << x << m
             << setw(10) << Sum[x][m] << '\n';
      }
    }
    getch();
  }
}
```

My output reads:

```
  11 25000
  11 25000
  12 35000
  12 35000
```

so on..so forth

My Comments

This time our esteemed editor does not have to spend valuable time offending those whose submissions are considered inferior to the winner's because there was only one submission, which by definition, has to be the winner. That is sad because I guess that many more of you could have had a go. Yes, many winners write well but so do many who do not win and as I have written before, it is participating that is the biggest reward. In order to provide an alternative I asked my friend who writes as the Harpist to have a look and provide his thoughts.

The Harpist's Response

I notice that in setting the problem Francis asked that we focus on helpful advice to the author. So here goes:

A Clear Statement of Intent

Before writing a single line of code you need to have a clear idea about what you are trying to produce. In this case it seems that the objective is to produce a file of values generated by some form of generator. I am certain that the actual code does not do what the programmer thinks because it is littered with quirky bits. If it did actually generate the desired values it would be pure chance.

Understand Your Tools

Fundamentally there are two ways to create an output file. The first way is to have the program create the required file. The second way that is often forgotten is to simply capture the program output into a file. What you do not do (or only very rarely) is to capture the output from the screen and paste it into a file. It seems that the student has been trying to use this last method.

Write Portable Code

There are times when we have to fall back on implementation specific features and library functionality but they are much less common than most students believe.

Never Do Things Till You Have To

That almost speaks for itself so I will just leave it as a thought to put alongside 'Do things in time.'

Applying These Principles

Here is my rewritten code up to the student's first comment:

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

int main() {
  static int sum[5000][5000];
  sum[1][1]=25000;
  sum[2][1]=35000;
```

Two of the header files go. We do not need `conio.h` because we are not going to use the screen as some intermediary for getting the output where we want it. Looking forward I realise that the programmer only needed `math.h` to give him access to the declaration of `pow()`. Using that function to compute successive powers of two is overkill. It is the kind of overkill that inexperienced programmers are very prone to. They think 'power of two' and forget that what they needed was 'next power of two' which is simply two times the current one.

While I was about it I replaced the header files with the equivalent Standard C++ headers.

Next I tidied up the declaration of `main()` to return the required `int`, and threw out the definitions of `i` etc. because this is far too early to be

declaring single letter variables (I have nothing against them but they should always be declared in the context of their use.) Even C now allows that. `clrscr()` went because it serves no useful purpose.

I think the student was getting away with an implicit `int` in the declaration of `Sum`, which I have amended to `sum` because I cannot abide gratuitous use of upper case. In case you are wondering, removing the `static` storage class specifier would result in undefined behaviour because the array would not have been initialised. Of course there are other ways of ensuring that an array is fully initialised.

In passing, I feel profoundly uncomfortable with that requirement for storage for twenty-five million `int`s. I am sure that a correct analysis of the required computation will demonstrate that a much less memory demanding approach is viable.

Now let me look at the computation section. This definitely can do with some restructuring because, frankly, it is currently a hacked together mess.

```
int i_limit = 4;
for(int j=2; j != 12; ++j, i_limit *=2) {
  sum[1][j] = sum[1][j-1] + 25000;
  sum[2][j] = sum[1][j-1] + 35000;
  for(int i=3; i<= i_limit; i += 2) {
    sum[i][j] = sum[2][j-1] + 25000;
    sum[i+1][j] = sum[(i+1)/2][j-1] + 35000;
  }
}
```

Where has the `r` gone? Careful analysis of the control flow of the original code showed that `r` was always 2 when used. I find that very suspicious and think that it probably should have been equal to `i`. However the point I would make to the student is that by cleaning up the code we can see what is happening. This will make it much easier to correct errors in the code. That '2' where `r` was sticks out like a sore thumb. Indeed I suspect that the correct code should be:

```
int i_limit = 4;
for(int j=2; j != 12; ++j, i_limit *=2) {
  for(int i=1; i<= i_limit; i += 2) {
    sum[i][j] = sum[i][j-1] + 25000;
    sum[i+1][j] = sum[(i+1)/2][j-1] + 35000;
  }
}
```

Once we have it down to this level, we can see that the original dimensions of `sum` are ridiculous. In fact he has 10 sets of results, each set solely dependant on the immediately prior set. Allowing indexing from 1 rather than 0 and not even attempting to optimise storage we get:

```
static int sum[2049][12];
```

as being entirely sufficient. This reduces the storage requirement to something reasonable on most hardware (around about a hundred thousand bytes on a platform using 4-byte `int`s.

Finally let me look at the output section. This also has its oddities. Here is my rewritten version:

```
i_limit = 2; // reset to initial value
for(int j=1; j != 12; ++j, i_limit *= 2) {
  for(int i=1; i <= i_limit ; ++i){
    std::cout << i << j << std::setw(10)
              << sum[i][j] << '\n';
    std::cout << i << j << std::setw(10)
              << sum[i][j] << '\n';
   }
}
```

Now why the student wants to output the values of `i` and `j` without an intermediate space, and why he wants to output each result twice will for ever remain a mystery but the process of cleaning up his code makes it quite clear that that is exactly what he does. By the way, for consistency I changed the output loop so that it will display the initial data as well as the computed data.

Now I suspect that all the code mangling came about while the student struggled to solve the problem of getting the output into a text file. His mind was so focused on trying to cope with this that everything else went up in flames. So how to get the data into a file? Simply, invoke the program with:

```
program > data.txt
```

Which I think makes it clear that the student actually needed a very different answer from the one he was asking for.

Student Code Critique 24 from Roger Orr

<rogero@howzatt.demon.co.uk>

"Hmm", said Bill, "I wonder what this program does. Any ideas, Jim?"

Jim came over and stared at Bill's screen. "Well", he said, "all the user wants is to be able to copy the output to an ASCII text file. Does it matter what it does?"

Bill thought a moment and then said, "Yes, it does matter – partly for professional pride but also to try and understand what he means here where he writes 'my coding is not really that efficient'".

Jim looked puzzled and tentatively suggested that it didn't matter – surely the code was just a bit slow and so what it was doing was irrelevant since you obviously don't need to understand the purpose of code to make it faster.

"No, I'm afraid you've missed the point of my remark– efficiency is not necessarily anything to do with speed. In this case the user is currently pasting data into notepad – the speed of the program is a tiny part of the overall time. I suspect the user is actually worried about either the inefficient use of memory or the amount of effort it took turn the original formula into code. Look, I'll show you what I mean."

Bill turned to the keyboard, compiled the sample code and ran the program. There was a long pause.

"You're wrong", said Jim, "look how long it's taking to get anywhere – what *is* the program doing?" Bill simply pointed to the CPU meter, which was showing almost zero, and said "No, it's not busy – what is it waiting for? …Ah of course, silly me."

With that he leaned on the space bar and almost immediately the screen filled with numbers.

"The problem is the mix of I/O since the output uses the C++ streams library – well, an old one anyway – which buffers up output to write many characters at once for speed, but the input uses a non-standard low-level C runtime call. It isn't a good idea to pair up buffered output and unbuffered input. It looks like the user really wanted to all the pending output to appear each time before waiting for a keypress. Oh well, so much water under the bridge – we want to write the entire output to an ASCII text file now so there's no need to wait – I'll remove the `getch()` call and the `#include` of `conio.h`".

Bill applied himself to the keyboard, and made these two changes. He then noticed that the '`clrscr()`' function was not in fact needed, so removed that line too. "Just reversing entropy", he muttered to himself, "gets in everywhere doesn't it?"

He then decided to bring the program into the world of the C++ standard. This meant making '`void main()`' become '`int main()`' and then changing the old includes '`iostream.h`' and '`iomanip.h`' to the new ones '`iostream`' and '`iomanip`'. Of course, the standard C++ streams library is in the namespace '`std`' so he had to added '`std::`' before '`cout`' and '`setw`'.

"Ok, let's try that", he said, "Oh … our output doesn't match what the user says he gets – odd. Perhaps I've broken something….no – look – the output loop starts with `m=2` but the user's output starts with '`11`'. Looks like the code he sent us doesn't match what he's actually using. Not the first time, eh? Now we really *do* need to know what the program does so we can decide whether the program or the sample output is wrong – or both of course! Ok, for now we'll change the second loop to start with `m=1` and now the results do look like what the user wants. Hey, look here – this variable '`r`'. The first time round the input loop it isn't used, just set to 2. All the other times round the loop it is used, then incremented, and then reset to 2. I wonder if that is really right? For now I'll put a warning comment in the margin."

"Right, now we're ready. Here goes". Bill opened a command prompt and typed:

```
scc24 > x.txt
```

"That's got the output into a text file", he said.

"Isn't that cheating – doesn't the user want to do that with C++?", Jim objected.

"He is using C++", said Bill, "and we're simply using the features of his chosen environment. KISS."

"Pardon??"

"Keep It Simple, Stupid", explained Bill, "Why write code when you don't have to? Let the operating system solve that part of the problem! Now let's send the user the tidied-up program and instructions on how to run it."

The Winner of SCC 24

The editor's "choice" is:

Roger Orr

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

[SCC 25 at foot of next page]

Francis' Scribbles

Francis Glassborow

C & C++ Standards in Hardcopy

I was sitting in a meeting room just outside Nice in March 1998 and the topic was what we should do with the Standard that we had just finished. I proclaimed that apart from anything else I would get it published through BSI for £25.

That bold statement ruffled a few feathers higher up because it was not long after that ANSI made a pre-emptive strike and provided the C++ Standard in PDF format for $18.

Getting BSI to realise that the only way it would get any real income from either the C++ Standard or the coming new C one was to use a professional publisher and obtain some royalties was immensely harder than I had naïvely expected.

Even when we had got tentative agreement other events intervened, such as having the Wiley editor handling the project be made redundant. However with the help of other ACCU members, in particular Lois Goldthwaite, we have finally completed the project for both C and C++. The cover price isn't the £25 I was aiming at but if you go to `www.amazon.co.uk` you will find they are offering both volumes at a discounted price of £24.47 each. That means that for an investment of under fifty pounds (including postage) those living in the UK can have two excellently bound (the pages do not turn with a will of their own) books that contain the most up-to-date versions of the C and C++ Standards (both include the latest Technical Corrigendum folded into the text). As a bonus, the C Standard also includes the current Rationale that explains the decisions that WG14 made when changing the original 1989 standard to the new 1999 one.

Those living in the US might check `www.bookpool.com`, which seems to give better deals on computer books than `www.amazon.com`.

I hope ACCU members will give serious consideration to owning their own copy (or getting their employer to do so) of either or both Standards. When thinking about it you might note that despite being in a privileged position (i.e. entitled to free electronic copies of the relevant Standard) many members of WG14 & WG21 have ordered one or more printed copies (yes, they did get a heavily discounted special offer but even so they recognised that the printed version would assist them.)

Decimal Floating Point

Most of us know that the C and C++ Standards allow a degree of latitude in the representation of floating point types. In addition you probably know that the two commonest radices used are two (binary) and sixteen (hexadecimal). While ten is allowed it is rarely, if ever, used.

Some of you may know that the use of radices that are not capable of exactly expressing decimal values is very problematical in some application domains. The whole financial sector is hemmed in by national legislation requiring levels of accuracy that are very difficult to achieve unless calculations are done in base ten.

Historically hardware has been based on binary (which, of course, makes octal and hexadecimal straightforward) because it was immensely easier to design and implement electronic circuits in terms of two-valued systems. However this incurs a steep penalty because providing exact decimal arithmetic requires substantial software.

For example comparing computation using a normally coded (binary) double with the equivalent computation done with an exact decimal type (provided by software) produces performance factors of between 100:1 and 1000:1. In numerically intense computations for accounting purposes those figures really hurt. Accounting packages typically spend between seventy and over ninety percent of their execution time in decimal evaluations. If that could be moved from software emulation to hardware decimal floating point units we could achieve a net performance gain by a factor of over three. This is in an industry where a performance improvement of ten percent is considered a major breakthrough.

One of the biggest players in the financial world is IBM so it is not surprising that they have both recognised the problem and come up with proposals to tackle it with specialised decimal FPUs.

That is fine, but to profit from this we need language facilities. When we look at the problem in more detail it seems that supporting such things as IEEE 754R requires more than simply moving to using a radix of 10 in our existing floating point types.

IBM recognised this and came to WG14 (C) via the BSI C Panel with a request that they (WG14) started a work project to investigate how best to support the planned new hardware. At that time I strongly recommended that WG21 (C++) were approached in the same time frame. Even though time was short, IBM were persuaded and briefed its representatives on WG21 to present the case there as well.

Both Committees were convinced that they needed to consider how best to support the coming hardware. Both Committees were initially sceptical that much needed to be done but both have been convinced that there is more to it than a first glance might suggest.

Please note that neither Committee thinks it knows the right way to use the coming hardware. Both recognise that they have work to do to build the best possible language framework on top of both the coming hardware and the already existing ISO, IEEE and ECMA standards.

It is what happened next that is quite exceptional and firmly gives the lie to those who claim that WG14 and WG21 are in some way competitors.

Student Code Critique 25

(Submissions to `francis@robinton.demon.co.uk` by Jan. 14th)

It is always worth sending in a late entry, depending on my workload it may or may not be considered for the prize but it will eventually get published.

This time I am presenting two short C programs. Please attempt to critique either or both of them.

Program 1

I am little confused about return values of `sizeof` operator.

Here is a simple C program which is putting me in doubt:

```
#include <stdio.h>
int main() {
  int a[10];
  int *p =(int *) malloc (10);
  printf("Size of a = %d \n",sizeof(a));
  printf("Size of p = %d \n",sizeof(p));
}
```

Output is :

```
Size of a = 40
Size of p = 4
```

My understanding says even array name is a pointer. If so why it does not show `sizeof(a)` as 4? Or if `sizeof` shows the total allocated memory then why `sizeof(p)` does not show 10?

There are numerous errors in both the code and the student's understanding. Please address these comprehensively, perhaps including places where your explanation would be different if this were a C++ program.

Program 2

I am getting an error linker error. Here is the code:

```
// Define Libraries
#include <stdio.h>
#include <math.h>
//Start Of Main Program
main(){
  double hypo, base, height;
/* Enter base and height */
  printf("Enter base:");
  scanf("%f", &base);
  printf("Enter height:");
  scanf("%f", &height);
  hypo = sqrt(pow(base, 2)+pow(height, 2));
  printf("hypotenuse is %f", &hypotenuse);
}
```

Ignore the question asked by the student and address the serious problems with the code itself.

ISO and IEC singularly fail to provide any mechanism by which two work groups can directly cooperate on a single project yet such is the desire of the two Committees to work together on this issue that they searched around for a mechanism.

The agreement is that both WG14 and WG21 will submit identical work requests to their parent committee, SC22 and nominate the same editor. They then plan to do the actual work via a Rapporteur group. This is a mechanism that allows non-committee experts to participate freely. Each WG's Rapporteur group will meet with those from the other WG as technical experts.

It really takes highly motivated and creative programmers to come up with a solution to side step the constraints of bureaucracy. Where there is a will there is a way. And I am very happy that the will is there and do not ever let anyone get away with claiming that WG14 and WG21 cannot work together, or even do not want to. Clearly where there is shared interest they can and will.

If you want to know more about IBM's proposals, the background and existing work in the area try starting at:
`http://www2.hursley.ibm.com/decimal/`

More Cooperation

As if agreeing to cooperate on one major item were not enough WG14 noted that the current C++ Library Technical Report included support for a range of special maths functions (such as gamma). If C were to support these there are various issues that would have to be addressed because of the lack of overloading, default arguments etc. in C.

It was clear that the reasons for C++ including these special functions were also valid in C, but it was also clear that C should include them in a way that was entirely compatible with C++. WG14 requested through its extended liaison with WG21 that the necessary work be done between the Committees so that this part of the C++ Library TR could be paralleled by a C Library TR. Not only was this agreed but certain changes were made to the C++ TR in order to ensure that C could provide maximal compatibility.

WG21 Meets Some Users

Tom Plum, our host for the recent WG21 meeting in Hawaii, arranged a trip to the high altitude observatories. Unfortunately I had to miss it because I was suffering from a serious cold (not a good idea to go to 13500 feet in such circumstances) but a group of those who did go had lunch with some of the astronomers in order to discuss how C++ might help with their software.

Those that went tell me that it was an interesting meeting. Many members on the WG21 side had not previously met technical professionals for whom programming was a necessary part of their work whilst being entirely a chore and incidental to their main interests. These are the kind of programmer who can be forgiven for not reading books and magazines about programming. Just as many of us probably have wild misconceptions about astronomical tools (garnered from popular publications) they had wholly inaccurate views of what might be achieved through using C++. They do have real needs for high performance software and had come to believe that that meant they needed to use C rather than C++. On the other side they needed software that would work happily over the Internet and believed that Java was their principal option.

The astronomers went away with at least a sense that modern C++ might help them. WG21 went away with an understanding of how important it was to provide first class special purpose libraries that would allow such incidental programmers to use the full power of C++ without having to become expert at all the dirty detail.

Kona in Retrospect

Kona on the Big Island of Hawaii is a beautiful location in which to work (and I can assure you that both WG14 and WG21 members who were there worked very hard) but this particular set of meetings seems to have been productive in subtle ways that I think will become more evident in future years.

For me personally it will always be tinged with a little sadness because on the Sunday morning between the meetings as I was preparing to have a relaxing day visiting places that I have become fond of in previous visits my son rang to tell me that my mother had died. Actually for her that was good news because the last couple of years of her life have been ones of great pain both mental and physical. However losing a greatly loved parent hurts however much one is prepared for it.

My Book

My book is due out in early December so make sure you recommend it to all those friends and relatives who keep asking you to explain what it is that you do.

Problem 12

Please look at the following two functions that are intended to save and restore the red, green and blue intensities of a palette of 256 24-bit colours. Actually you do not need to know the gory details, all you need to know is that the first function writes some data to a file and the second is supposed to restore it. What is wrong?

```
void save_palette(playpen const& canvas,
                  string filename) {
  ofstream out;
  open_ofstream(out, filename);
  if(out.fail())
    throw problem("Could not open file");
  for(int i(0); i != 256; ++i) {
    HueRGB const mix(canvas.get_entry(i));
    out << int(mix.r) << " "
        << int(mix.g) << " "
        << int(mix.b) << '\n';
  }
}
void restore_palette(playpen & canvas,
                     string filename) {
  ifstream in;
  open_ifstream(in, filename);
  if(in.fail())
    throw problem("Could not open file");
  for(int i(0); i != 256; ++i) {
    canvas.set_entry(i,
                     HueRGB(read<int>(in),
                            read<int>(in),
                            read<int>(in)));
  }
}
```

Commentary on Problem 11

The following is a template function to extract a value from an input stream.

```
template<typename in_type>
in_type read(std::istream & in) {
  in >> temp;
  if(in.fail() and not in.eof())
    throw fgw::bad_input("Corrupted data");
  if(not in.eof())
    return temp;
  else ???
}
```

The problem is that there are two distinct circumstances in which the `eof` flag may be set. There are also other problems with the above code, such as the use of an undeclared variable.

In the first it happens as part of the process of successfully reading the last item of data. This is the case where the input stream does not include a final carriage return. Clearly in this case we must not throw an exception because the wheels have not come off yet.

The second case is the more normal one where the last entry in the input stream is followed by a carriage return. What happens is that the function tries to extract an item of data but fails and as part of the process sets the `eof` flag. The user cannot know this will happen (unless they write code that looks ahead which would sort of reduce the usefulness of the template.)

Typically the user wants to be able to write something such as:

```
while(true) {
  int value(read<int>(infile));
  // process result
  if(infile.eof())
    break;
}
```

We will see that we cannot quite achieve that simplicity but we can get close.

The problem case is the second of the two above. I do not want to throw an exception for a variety of reasons, not least that reaching the end of a file is not exceptional. That means that if an attempt to extract an item fails because the end of file has been reached we need to return some dummy value to give the user a chance to detect both the end of file condition and that the returned value is just a place holder rather than the last item in stream.

I want to allow users to provide the dummy value, but not require that they do. That makes me consider using a parameter with a default argument. Consider this code:

```
template<typename in_type>
in_type read(std::istream & in,
             in_type dummy = in_type()) {
  in_type temp;
  in >> temp;
  if(in.fail() and not in.eof())
    throw fgw::bad_input("Corrupted data");
  if(in.fail()) return dummy;
  return temp;
}
```

This is not very different to the original but that default argument has an added advantage in that it makes it clear that the function only works with types that are default constructible.

Notice how the last three statements progressively eliminate the special cases. If you are one of those who cannot abide multiple return statements then the code is easily fixed up to cater to your prejudices:

```
return in.fail() ? dummy : temp;
```

Now let me view this from the user's perspective. The following will work quite safely even if it might process one too many elements:

```
while(not infile.eof()) {
  int item(read<int>(infile));
  // process item
}
```

The programmer can do a better job with:

```
while(not infile.eof()) {
  int item(read<int>(infile));
  if(not (infile.eof() and (item == int()))) {
    // process item
  }
}
```

Note that the right hand operand of the `and` is only evaluated if the end of file flag is set. This means that, in general, the more expensive expression is only evaluated if it is needed to make the decision.

I would be interested in reader feedback on this because it is part of an ongoing process of refining a useful and safe mechanism for extracting data from input streams. I think that having such a mechanism is important because it makes C++ more accessible to novices.

Cryptic Clues for Prizes

Last time I set you the following little problem concerning 'Greek' style clues. Unfortunately I miscounted on my fingers and gave you the wrong number (I meant to give you 271, that is covered by 'apt', 'tap' and 'pat'). 261 only provides 'oat' and 'Tao'. Perhaps this error confused you into thinking you did not understand the problem. I hope that was the reason that I have not had any responses because otherwise it says little for the creativity of my readers. Anyway here are a couple of possible clues for 261:

> The result if Tao were valued in Greece.
> Counting wild oats in a singularly Greek fashion.

And for the intended value (271):

> A Greek tap is particularly apt in a numerical way.
> A Greek tap is numerically indistinguishable from a pat on the head.

[Based on: A-J representing 1 to 10, J-S representing 10 to 100, S-Z representing 100 to 800.]

Christmas Competition

Happy Christmas, or for those of the Jewish faith Hanukkah, and for the Chinese, Dõngzhì and a greatly belated happy Id al-Fitr for those following an Islamic calendar. For the rest, have a Happy Winter Solstice.

Now while you are enjoying some seasonal relaxation have a look at a mobile phone and note that this provides yet another way of relating numbers to letters.

Because this is a special competition, any form of clue can be submitted but the solution must give 4277919627.

Deadline for submissions: January 30.

Francis Glassborow

Write for ACCU!

Pete Goodliffe <pete@cthree.org>

Your name in lights

ACCU membership is not just a magazine subscription; your subscription fee entitles you to more than just a few magazines a year. It's the opportunity to contribute, to see your name in lights, and to share your thoughts with the development community. In fact, if you don't then the ACCU journals will die.

We need members to write articles. And that means you! If you enjoy reading C Vu and Overload, and want to see them continue then please get writing something. You don't have to have been published before – new writers are positively welcomed.

Prizes! Prizes! Prizes!

Not only will you develop your own skills and have something excellent to put on your CV, you'll also stand the chance of winning one of the new ACCU awards. This year the ACCU is awarding prizes for published articles in a number of categories. These are:

- Best C Vu article
- Best Overload article
- Best article by a new writer

The awards will be announced at next year's AGM and the prize will be an ACCU T-shirt and untold acclaim. So what are you waiting for?

What to write?

We need articles at *all levels*, and on a wide range of topics. This means that you *already* have an article in you somewhere, something valuable that ACCU members want to read about. Really, you do. What are you doing right now? What do you know about?

Here is another selection of title suggestions. For more inspiration look at the lists in back issues.

- **.NET experiences**
- **1st steps with the STL**
- **Creating interfaces with Qt**
  Or any other toolkit
- **Writing mobile applications**
- **The tools I can't live without**

Don't hold back

Don't think you have nothing to write; you do. Don't think that your writing skills aren't up to it; try to sketch something out and you'll probably surprise yourself. We have a team of friendly people who'll help to craft your submission into the final printed copy, so you won't be on your own.

How to submit

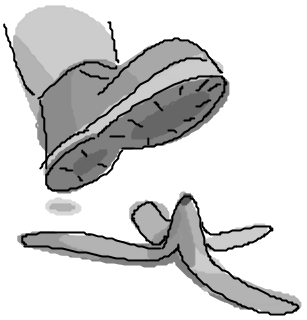
You can send submissions by email to `editor@accu.org`. Plain text is perfectly acceptable; there is a Word document template you may wish to use if you want to retain formatting. That's all there is to it – ***get writing***.

Features

Professionalism in Programming #23

To err is human

Pete Goodliffe <pete@cthree.org>



> We know that the only way to avoid error is to detect it, that the only way to detect it is to be free to inquire.
>
> J. Robert Oppenheimer

At some point in their life everyone has this epiphany: *the world doesn't always work as you expect*. My one-year old friend Tom learnt when climbing a chair four times his size. The equal and opposite reaction came as quite a shock; he ended up under a pile of furniture.

Is the world broken? It is wrong? No. The world has happily plodded along in its own way for the last few million years, and looks set to continue for the foreseeable future. It's *our expectations* that are wrong and need adjustment. As they say: *bad things happen, so deal with it*. We must write code that deals with the Real World and its unexpected ways.

This is particularly difficult because the world *mostly* works as we'd expect, constantly lulling us into a false sense of security. The human brain is wired to cope, with inbuilt fail-safes. If someone bricked up your front door you'll stop automatically, before walking into an unexpected wall. Programs are not so clever; we have to tell them what to do when it all goes wrong.

From whence it came

> To expect the unexpected shows a thoroughly modern intellect.
>
> Oscar Wilde

Errors can and will occur. In a large program, an error may occur for one of a thousand reasons. But it will fall into one of these three categories:

User error

The stupid user manhandled your lovely program. Perhaps they provided the wrong input, or attempted an operation that's patently absurd. A good program will point out the mistake, and help the user rectify it. It won't insult them, or whinge in an incomprehensible manner.

Programmer error

The user pushed all the right buttons, but the code's broken. This is a *bug*, a fault the programmer introduced that the user can do nothing about (except try to avoid it in the future).

Exceptional circumstances

The user pushed all the right buttons, and the programmer didn't mess up. Fate's fickle finger intervened, and we ran into something that couldn't be avoided. Perhaps a network connection failed, we ran out of printer ink, or there's no hard disk space left. These are the most common forms of errors and, unfortunately, the hardest to deal with.

Each of these error categories has a different audience. Users don't want to be bothered by programmer errors – there's nothing they can do about them anyway (short of never using that piece of tatty software again).

In our code we need a well-defined strategy to manage each kind of error. An error may be detected and reported to the user in a pop-up message box, or it may be detected by a middle-tier code layer, and signaled to the client code programmatically. At both levels the same principles apply. It may be a human choosing how to handle the problem, or lower down the food chain it's your code making a decision – someone is responsible for acknowledging and acting on errors. Errors are generally raised by subordinate components and communicated upwards, to be dealt with by the caller.

Errors are reported in a number of ways; we'll look at these in the next section. To take control of program execution we need to be able to:

- raise an error when something goes wrong,
- detect all possible error reports,
- handle them appropriately, and
- propagate errors we can't handle.

Each of these tasks are addressed in the subsequent sections.

Errors are hard to deal with. The error you encounter is often not related to what you were doing at the time (they are mostly 'exceptional circumstances'). They are also tedious to deal with – we want to focus on what our program *should* be doing, not on how it may go wrong. However, without good error management your program will be brittle – built upon sand not rock; at the first sign of wind or rain it will collapse.

Take error handling seriously. The stability of your code rests upon it.

Error reporting mechanisms

There are several common strategies for propagating error information to client code. You'll run into code that uses each of them, so you *must* know how to speak every dialect. Some reporting mechanisms are particularly suited to certain languages or operating environments, as we'll see. This list contains nothing unexpected, but we should take time to understand how the techniques compare, and which is most effective for any given situation.

None

The simplest error reporting mechanism is: *don't bother*. This works wonderfully in the cases where you want your program to behave in bizarre unpredictable ways, and crash randomly.

If you encounter an error and don't know what to do about it, blindly ignoring it is not a viable option. You probably can't continue your work, and returning without fulfilling your function's 'contract' will leave the world in an undefined and inconsistent state.

Never ignore an error condition.

If you don't know how to handle the problem, then signal a failure back up to the calling code. Don't sweep an error under the carpet and hope for the best.

An alternative to ignoring errors is to instantly abort the program upon encountering a problem. It's easier than handling errors throughout the code, but hardly a well-engineered solution! *[Though for some situations it can be the best solution – JAD]*

Return values

The next most simple approach is to return a success/failure value from the function. What that value gets set to depends on: the number of ways a function may fail, the diligence of the programmer, and any project conventions. Most functions will return a boolean value, a simple yes or no answer. More advanced mechanisms enumerate the possible ways a function can exit and return a status value to signify this, known as a *reason code*. One of those reason code values will mean 'success', the rest represent the many and varied abortive cases. This enumeration may be shared across the whole codebase, in which case your function returns a subset of the available values. You should therefore document what the caller can expect.

Whilst this approach works well for procedures that don't return other computed values, passing error codes back *with* returned data gets messy. For example, if you have a function `int count()` that walks down a linked list and returns the number of elements, how can it signify a list

structure corruption alongside that `int` return value? There are three approaches:

- Return a compound data type (or *tuple*) containing both the return value and an error code. This is easier in some languages than others. Whilst reasonably elegant, this technique is seldom used, it is rather clumsy in the popular C-like languages.
- You could pass the error code (and/or the original return value) back through a function parameter. In C++ this parameter would be passed by reference. In C you'd indirect the variable access with pointers. This approach is ugly and non-intuitive; there is no syntactic way of distinguishing a return value from an error code.
- The alternative is to reserve a range of the return values to signify failure. The `count` example above could nominate all negative numbers as error reason codes. They'd be meaningless answers, anyway. Negative integer values are a common choice for this. Pointer return values may be given a specific 'invalid' value, which by convention is zero (or `NULL`). In Java you could return a `null` object reference.

  This technique doesn't always work well. Sometimes it's hard to reserve an error range – all return values are equally meaningful and equally likely. It also has the side effect of reducing the available range of 'good' values; the use of negative values reduces the possible positive values by an order of magnitude[1].

Error status variables

This approach attempts to manage the contention between a function's return value and its error status report. Rather than return a reason code the function sets a shared, global, error variable. After calling the function you must then inspect this status variable to find out whether it completed successfully or not.

The shared variable reduces confusion and clutter in the function's signature, and doesn't restrict the return value's data range at all. However, errors signaled through a separate channel are much easier to miss or to wilfully ignore. A shared global variable also has some nasty thread safety implications.

The C standard library employs this technique with its `errno` variable – it's a good example of why error status variables are a bad idea. Its use is fraught with peril, nowhere near as simple as a function return value. You must clearly understand the semantics of its operation: before using any standard library facility you have to manually clear `errno`. They never set a 'succeeded' value to `errno`. This is a common source of bugs, and makes calling each library function more tedious. To add insult to injury, not all C standard library functions use `errno`, so it is less than consistent.

A shared error variable is functionally equivalent to the previous reporting mechanism, but has enough disadvantages to make you avoid it. Don't write your own error reporting mechanism this way, and use existing implementations with the utmost care.

Exceptions

Exceptions are a structured language-level facility to manage errors. They help to distinguish the 'normal' flow of execution from 'exceptional' cases. When your code encounters a problem it can't handle at that point, it stops dead and throws an error message up in the air. The language runtime then automatically steps back up the call stack until it reaches some handler code. The error message lands there, and the program gets a chance to handle the problem.

There are two operational models:

- the *termination model* (provided by C++ and Java), where execution carries on after the *handler* code that caught the exception, and
- the *resumption model*, where execution carries on from where the exception was raised.

The former model is easier to reason about, but it doesn't give ultimate control. It only allows fault handling (you can execute code when you notice an error), not fault rectification (a chance to remove the cause of the problem and try again).

An exception cannot be ignored. If it isn't caught and handled, it will propagate to the very top of the call stack and stop the program. The language runtime automatically cleans up as it unwinds the call stack. This makes exceptions a tidier and safer alternative to handcrafted error handling code. However, throwing exceptions through sloppy code can lead to memory leaks and problems with resource cleanup[2]. You must take care to write *exception-safe* code. The sidebar explains what this means in more detail.

Code that handles an exception is distinct from the code that raises it, and may be arbitrarily far away. Exceptions are usually implemented in OO languages, where error messages can be defined by a hierarchy of exception classes. A handler can elect to catch a quite specific class of error (by accepting a leaf class), or a more general category of error (by accepting a base class). Exceptions can be the only error reporting mechanism available in certain situations – how else can you signal an error in a constructor?

Exceptions don't come for free; the language support incurs a performance penalty. In reality this isn't very significant, and is only ever seen in the presence of the exception handling statements. Exception handlers reduce potential for optimisation opportunities. Throwing an exception is an expensive operation, so they should only be used for genuinely *exceptional* events.

Signals

Signals are a more 'extreme' reporting mechanism – largely used for errors signaled by the execution environment to the running program. The operating system traps a number of exceptional events, like a *floating point exception* triggered by the maths coprocessor. These well-

Whistle-stop tour of exception safety

In languages without automatic cleanup facilities, resilient code must be *exception-safe*. It must work correctly no matter what exceptions come its way, for some definition of 'correctly' (we'll define this below). It doesn't matter whether the code handles any exceptions or not.

Exception neutral code propagates *all* exceptions up to the caller; it won't consume or change anything. This is an important concept for 'generic' programs, like C++ template code – the template types may generate all sorts of exceptions that template implementors don't understand.

There are several different levels of exception 'safety'. They are described in terms of guarantees to the calling code. These guarantees are:

Basic guarantee

If exceptions do occur in a function (resulting from an operation we perform, or the call of another function) we will not leak resources. The code state will be consistent (i.e. it can still be used correctly), but will not necessarily be left in a known state. For example, a member function should add ten items to a container, but an exception propagates through it. We guarantee the container is still usable, but maybe no objects were inserted, maybe all ten were, or perhaps every other object was added.

Strong guarantee

This is far stricter than the basic guarantee. Here we ensure that if an exception propagates through our code the program state will remain completely unchanged. The object hasn't been altered, no global variables changed, nothing. In our example above, we can assert that no objects will have been inserted into the container at all.

Nothrow guarantee

The final guarantee is the most restrictive. We guarantee that an operation can *never* ever throw an exception. If we are 'exception neutral' then this implies that the function cannot call any function that itself might throw.

Which of the guarantees you provide is entirely your choice. The more restrictive the guarantee, the more widely (re-)usable the code is. In order to implement the strong guarantee you will generally require the use of a number of functions that provide the nothrow guarantee.

Most notably, every destructor you write should *always* honour the nothrow guarantee. Always. Otherwise all exception-handling bets are off. In the presence of an exception, object destructors will be called automatically as the stack is unwound. Raising an exception whilst handling an exception is not permissible.

1 If you used an `unsigned int` you'd have a power of two more values available, reusing the `signed int`'s sign bit.

2 For example, you could allocate a block of memory, and then exit early as an exception propagates through. The allocated memory would leak. This kind of problem makes writing code in the face of exceptions a more complex business.

defined error events are delivered to the application as a 'signal'. A signal interrupts the program's normal flow of execution and jumps into a nominated *signal handler* function. Your program can receive a signal at any time, and the code must be able to cope with this. When the signal handler completes, program execution continues at whatever point it was interrupted.

Signals are almost the software equivalent of a hardware interrupt. They are a Unix concept, now provided on pretty much every platform (a basic version is a part of the ISO C standard). The operating system provides sensible default handlers for each signal (some of which do nothing, others abort the program with a neat error message). You can override these with your own handler.

The defined signal events include: program termination, execution suspend/continue requests, and maths errors. Some environments extend the basic list with many more events.

Each of these mechanisms has different implications for the *locality of error*. An error is local in time if it is discovered very soon after it is created. An error is local in space if it is identified very close to (or even *at*) the site where the error actually manifests. Some approaches specifically aim to reduce the locality of error to make it easier to see what's going on (e.g., error codes). Others (like exceptions) aim to extend the locality of error so code doesn't get entwined with error handling logic.

The favoured type of reporting mechanism may be an architectural decision. It might be considered important to define a homogeneous hierarchy of exception classes, or a central list of shared reason codes.

Detecting errors

How you detect an error obviously depends on the mechanism reporting it. In practical terms, this means:

Return values

You determine whether a function failed by looking at its return code. This failure test is bound tightly to the act of calling the function; by making the call you are implicitly checking its success. Whether you do anything with that information is up to you. *[Though I am working on a proposal for C++ that would allow the author of a function to specify that the return type could not be ignored. – JAD]*

Error status variables

After calling a function which sets an error status variable, you must inspect this variable. If it follows C's `errno` model of operation you needn't actually test for error after every single function call. Reset `errno` first, then call any number of standard library functions back-to-back. Afterwards, inspect `errno`. If it contains an error value, then one of those functions failed. Of course, you don't know which one fell over, but if you don't care about that level of detail, this is a slightly streamlined error detection approach.

Exceptions

If an exception propagates out of a subordinate function, you can chose to catch and handle it, or to ignore it and let the exception flow up a level. You can only make an informed choice when you know what kinds of exception might be thrown. You'll only know this if it's been documented (and if you trust the documentation).

Java's exception implementation places this documentation in the code itself. The programmer has to write an *exception specification* for every method, describing what it can throw; it is a part of the function's signature. Java is the only mainstream language to enforce this approach. You cannot leak an exception that isn't in the list, the compiler performs static checking to prevent this from happening[3].

Signals

There's only one way to detect a signal: install a handler for it. There's no obligation. You can choose not to install any signal handlers at all, and accept the default behaviour.

As various bits of code converge in a large system, you will probably need to detect errors in more than one way, even within a single function. Whichever detection mechanism you use, the key point is this:

Never ignore *any* errors that might be reported to you.

If an error report channel exists, it's there for a reason.

When you let an exception propagate through your code you are not ignoring it – you *can't* ignore an exception. You are allowing it to be handled by a higher level. The philosophy of exception handling is quite different in this respect.

Even if you think that an error has no implication for the rest of your code, it is a good practice to write the detection scaffolding anyway, and to not take any action in the handler. This makes it clear to a maintenance programmer that you are fully aware how the function may fail, and you have consciously chosen to ignore any failures.

Handling errors

> Love truth, and pardon error.
>
> Voltaire

Errors happen. We've seen how to discover them, and when to do so. The question now is: what do you do about them? This is the hard part. The answer depends largely on circumstance and the gravity of an error – whether it's possible to rectify the problem and retry the operation, or to carry on regardless. Often there is no such luxury; the error may even herald the 'beginning of the end'. The best you can do is clean up and exit sharply, before anything else goes wrong.

To make this kind of decision you must be informed. You need to know a few key pieces of information about the error:

Where it came from (which is quite distinct from where it's going to be handled). Is the source a core system component, or a peripheral module? This information may be encoded in the error report, or else we know what function was called and can figure it out manually.

What you were trying to do. What provoked it? This may give a clue toward any remedial action. We probably only know this from our understanding of the error's context – you know what function was called. Error reporting seldom contains this kind of information.

Why it went wrong, and the nature of the problem. This only makes sense in the context of the error's source, and what was being done. You need to know exactly what has happened, not just a general hand-wavy *class* of error. It's important to know how much of the erroneous operation completed – *all* or *none* are nice answers, but generally the program will be in some indeterminate state between the two.

When it happened. This is the locality of the error in time. Has the system just failed, or has a two-hour old problem only just been spotted?

The severity of the error. Some problems are more serious than others, but when detected one error is equivalent to any other – we can't continue without understanding and managing the problem.

The level of severity is usually the caller's opinion, based on how easy it will be to recover or work around the error. If it's not a big deal, the strategy may just be to live with the problem. If it affects core functionality this isn't acceptable; the code must do everything possible to fix the problem and continue as if nothing happened.

How to fix it. This may be obvious (e.g., insert a floppy disk and retry), or may not (e.g., you need to modify the function parameters so they are consistent). More often than not we have to infer this knowledge from the other information collated.

Given this depth of information you can formulate a strategy to handle each error. Forgetting to insert a handler for any potential error will lead to a bug – it might be a hard to exercise bug, and hard to track down – so think about every error condition carefully.

When to deal with errors

So *when* should you handle each error? This can be separate from when it's detected. There are two schools of thought.

As soon as possible

Handle each error as you detect it. Since the error is handled near to its cause you retain important contextual information, making the error

3 C++ also supports exception specifications, but leaves their use optional. It's idiomatic to avoid them - for performance reasons, among others. Unlike Java, they are enforced at run time.

handling code clearer. This is a well-known self-documenting code technique. Managing each error near its source means there's less code which control passes through in an 'invalid' state; too much of that leads to very dense logic.

This is usually the best option for functions that return error codes.

As late as possible

Alternatively, defer error handling as long as possible. This recognises that code detecting an error rarely knows what to do about it. Often it depends on the context it is being used in: a missing file error may be reported to the user when opening a document, but silently handled when hunting for a preferences file.

Exceptions are ideal for this approach, you can ignore an exception until you know how to deal with the error. This separation of detection and handling may be clearer, but can make code more complex. It's not obvious that you are deliberately deferring error handling, nor is it clear where an error came from by the time you do handle it.

In theory, it's nice to separate 'business logic' from error handling. Often you can't, as cleanup is necessarily entwined with that business logic. It can be more tortuous to write the two separately. However, centralising error handling code has advantages: you know where to look for it, and can put the abort/continue policy in one place, rather than scattered through many functions.

Thomas Jefferson opined "delay is preferable to error". There is truth there, the actual *existence* of error handling is far more important than *when* an error is handled. Nevertheless, choose a compromise that's close enough to prevent obscure and out of context error handling, whilst being far enough away to not cloud 'normal' code with labyrinthine paths and error handling dead ends.

Handle each error in the most appropriate context, as soon as you know enough to handle it correctly.

This is usually the context that created the error.

Possible reactions

You've caught an error. You're poised to handle it. What are you going to do now? Hopefully, whatever is required for correct program operation. Whilst we can't possibly list every recovery technique under the sun, here are the common reactions to consider.

Logging

Any reasonably large project should already be employing a logging facility. It allows you to collect important trace information, and is an entry point for the investigation of nasty problems.

The log exists to record interesting events in the life of the program, to allow you to delve into the inner workings and reconstruct paths of execution. For this reason all errors you encounter should be detailed in the program log; they are one of the most interesting and telling events of all. Aim to capture all pertinent information – as much of the list above as you can.

For really obscure errors that predict catastrophic disaster, it may be a good idea to get the program to 'phone home' – to transmit either a snapshot of itself, or a copy of the error log, to the developers for further investigation.

What you do *after* logging is another matter...

Reporting

A program should only report an error to its user when it doesn't know what else to do. The user does not need to be bombarded by a thousand small nuggets of useless information, or be badgered by a raft of pointless questions. Save the interaction for when it really is vital. Don't report when you encounter a recoverable situation. Log the event by all means, but keep quiet about it. Provide a mechanism for the user to read the event log if you think they might care one day.

There *are* some problems that only the user can fix. For these it is good practice to report the problem immediately, to give the best chance to resolve the situation, or to decide how to continue.

Of course, this kind of reporting depends on whether the program is interactive or not. Deeply embedded systems are expected to cope on their own.

Recovery

Sometimes your only course of action is to stop immediately. But not every error spells doom. Some are quite expected. If your program saves a file, one day the disk will fill up and the save operation will fail. The user expects your program to continue faultlessly under these situations, so be prepared.

If your code encounters an error and doesn't know what to about it, pass the error upwards. It's more than likely your caller will have the ability to recover.

Ignore

I only include this for completeness. Hopefully by now you've learnt to scorn the very suggestion of ignoring an error. If you choose to forget all about handling it, and to continue with your fingers crossed: *good luck*. This is where most of the bugs in any software package will come from. Ignoring an error whose occurrence may cause the system to misbehave inevitably leads to hours spent debugging. Ignoring errors does not save time.

You'll end up spending far longer working out the cause of bad program behaviour than you ever would have spent writing the error handler.

You can, however, write code that allows you to *do nothing* when an error crops up. Is that a blatant contradiction of what you just read? No. It is possible to write code that copes with the world not being right, that can carry on correctly in the face of an error, but it often gets quite convoluted. If you adopt this approach, you must make it clear in the code. Don't risk it being misinterpreted as ignorant and incorrect.

Propagate

When a subordinate function call fails you probably can't carry on, but don't know what else to do. The only option is to clean up, and propagate the error report upwards. You have options. When propagating an error you can either

- export the same error information you were fed, or
- reinterpret the information, sending a more meaningful message to the next level up.

Ask yourself this question: does the error relate to a concept exposed (directly, or indirectly) through the module interface? If so, it's OK to propagate that same error. Otherwise, recast it in the appropriate light, choosing an error report that makes sense in the context of your module's interface.

This is a good self-documenting code technique. For example, you can catch and wrap up exceptions, or return a different reason code to the one you received.

Crafting error messages

Inevitably your code will encounter an error that the user has to sort out. Human intervention may be the only option; your code can't insert a floppy disk by itself or switch on the printer. (If it can, you'll make a fortune!)

If you're going to whinge at the user, there are a few general points to bear in mind.

- Users don't think like programmers, so present information the way they'd expect. When displaying the free space on a disk you might print this: Disk space: 10K. If there's no space left, a zero could be misread as 'OK' – the user will not be able to fathom why they can't save a file when the program says everything's fine.
- Make sure your messages aren't too cryptic. You might understand them. Can your granny? (It doesn't matter if your granny won't use this program, it will almost certainly be driven by someone with a lower intellect than her.)
- Don't present meaningless error codes (unless as some 'additional info' to send to the developers). No user knows what to do when faced with an Error code 707E.
- Make it clear what's an error and what's a mere warning. You could include this in the message text (perhaps with an Error: prefix), and can emphasise it in message boxes with an accompanying icon.
- Only ask a question (even a simple one like Continue: Yes/No?) if the user fully understands the ramifications of each choice. Explain it if necessary.

What you present to the user will be determined by interface constraints, and application or OS style guides. If your company has a user interface engineer, then it's their job to make these decisions. Work with them.

Code implications

Show me the code! Let's spend some time investigating the implications of error handling in our code. As we'll see, writing good error handling that doesn't twist and warp the underlying program logic is not a simple task.

Starting off in the world of C, the first piece of code we'll look at is a common error handling structure. Yet it's not a particularly intelligent approach for writing error-tolerant code. Our basic aim is to call three functions sequentially – each of which may fail – performing some intermediate calculations along the way. Spot the problems with this:

```
void nastyErrorHandling() {
  if (operationOne()) {
    ... do something ...
    if (operationTwo()) {
      ... do something else ...
      if (operationThree()) {
        ... do more ...
      }
    }
  }
}
```

Syntactically it's fine; the code will work. Practically, it's an unpleasant style to maintain. The more operations you need to perform, the more deeply nested the code gets, the harder it is to read. This kind of error handling quickly leads to a rat's nest of conditional statements. It doesn't reflect the actions of the code well; each intermediate calculation could be considered the same level of importance, yet they are nested at different levels.

Can we avoid these problems? Yes – there are a couple of alternatives. The first variant flattens the nesting. It's semantically equivalent, but introduces *some* new complexity, since flow control is now dependent on the value of a new 'status' variable, `ok`:

```
void flattenedErrorHandling() {
  bool ok = operationOne();

  if (ok) {
    ... do something ...
    ok = operationTwo();
  }

  if (ok) {
    ... do something else ...
    ok = operationThree();
  }

  if (ok) {
    ... do more ...
  }

  if (!ok) {
    ... clean up after errors ...
  }
}
```

We've also added an opportunity to clean up after any errors. Is that sufficient to mop up all failures? Probably not; the necessary cleanup may depend on how far we got through the function before lightning struck. There are two C-style cleanup approaches:

- Perform a little cleanup after each operation that may fail, then return early. This inevitably leads to duplication of cleanup code. The more work you've done, the more you have to clean up, so each exit point will need to do gradually more unpicking.

  If each operation in our example allocates some memory, each early exit point will have to release all allocations made to date. The further in, the more releases. That will lead to some quite dense and repetitive error handling code, and makes the function far larger and far harder to understand.
- Write the cleanup code once, at the end of the function, but write it in such a way as to only clean up what's dirty. This is neater, but if you inadvertantly insert an early return in the middle of the function, the cleanup code will be bypassed.

If you're not anal about writing *Single Entry, Single Exit* (SESE) functions, this next example removes the reliance on a separate control flow variable[4]. We do lose the clean up code again, though. Simplicity renders this a better description of the actual intent:

```
void shortCircuitErrorHandling() {

  if (!operationOne()) return;
  ... do something ...

  if (!operationTwo()) return;
  ... do something else ...

  if (!operationThree()) return;
  ... do more ...

}
```

A marriage of this 'short circuit' exit with the requirement for cleanup leads to the following approach, especially seen in low level C systems code. Some people advocate it as the *only* valid use for the maligned `goto`. I'm still not convinced

```
void gotoHell() {

  if (!operationOne()) goto error;
  ... do something ...

  if (!operationTwo()) goto error;
  ... do something else ...

  if (!operationThree()) goto error;
  ... do more ...

  return;

  error:
    ... clean up after errors ...

}
```

In C++ you can avoid such monstrous code using *RAII (Resource Acquisition Is Initialisation)* techniques, like smart pointers [Stroustrup97]. This has the added bonus of providing exception safety – when an exception terminates your function prematurely, resources are deallocated automatically. These techniques avoid a lot of the problems we've seen above, moving complexity to a separate flow of control.

The same example using exceptions would look like this, presuming that the subordinate functions do not return values, but throw an exception.

```
void exceptionalHandling() {
  try {

    operationOne();
    ... do something ...

    operationTwo();
    ... do something else ...

    operationThree();
    ... do more ...

  }
  catch (...) {
    ... clean up after errors ...
    ... and probably rethrow  ...
  }
}
```

This is only the most basic example. A sound code design wouldn't need the `try`/`catch` block at all. Writing good code in the face of exceptions requires an understanding of principles beyond the scope of this article.

4 Although this clearly isn't SESE, I contend that the previous example isn't, either. There is only one exit point, at the end, but the contrived control flow is simulating early exit – it's as good as multiple exit. This is a good example of how being bound by a rule like SESE can lead to bad code, unless you think carefully about what you're doing.

Raising hell

We've put up with other people's errors for long enough. It's time to turn the tables and play the bad guy. It's pitifully clear that when writing a function, erroneous things happen that you'll need to signal to your caller. Make sure you do – don't swallow any failure silently. Even if you're sure that caller won't know what to do in the face of the problem, they *must* be kept informed. Don't write code that lies, and pretends to be doing something it's not.

Which reporting mechanism should you use? It's largely an architectural choice; obey the project conventions, and the common language idioms. In C++ and Java it is common to favour exceptions, but only use them if the rest of the project does. A C++ architecture may choose to forego this facility to allow portability to platforms with no exception support.

One aspect of error raising is the propagation of errors from subordinate function calls. We've seen strategies for this already. Our main concern here is reporting fresh problems encountered during execution. How you determine these errors is your own business, but when reporting them consider:

- Have you cleaned up appropriately first? Reliable code doesn't leak resource, or leave the world in an inconsistent state, even when an error occurs – unless it's *really* unavoidable. If you do, it should be documented carefully. Consider what will happen after this error has manifested; when your code is next called, ensure it will still work.

- Make sure your error doesn't leak inappropriate information to the outside world. Only return useful information that the caller understands and can act on.

- Use exceptions correctly. Don't throw an exception for 'unusual' return values – the rare but not erroneous cases. Use exceptions only to signal exceptional execution circumstances. Don't use them for flow control; that is an abuse[5].

- Consider using assertions if you're trapping an error that should 'never' happen in the normal course of program execution, a genuine programming error.

- If you can push any tests to compile time, then do so. The sooner you detect and rectify an error, the less hassle it can cause. Compile-time assertions can be used in both C and C++.

- Make it hard for people to ignore your errors. Given half a chance someone *will* use your code badly. Exceptions are good for this – you have to be quite deliberate to hide an exception.

What errors should we be looking out for? This obviously depends on what the function's doing. Here's a checklist for the general kinds of error checking you should be doing in each function:

- Check all function parameters. Ensure you have been given correct and consistent input. Consider using assertions for this, depending on how strictly your contract was written (i.e. if it is an 'offence' to supply bad parameters),

- Check invariants are satisfied at interesting points in execution.

- Check all values from external sources for validity before you use them. File reading and the user's input values should be sensible, with no bits missing.

- Check the return status of all system calls and other subordinate function calls.

Managing errors

The common principle uniting the raising and handling of errors is to have a consistent strategy for dealing with failure, wherever it manifests. These are considerations for managing the occurrence, detection and handling of program errors:

- Try to avoid things that could cause an error. Can you do something guaranteed to work instead? For example, avoid allocation errors by reserving enough resource beforehand. With an assured pool of memory your routine cannot suffer memory restrictions. Naturally, this will only work when you know how much resource you need up front; many times you do.

- Define the program or routine's expected behaviour under abnormal circumstances. This determines how robust the code needs to be, and therefore how thorough your error handling should be. Can a function silently generate bad output, subscribing to the historic *GIGO*[6] principle?

- Define clearly which components are responsible for handling which errors. Make it explicit in the module's interface. Ensure a caller knows what will always work and what may one day fail.

- Check your programming practice: *when* do you write error handling code? Don't put it off for later, you'll forget to handle something. Don't wait until your development testing highlights problems before writing handlers – that's not an engineering approach.

  Write all error detection and handling *now*, as you write the code that may fail. Don't put it off until later. If you must be evil and defer handling, *always* write the detection scaffolding now.

- When trapping an error, have you found a symptom or a cause? Consider whether you've discovered the source of a problem which needs rectifying here, or have discovered a symptom of an earlier problem. If it's the latter then don't write reams of handling code here, put that in a more appropriate error handler.

Conclusion

> To err is human; to repent, divine; to persist, devilish.
>
> Benjamin Franklin

To err *is* human (but computers seem quite good at it, too). To handle this error is divine.

Every line of code we write should be balanced by appropriate and thorough error-checking and handling. A program without rigorous error-handling will not be stable. One day an obscure error may occur, and the program will fall over as a result.

Handling errors and failure cases is hard work. Its bogs programming down in the mundane details of the Real World. However, it's absolutely essential. As much as 90% of the code you write will be handling exceptional circumstances [ShawBentley82]. That's quite a surprising statistic, so write code expecting to put far more effort into the things that can go wrong than the things that will go right.

Pete Goodliffe

Homework

Here are a couple of questions to mull over, and discuss on `accu-general`.

1. How should you handle the occurrence of errors in your error-handing code?

2. Are *return values* and *exceptions* equivalent mechanisms? Prove it.

References

[ShawBentley82] Bentley, Jon Louis. *Writing Efficient Programs.* Prentice Hall Professional, 1982. ISBN: 013970244X

[Stroustrup97] Stroustrup, Bjarne. *The C++ Programming Language, Third Edition.* Addison Wesley, 1997. ISBN: 0-201-88954-4

5 I've seen people break a loop or end recursion by throwing exceptions. This uses an exception like a non-local `goto`. It's a curiosity, but a plain *wrong* use of exceptions.

6 That is, *Garbage In Garbage Out* – feed it rubbish, and it will happily spit out rubbish.

I_mean_something_to_somebody

Derek Jones <derek@knosof.co.uk>

Software developers are constantly exhorted to use *meaningful* identifier names. Unless they are chosen at random all identifiers are likely to be have some meaning, at least to the person who created them. The implicit assumptions in the exhortation is that the names are meaningful to subsequent readers of the source code and also that all readers of the source code containing them agree on what that meaning is.

Surprisingly there have been no studies investigating whether developers agree on the meaning assigned to uses of particular identifier names, although there have been studies that have investigated related issues. These studies include the meanings assigned to (human language) words, and words/phrases invented by people to describe something.

This article reports on an experimental study, performed during the 2003 ACCU conference, that attempted to measure one particular aspect of developer identifier meaning assignment behavior. The study investigated the extent to which belief in the applicable application domain affects the meaning assigned to identifier names.

The experiment is discussed in two articles. The first, this one, discusses the background to the experiment and some of the applicable characteristics of the subjects taking part; the second provides a review of other studies that have investigated meaning assignment and discusses the results of the ACCU experiment.

Why is an experiment necessary?

Developers are often unable to give a coherent answer to how they assign a meaning to an identifier. This kind of human behavior (knowing something without being able to state what it is) has been duplicated in many studies.

A study by Reber and Kassin [1] compared implicit and explicit pattern detection. Subjects were asked to memorise sets of words containing the letters *P*, *S*, *T*, *V*, or *X*. Most of these words had been generated using a finite state grammar. However, some of the sets contained words that had not been generated according to the rules of this grammar. One group of subjects thought they were taking part in a purely memory based experiment, the other group were also told to memorise the words but were also told of the existence of a pattern to the letter sequences and that it would help them in the task if they could deduce this pattern. The performance of the group that had not been told about the presence of a pattern almost exactly mirrored that of the group who had been told on all sets of words (pattern words only, pattern plus non-pattern words, non-pattern words only). Without being told to do so, subjects had used patterns in the words to help perform the memorisation task.

How do developers assign a meaning to an identifier?

This issue is discussed in some detail in part 2 of this article. For the time being we assume that the meaning assigned to an identifier by a developer is created using a repertoire of previously learned techniques operating on a substantial body of knowledge they have in their head. The significant techniques and knowledge are assumed to be:

- experience of human and computer languages,
- knowledge of particular domains (e.g., software engineering concepts, or knowledge of an application domain such as accounting),
- experience in reading and writing source code,
- information obtained from the context in which the identifier occurs. This issue is discussed further in part 2.

Prior experience

Studies have found that nearly every task that exhibits a practice affect follows the *power law of learning*. This law has the form:

$$RT = a + bN^{-c}$$

where RT is the response time, N is the number of times the task has been performed (not the amount of time spent performing the task), and a, b, and c are constants.

The value of c is usually greater than 1, which means that ever larger amounts of practice are needed to obtain the same increase in performance.

Studies have found that practice effects exist in peoples use of language. For instance, a large number of studies have verified that a *word frequency effect* exists. The number of times a person has been exposed to a (natural language) word has a significant effect on their recognition of and performance in handling that word.

Education within specific knowledge domains has also been found to affect word handling performance. For instance, a study by Gardner, Rothkopf, Lapan, and Lafferty [2] used 10 engineering, 10 nursing, and 10 law students as subjects. These subjects were asked to indicate whether a letter sequence was a word or a nonword. The words were drawn from a sample of high frequency words (more than 100 per million), medium frequency (10-99 per million), low frequency (less than 10 per million), and occupationally related engineering or medical words. The nonwords were created by rearranging letters of existing words, while maintaining English rules of pronounceability and orthography.

The results showed engineering subjects could more quickly and accurately identify the words related to engineering (but not medicine). The nursing subjects could more quickly and accurately identify the words related to medicine (but not engineering). The law students showed no response differences for either group of occupationally related words. There were no response differences on identifying nonwords. The performance of the engineering and nursing students on their respective occupational words was almost as good as their performance on the medium frequency words.

Domain knowledge

Given the likelihood that subjects taking part in the experiment would have a wide variety of software related backgrounds it was decided to attempt to restrict the domains they considered when answering experimental questions. Two domains that subjects were likely to be broadly familiar with were chosen. These two domains were operating systems (the Linux Kernel) and computer games (Doom, a product of ID Software).

It was hoped that specifying a particular domain would provide a global context within which it would be possible for subjects to provide a few answers, that they considered to be the likely meanings. For instance, the word "bank" has a number of possible meanings. Being told that it occurs in a financial context is likely to cause people to associate a meaning with it that is different than if they had been told it occurred in a discussion about rivers. It is also possible for a localised context to override a global context. For instance, discussing the watery view from an accountancy company's offices might suggest a river bank, while a discussion of the cost of restocking a river with fish might trigger finance related thoughts.

A number of subjects indicated (both after taking part in the experiment and through writing on their response sheet) that they were unable to provide a meaning to some identifiers because insufficient context was available to them.

Source code experience

Given that a subject's performance is driven by the amount of time they have spent performing the task, we need some way of measuring the amount of time spent working directly with source code.

Traditionally, developer experience is measured in number of years of employment (performing some software related activity). It is relatively easy for a person to calculate the amount of time they have been employed in a software development related role. However, the extent to which the amount of time spent in software related employment correlates with experience working on source code is not known (there are many software related employment activities that do not involve a person working at the source code level).

The quantity of source code (measured in lines, not time spent) read and written by a developer (developer interaction with source code overwhelmingly occurs in its written, rather than spoken, form) is a more direct measure of experience. Interaction with source code is rarely a social activity (one situation where it does occur socially is during code reviews) and the time spent on these activities may be small enough to ignore. The problem with this measure is that it is very difficult to obtain reliable estimates of the amount of source read and written by a developer. The problems include:

- readers don't always read code on a line by line basis. For instance, they may scan the source looking for some construct, or may only read part of a line,

- the same code is often read several times. Does each instance of reading have the same learning affect?
- code may be written and then rewritten or deleted (existing productivity measures are based on final number of lines of debugged code),
- few developers regularly measure the amount of code they write. This means they are very unlikely to be able to make an informed estimate of the total amount of code they have read or written.

Even although there appear to be significant problems in obtaining reliable answers from developers on the amount of source they have read and written your author believed something could be learned from the subjects' responses.

Experimental setup

The experiment was run by your author during two 30 minute sessions (on different days) of the 2003 ACCU conference held in Oxford, UK. Approximately 250 people attended the conference, 45 (18%) of whom took part in the experiment. Subjects were given a brief introduction to the experiment, during which they filled out background information about themselves, and they then spent 15 minutes working on the identifier list. All subjects (31 on the first day, 15 on the second) volunteered their time and were anonymous.

The requested subject background information was as follows:

- What is is your native language?
- Please list any human languages that you speak fluently:
- Please list any of these languages that you use at least once a week:
- Please list the computer languages that you have spent a significant amount of time reading and writing (at least 100 hours, i.e., 3 work weeks) over the last two years:
- How many lines of code would you estimate you have *written*, in total, over your career:
  - 10,000
  - 25,000
  - 50,000
  - 75,000
  - 100,000
  - 150,000+
- How many lines of code would you estimate you have *read*, in total, over your career:
  - 10,000
  - 25,000
  - 50,000
  - 75,000
  - 100,000
  - 150,000+
- How many years have you been writing software professionally?

Subjects' background

On the first day 30 subjects handed in their completed response sheets (one subject was not happy with their performance and it was agreed that they could keep the sheet containing their responses), on the second day 15 response sheets were handed in.

Most of the subjects were native speakers of English:

Native Language	Number of subjects
English	35
French	2
German	2
Italian	1
NL	1
Russian	2
Slovenian	1
Swiss German	1

Table 1: Number of subjects having the given language as their native language.

The response "NL" is assumed to refer to the country code for the Netherlands.

The subjects regularly used a wide range of computer languages, see table 2. The most commonly used language was C++, followed someway down the list by C, and Java (the ACCU is the Association of C and C++ Users, and a Python conference was also taking place in the same place at the same time).

Computer Language	Number of subjects	Computer Language	Number of subjects
Assembler	1	ML	1
Basic	1	OLC	1
C	20	Pascal	2
C#	2	Perl	9
C++	41	PHP	3
Cobol	1	Python	6
Fortran	3	Rebol	1
Haskell	1	shell	6
HTML	3	SQL	3
IDL	1	TCL	1
Java	12	VB	7
Javascript	4	VBA	1
Lisp	1	VBscript	2
make	1	XML	3

Table 2: Number of subjects regularly using a particular computer language.

"shell" was the generic term used for a variety of command line shells.

Over 50% of the subjects had more than 11 years of experience in software development (see figure 1). Whether this figure is representative of the general population of developers (or even of those attending the conference; there were other events taking place throughout the extended lunch break during which the experiments were run) is not known.

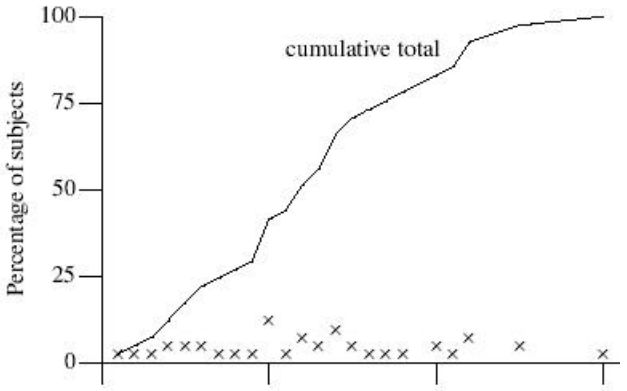


Figure 1: Number of years experience.

Cumulative percentage (line) of subjects who have been writing software professionally for the given number of years. The crosses represent the relative number of subjects having the given number of years employment. Four subjects did not provide an answer to this question.

As figure 2 (see next page) shows, your author clearly underestimated the number of lines of code that subjects believe they have read and written (he also has to admit to thinking that the average number of years of professional software development would be lower).

Conclusions

Based on years of employment the majority of the subjects have a significant amount of software development experience. The measurements based on lines of code read are likely to suffer from incorrect self-calibration on the part of subjects and a ceiling effect caused by overly restrictive response options.

Further conclusions will be given in part 2 of this article.

Derek Jones

Maintaining Context for Exceptions (Alternative)

Andy Nibbs <andrewn@chersoft.co.uk>

At ACCU 2002 Andrei Alexandrescu talked about storing contextual information in the event of an exception being thrown. The individual elements of context mirror the unwinding of the stack and the progress from lower level code to wherever the exception finally stops and is communicated to the user.

It is at the lower level that things (normally) go wrong because that's where the work is done. Unfortunately at that level we are in often the worst position to compose a meaningful error message. The low level code lacks knowledge of the overall intent of the code – this is because we needed to break things down and wanted reuse!

We are interested in what went wrong and what we were trying to do at the time. At the point of the error we know what went wrong, and as we unwind the call stack we move through the levels of detail about what was being attempted.

If, when an error occurs we use an exception to report the specific problem and as that exception unwinds the stack we record the context, we can get a full report on what went wrong. This is helpful to the user and anyone supporting the product (who are helpful to the user).

Rob Hughes' article in the August CVu explained a system that he had developed and used in his projects. At Chersoft we were also inspired by Andrei to create a parallel system which we have successfully used in commercial 'shrink wrap' software.

In this article I'll outline the similarities and differences between our approaches, some extra bits we do and the evolution of our system – some of which has come from looking at Rob's August piece. It might be worth getting hold of it and reading it before tackling this one.

Similarities and Differences

Both systems use a class for each piece of context and macros to put instances of it onto the stack (singleton pattern). We have not needed to do a 'thread specific singleton' though we considered it.

We have both 'user' and 'technical' messages – we can put all sorts of useful stuff in the technical messages that might scare the users but would be handy for programmers supporting a product.

std::uncaught_exception()

Rob reports that `std::uncaught_exception()` always returns `false` with Borland C++ builder. Same for VC++ 6.0 – in the Dinkum STL you can read code a bit like:

```
bool std::uncaught_exception() {
  return false;
}
```

It doesn't even try! We tried writing our own but it was desperate and we retreated.

A properly service packed VC++.NET works correctly. We develop our code to compile and run using both VC++ 6.0 and VC++.NET, so we have a workaround – our own exception base class. `CExceptionBase`, as it's known, increments a `static int` on construction and decrements it on destruction.

```
static bool CExceptionBase::Uncaught() {
  #if _MSC_VER >= 1300
  return std::uncaught_exception();
  #else
  return s_nInstances > 0;
  #endif
}
```

The context class uses the above static method to determine whether an exception is active. As long as classes derived from `CExceptionBase` are just thrown as exceptions it is fine.

We derive various exceptions from this base and have a dialog which reports on the exception and the context stack.

So we are ok if we throw our own exceptions.

But we are at risk from STL exceptions and platform specifically:

- Various MFC exceptions
- COM exceptions
- Win32 structured exception handling exceptions!

So in various ways we convert these exceptions into our own.

Exception Translation

```
catch(CFileException * pEx) {
  throw CExceptionMFC_File(pEx);
}
catch(CArchiveException * pEx) {
  throw CExceptionMFC_Archive(pEx);
}
catch(CException * pEx) {
  throw CExceptionMFC(pEx);
}
catch(std::exception& ex) {
  throw CExceptionSTL(ex);
}
catch(const _com_error& ex) {
  throw CExceptionCOM(ex);
}
catch(csx::CExceptionBase&) {
  throw;
}
```

This deals with the nasty way MFC `new`s exceptions and has them deleted and so on. The information contained in the exceptions is sucked out, put in one of ours and thrown.

We put this in a macro and called it `CATCH_RETHROW`. The translation is handy, we can still use our dialog:

[concluded on next page]

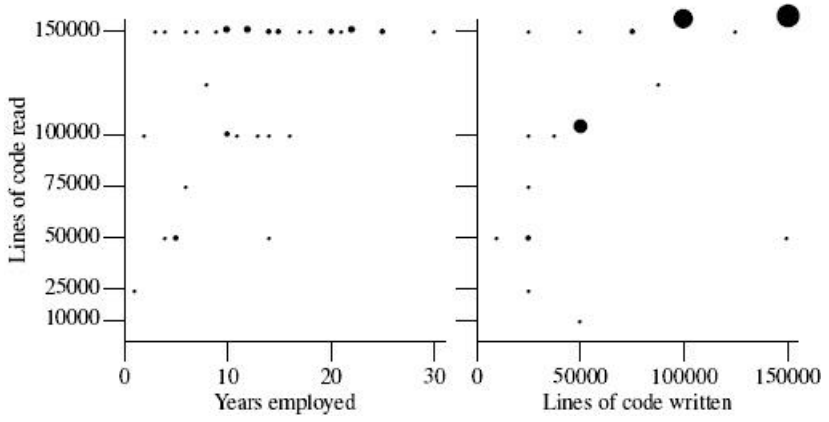


Figure 2: Lines of code read and written.

The graph on the left depicts number of line of code read against number of years of writing software professionally. The graph on the right depicts the number of lines of code read against the number of lines of code written for each subject. The size of the circle indicates the number of subjects specifying the given values. In those cases where subjects listed a range of values (i.e., 50,000-75,000) the median of that range was used.

References and further reading

[1] Reber and Kassin
[2] Gardner et al

The Psychology of Language by Trevor Harley is an undergraduate level introduction to the subject. Those wanting a lighter read might like to try *Word in the Mind* by Jean Aitchison,

Learning and Memory by John Anderson is an undergraduate level introduction to the subject. Those wanting a lighter read might like to try *Essentials of Human Memory* by Alan Baddeley.

Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiment, and ACCU for making conference slots available to run it.

```
try {
  try {
    CONTEXT("Doing the work!");
    // do the work involving lots of
    // method calls etc..
  }
  CATCH_RETHROW;
}
catch(const CExceptionBase& ex) {
  CexceptionDlg dlg(ex, CexceptionContextStack
           ::Instance().GetAndResetContext());
  dlg.DoModal();
}
```

Wonderful! But on a compiler without a proper `std::uncaught_exception` we miss out on an awful lot of contexts. We only get context information after `CATCH_RETHROW` has been done. We need more of them!

`try..catch` blocks are costly, the compiler needs to put up all sorts of scaffolding to support their operation. We don't want every function to have one.

As an aside... at a previous company I worked an a system where (nearly) every function was something like...

```
void foo() {
  try {
    // work
  }
  catch(Cexception& ex) {
    ex.AddMessage("Bit of context");
    throw ex;
  }
}
```

It was more complicated and macros were used. The code was slightly obscured, larger and slower. On the other hand it could produce good errors and there was a consistent application wide error handling scheme.

Getting back to the present...

We tend to put the extra `CATCH_RETHROW` blocks around where we think an exception might emerge – looking at the list of exceptions that's not too hard to do. When we switch to the VC++ .NET compiler completely the problem goes away (and it is a better compiler).

Key difference: push..pop or just push

Rob's solution has a context class that pushes information onto a stack on construction and removes it if an exception is not active. Our context class pushes on destruction. If an exception were thrown during our pushing there would be two exceptions on the go at once and the application would terminate because that's not allowed.

Rob's code is exception safe as the STL `deque` guarantees that `pop_back()` will not throw.

We take a risk that on Windows memory allocation doesn't tend to fail – it just takes a very long time when memory is low. By this stage everything has gone horribly wrong anyway.

In return for taking the risk we gain quite a lot of efficiency because the context class does very little, excepting the exceptional case.

Win32 SEH Translation

We convert Win32 structured exceptions like 'Access Violation' into C++ exceptions derived from our exception base class. This is done by hooking in using the function `_set_se_translator`.

It's OS specific so I'll not go into detail. Googling on the function name above will help anybody interested. It means we get a bit more information if a 'crash' occurs.

How our system has evolved: Clearing the stack

If the stack has context information left over from a previous exception on it when an exception is thrown the user could be exposed to two sets of context, which is confusing.

Our first stab at dealing with this potential hazard was for the method which returned the context information to return a copy of the stack and then clear the stack.

```
const CContextInformation
CExceptionContextStack::GetAndResetContext();
```

But, before long another method had turned up:

```
const CContextInformation& Context() const
```

It was handy to peek at the stack in order to write it out to a debug window.

The stack could be cleared automatically in the context class constructor – if there is not an exception on the go the stack can be cleared.

```
CContext::CContext(..) :
m_bExAtStart(std::uncaught_exception()) {
  if(m_bExAtStart)
    ClearStack();
}
```

It adds to the overhead of using context of course. It depends on the application whether or not this is a concern.

Tightening up the context class

Originally we did not have a flag in the context class that recorded whether or not there was an active exception on construction.

```
CContextString::~CContextString() {
  if(!m_bExceptionAtStart &&
                  ExceptionBase::Uncaught()) {
    PushContext();
  }
}
```

This prevents the case where a context macro is encountered in code during the unwinding process (perhaps in a library call). This potentially results in a confusing extra message on the context stack.

The Hard Bit

Now that the technical bits are out of the way the system must be used, you still need to write appropriate, useful messages. What is the audience for them? The more wide ranging the worse it is.

You need enough information for the more able users to sort things out themselves, but error messages can scare less expert users who fear for their lives and freedom at talk of 'fatal' errors and 'illegal' operations.

We've found that it is worth provoking a few errors to see what messages emerge, by changing the code, deleting files or whatever it takes. We modify the results by changing text, adding more context macros and so on.

Redundant or inappropriate information is almost as bad as too little. An agreed policy helps.

One case we've seen is 'file not found', in most situations you need to know what the filename is and where it is trying to find it. Do you want to include the filename in the context message or the exception?

```
CONTEXT("Trying to open " + filename);
if(!FileExists(filename)) {
  throw CException("Could not find the file "
                   + filename);
}
```

Here we get the filename twice which is cluttered. Throw in some more context and other redundant sloppiness and you have something that either needs careful reading or stays unread by the user and turns into a support call – this in turn makes the user feel less empowered and costs money.

So the hard bit is writing good consistent messages for the user. My advice:

- Provoke errors (modify if they are not good enough)
- Have a consistent policy.

When and where

We don't use contexts all the time. Our software is used by the marine community for various applications around navigation, planning, tide prediction and so on. The applications involve the importing and updating of data provided by third parties. Navigational charts are the obvious example.

It is these complex batch-like bits of the software, say importing 4000 charts, that most lend themselves to the exceptions and context error handling strategy. Something goes wrong deep down and the exception unwinds to a single point where it can be reported to the user or logged.

The overhead imposed by contexts does not worry us here.

In the day to day GUI use of our applications, when drawing a route on a chart we have far more control. Exceptional situations should not happen and performance is an important consideration. (Drawing performance is something that can separate our applications from that of competitors). We don't use contexts in this situation.

Andy Nibbs

BRACKETS OFF!

Thomas Guest <thomas.guest@ntlworld.com>

The mathematical formula:

```
v = u + at
```

calculates the speed, `v`, of an object, with initial speed `u` and constant acceleration `a`, after time `t`. Placing the "`a`" next to the "`t`" is a convenient shorthand for "multiply `a` by `t`", which also makes it apparent that the multiplication must be done before the addition.

When the same formula is written in C, the multiplication operator needs explicit representation:

```
Example 0)   v = u + a * t
```

The layout of this expression no longer makes it clear that the multiplication should be done before the addition, so a programmer might choose to parenthesise:

```
v = u + (a * t)
```

Are these parentheses required to guarantee correct evaluation of `v`? If not, should they be included anyway, to help convey the meaning of the expression? How can coding standards help with such choices?

This article aims to answer these questions. It first presents some examples of the operator precedence and associativity rules in action, then offers some guidelines on when to parenthesise expressions, and finally argues that these guidelines should be replaced by a single rule.

More Examples

```
Example 1) x = 8 - 4 - 2
Example 2) r = h << 4 + 1
Example 3) str += ((errors == 0)
                   ? "succeeded" : "failed")
Example 4) *utf++ = 0x80 | ucs >> 6 & 0x6f
```

The expression presented in Example 0 contains three operators: assignment, addition and multiplication. These operators – indeed all operators – follow a strict precedence which defines the order of evaluation. Since multiplication has higher precedence than addition, which in turn has higher precedence than assigment, the expression is equivalent to:

```
v = (u + (a * t))
```

This means the compiler can be trusted with the expression as first presented. No parentheses are required. Good, the language does what we expect.

In Example 1, subtraction binds more tightly (i.e. has higher precedence than) assignment, so the subtractions are performed first. Since all the arithmetic operators associate left to right, the expression is equivalent to:

```
x = ((8 - 4) - 2)
```

In Example 2, arithmetic operators bind more tightly than shift operators, so the expression is equivalent to:

```
r = (h << (4 + 1))
```

Why did the programmer not write `r = h << 5` ? Probably because he really meant:

```
r = (h << 4) + 1
```

but bit shifting (like, say, finding the address of something, or subscripting an array) somehow seems closer to the machine and feels as if it ought to be of higher precedence than addition, so the crucial parentheses were missed[1].

In example 3, the parentheses are unneccessary, since the comparison operators bind more tightly than the conditional operator, which in turn binds more tightly than the assignment operators. Do the parentheses help you understand the meaning of this expression? Would you have left them out – and if so, would one of your team-mates have complained?

How should the fourth example be parenthesised, to make its meaning clear? It is equivalent to:

```
*(utf++) = (0x80 | ((ucs >> 6) & 0x6f))
```

which shows how complicated an expression looks when parentheses are added indiscriminately.

Coding Standards and Guidelines

In general – at least, in my experience – coding standards do not provide rules on how to parenthesise expressions. I suspect this is for two reasons. Firstly, because although all programmers use parentheses to clarify the meaning of expressions, they may well disagree on what makes an expression clear. Clarity seems a matter of taste. While programmers in a team may agree (to differ) on whether tabs or spaces are to be used for indentation, their coding standard leaves them free to rewrite Example 4 as:

```
str += errors == 0 ? "succeeded" : "failed"
```

And secondly, if a coding standard were to rule on how to parenthesise, it would be difficult to find a middle ground. This leaves as candidate rules the two extremes:

- parenthesise everything
- never parenthesise

The first quickly leads to unreadable code, and the second seems overly proscriptive. In the absence of a hard rule, here are some guidelines, which I hope are non-contentious, and which may help us reach a conclusion:

- have the operator precedence tables to hand and understand how to interpret expressions using them,
- understand the logic behind the operator precedence tables, but be aware of the traps and pitfalls,
- remember, parentheses are not the only way to make order of evaluation clear. For example, Ex 4 could be rewritten:

```
*utf++ = 0x80 |
             ucs >> 6 &
             0x6f
```

  or even:

```
*utf = ucs >> 6;
*utf &= 0x6f;
*utf |= 0x80;
++utf;
```

- if an expression is hard to understand, break it down into simpler steps, or extract it out as a function with a meaningful name,
- trust the compiler: it might not implement partial template specialisation correctly, but it will get operator precedence right every time,
- never use parentheses simply because you aren't sure how an expression will be evaluated without them: treat doubt as an opportunity to learn,
- all macro arguments must be parenthesised.

Concluding Thoughts

Any effort put into becoming familiar with precedence tables is likely to pay off across a range of languages. For example, although C++ introduces several new operators over C, there are no surprises. The precedence rules remain in force even if the operators have been overloaded (but that's the subject of another article). Java operator precedence is almost a subset of C's. Similarly, scripting languages are generally compatible with C, even where C's precedence rules are slightly screwy[2]. So, while PERL introduces lower precedence versions of the logical operators `not`, `and`, and `or`, it ensures that `not` binds more tightly than `and` which in turn binds more tightly than `or`[3]. Interestingly, in Python, where whitespace is syntactically significant, parentheses can be used not just to indicate order of evaluation, but also to wrap lengthy expressions over several lines.

[concluded at foot of next page]

1 This example is lifted straight from [1], from the section headed: "Operators do not always have the precedence you want".

2 According to [1], some of C's peculiarities can be blamed on its heritage: "The precedence of the C logical operators comes about for historical reasons. B, the predecessor of C, had logical operators that corresponded rougly to C's & and | operators. Although they were defined to act on bits, the compiler would treat them as && and || if they were in a conditional context. When the two usages were split apart in C, it was deemed too dangerous to change the precedence much."

3 This contrasts C/C++, where `not`, `and` and `or`, if available, are equivalent to `!`, `&&` and `||` respectively.

Reviews

Bookcase

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

Francis Glassborow writes:

If you look back over the last decade you will realise that I have been responsible for a very high proportion of reviews of books aimed at novices. With the publication of my own book aimed at this part of the market I feel that I should not continue to review books that are in direct competition (possibly I could ethically justify reviewing very good ones.) Therefore ACCU needs one or more volunteers to deal with the continuing flow, usually very poor and frequently technically inaccurate, books for newcomers either to programming or to C, C++ etc. Perhaps such volunteers would like to sharpen their teeth by doing an honest review (no holds barred) of my book.

Prices

My recent visit to the US made me aware of just how weak the US$ has become over the last year. Several publishers have responded by shifting their conversion rates for the list prices of their books. At least one has not (or does not appear to have done.) O'Reilly currently operates at around $/£ of 1.40 which is acceptable when I allow for the extra transport costs incurred in moving books from the US to Europe. For publishers going the other way (where it is the US distribution that nominally has extra transport costs) about 1.5 would seem more reasonable.

I hope Addison-Wesley will understand that I am always as even-handed as I know how to be. They have frequently had praise from me for the general quality of their books in the C++ domain as well as for books of more general interest to programmers. Being the best at one thing does not excuse being among the worst at something else. Their recent pricing has tended towards being exorbitant and this is further exacerbated by being among the worst for inflating their UK prices. A book that they list at £30 should be far closer to £26. If there is the slightest doubt that they are over-pricing look at the magnitude of discount that `amazon.co.uk` offers on many of their books (50% is not uncommon). Note how much smaller the discounts are when there is no listed UK price (i.e. they are discounted from the US price converted to Sterling)

And while I am on the subject of discounts, US readers might like to check `www.bookpool.com`.

All the best for 2004.

Francis

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

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

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

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

Francis

C & C++

The C Standard (0 470 84674 7), Wiley, 792pp @ £34-95/$65

The C++ Standard (0 470 84573 2) Wiley, 782pp @ £34-95/$65
reported by Francis Glassborow

In case you have not noticed the C++ Standard updated with Technical Corrigendum 1 is now available alongside the earlier published C Standard (also updated with its TC1). The latter book includes the Rationale (for changes and additions to the original Standard) written by WG14 – the ISO Committee responsible for producing the C Standard.

Both books are officially the corresponding BSI Standards but these are word for word identical to the ISO Standards and the equivalents for other National Bodies.

The two volumes are produced as a matching set. The binding has been chosen so that they can be used effectively as reference books (i.e. they lie flat on the table without pages spontaneously turning over).

You may like to note that when members of WG14 & WG21 were offered a chance to purchase their own copies (at a special discount price) a very high proportion of them did so (quite a few buying multiple copies for use by their colleagues). They recognise that even in an era of electronic copies (and being involved in Standardisation work they are entitled to free electronic versions of relevant standards) a well presented printed copy has its place.

Now that relatively inexpensive electronic and paper copies of both the C and C++ Standards are available, I would consider that a professional programmer has *no* excuse for being unfamiliar with the requirements of those Standards. If the work you do is in any way critical then you should have access to the appropriate Standard either through your employer or by personal purchase.

A Correction

Please note the following correction to my review of *C++ in a Nutshell*:

I just read your review of Ray Lischner's recent book, C++ In A Nutshell. In that review, you make some comments about page number references:

> Now such back references are fine in books designed for reading but they are not adequate in a reference book where a page number should always be given.
>
> ...
>
> Perhaps they can persuade the author to work harder at cross referencing for the next edition.

It's not really fair to blame Ray for the lack of page-number references. He'd probably prefer

[continued from previous page]

The more experienced I become as programmer, the fewer parentheses I use. Coming from a mathematical background, it was several months into my first job before I dared use the conditional operator – and when I finally did start using it, I parenthesised all the sub-expressions for safety. Later on in my career, when I first found myself working with the bitwise operators, again, I enclosed sub-expressions with brackets. As my confidence has increased, the brackets have peeled away.

This, though, is simply evolution. Familiarity with the languages you use makes it easier to read expressions without the unnecessary noise of parentheses. Evolving in this way, however, leaves a programmer vulnerable when working on code written by a more experienced team-mate, unless the experienced programmer writes to a lowest common denominator.

Surely it would be better for everyone to program to a highest common denominator. The operator precedence tables are a fundamental part of the language. The rules for using them are simple. Although there are many precedence levels, the operators do group logically. Update your Coding Standards. Prohibit unnecessary parentheses. Brackets off!

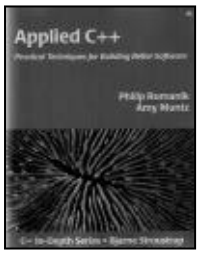
Thomas Guest

References

[1] Andrew Koenig, 1989, *C Traps and Pitfalls*, Addison-Wesley, ISBN 0-201-17928-8 2

to put them in, but they aren't an option that our production editors support. I won't argue the validity of your complaint, but in this case it's the **publisher's** fault (i.e. it's O'Reilly's fault) that the book doesn't include page-number references.

Best regards,
Jonathan Gennick
(Editor, O'Reilly & Associates)
`jgennick@oreilly.com`



Applied C++: Practical Techniques for Building Better Software by Philip Romanik and Amy Muntz (0-321-10894-9), Addison-Wesley*, 330pp @ £30-99 (1.29)
reviewed by Giles Moran

This book is the latest addition to the popular and well-received "C++ In-Depth" series. The introduction states that this is a practical guide to maintaining and developing high quality software. Looking through the book, initial impressions are that the production values of the book are very good. A CD is supplied and has source code listings and some third party software used in the book. Class diagrams use the format used in '*Large Scale C++ Software Design*' by Lakos rather than UML. This is odd as the notation in that book was developed in 1996 and the world has moved on since then.

The first chapter details some thoughts on the software design process and an introduction to some basic image processing ideas.

Chapter two moves onto the development of a simple application that turns large images into a thumbprint image.

Chapter three details the development of a custom memory handling class for the imaging system. Allocators and memory alignment are discussed.

A review of templates is included (templates in six pages!) The authors seem unclear on the use of `typename` vs `class` in template specification. If you didn't already understand the differences, it's unlikely this will help.

A prototype is developed and a discussion on why prototypes are useful follows. It would have been more readable if these sections were in reverse order. Two further prototypes are developed.

Chapter four covers naming conventions, code reuse and a debugging output class

Threads, exceptions, assertions and internationalisation are discussed in chapter five. This is probably the most useful chapter in the book.

Chapter six is concerned with implementation details for the imaging software example. Some imaging techniques and filters are then covered in detail.

Unit testing and performance issues are covered in chapter seven before advanced topics (use of `explicit` and `const`) are discussed. I'm not sure why the authors consider use of `explicit` and `const` as advanced topics.

The book suffers from having too much code on display. Entire header and implementation files are listed and only a few functions are discussed. The source code is provided so it's not necessary. The class naming convention is unclear, an `ap` prefix is used so an `Image` object is referred to as an `apImage` object which reduces readability.

Templates are heavily used and it's not uncommon to have five templated parameters in example code. Template parameter names are short and make the code unreadable. Common sense and `typedef`s would have helped.

Modern C++ is mainly absent. Namespaces warrant seven lines.

The STL is briefly discussed, `string`, `map`, `vector` and `list` are mentioned but little discussion is given. Algorithms are not discussed. The author's advice is to learn a small subset of the STL and just use that. This is dubious advice.

Design patterns discussion centre around the use of the singleton pattern. They are heavily used. No discussion is given to the disadvantages and no alternatives given.

There is just too much in this book; it covers templates, allocators, C++ idioms (reference counting and handle class), the singleton pattern, exceptions, internationalisation, threads, profiling, performance improvements, XML parsing, image formats, image processing and more in only 300 pages. What is discussed is tightly focused on the given application, such that it's hard to see how it's of any use to you. Comments such as 'Use iterators to access all of the pixels in the image' are given, this comment is only in context with respect to the class given in chapter six of the book. I got the distinct feeling that the application code led to the book's creation and that I was reading a software manual at times. The sections on imaging processing are ok, but there's not enough detail to learn much about the subject.

The introduction mentions development and maintenance of software, if you remove the section on debugging, then there is little on software maintenance or writing maintainable code.

This is a disappointing book, given the quality of other books in this series. It didn't help that the last software book I read was '*Agile Software Development: Principles, Patterns and Practices*', which is a much better book for intermediate programmers to invest in.

Second review by Paul F Johnson

This is what a good book should be like! It has a definite purpose, well written code and concise explanations.

The pretext of the book is that the author has already written a simple image viewer and now the authors are going to re-implement the software, but this time with a logical and systematic approach.

What is even better is that the code is platform independent (which is exactly as it should be) and extremely good code at that (all the code you need is supplied on the CD supplied with the book). It is well written, the process of how the original code could be re-written and improved upon is clear.

If you are a competent C++ programmer, you should seriously consider getting hold of this book, especially if your ultimate goal is clear code that can (with the minimal amount of effort) be recompiled on other platforms with the same results. I would not recommend it though if you are just setting out – it is seriously for advanced users. Highly recommended.

Teach Yourself C++.NET in 21 Days by Davis Chapman (0 672 32197 1), SAMS, 777pp @ £28-99 (1.38)
reviewed by Paul F. Johnson

As C++ books go, this book contains surprisingly little, in fact, for the C++ part, you can probably do over about 3 hours.

This book has one of the most misleading titles I've come across as the book is really a manual for the Visual xxx.NET IDE and MFC. With this in mind, I've decided to review the book as a manual.

As manuals go, this is rather good. Not brilliant (it tries to cram too much in at the expense of accuracy and detail), but rather good. The parts of the MFC touched upon are well written and well documented, but there are not enough of them or of sufficient complexity to be of much use.

For once though, the time frame for teaching yourself the ins and outs of an IDE are not too far off the mark – this is a significant improvement over the other SAMS teach yourself books.

If your copy of the MS Visual C++.NET manual has gone missing, this will fill the place it occupied and possibly do a better job. For the rest of us, it's of little use.

The C++ Standard Template Library by Plauger, Stepanov, Lee & Musser (0 13 437633 1), Prentice Hall, 485pp @ £34-99 (1.51)
Second review by Francis Glassborow

I was not the original reviewer of this book for ACCU but remember being a little surprised by the review that was provided. I hardly ever try to second guess ACCU reviewers as they take their task seriously and always write as they see. A review is always subjective to some extent.

I have regularly reminded readers that if they disagree with a review they have an obligation to say so and say why. The mystery is why it has taken me so long to realise that this obligation applies just as strongly to me as it does to others. It is long past time I put right the classification of this book that is the result of that first review.

I think that the single biggest problem with this book is that the title leads many to expect a very different book. It does not help to be familiar with the other work of the lead

author because he writes across a very wide range (I wish '*Programming on Purpose 2*' and '*Programming on Purpose 3*' were currently in print along with the reissued '*Programming on Purpose*'). Without the other authors you would expect this to be a companion volume to '*The Draft C++ Standard Library*' (another book desperately in need of a second edition) and '*The C Standard Library*' (also needs a new edition to encompass changes brought about by C99). Unfortunately adding in three experts who were very much responsible for the design and early development of the STL leads many to expect a tutorial or manual on the STL. The latter expectation is entirely false and the former completely correct.

PJP is one of the world's great implementers of libraries. He is more familiar with the fundamentals of library implementation than anyone else I know. He has also been very willing over the years to share this knowledge with others. You should not expect details of the final tweaks that make his implementations so highly regarded but within the covers of this book you will find a great deal of useful material at two levels.

The first level allows the serious working programmer to see inside the STL implementations and understand some of the forces that motivated the specific methods that were selected for the version of the Dinkumware C++ Library circa 2000 (experience has led to some changes in implementing some things). This can be important when you are having doubts about whether to handcraft some code or use components shipped with your compiler.

The second level is for those who aspire to becoming library implementers themselves. A close study of this book will greatly improve your understanding of library implementation, particularly as regards to templates.

If I want to use the STL I would first reach for my copy of Nico Josuttis' '*The C++ Standard Library*' but if I want to understand a good implementation, perhaps because I think I need to add my own hand-rolled replacement to the library I am using, then this is the book I reach for.

This is definitely not a book for everyone but on the other hand it is a valuable book for the serious C++ programmer who wants or needs insight into template implementation. It is also essential reading for the aspiring library implementer.

STL Pocket Reference by Ray Lischner (0 596 00556 3), O'Reilly, 120pp @ £6-95 (1.43)
reviewed by Francis Glassborow

Yet another of these pocket size books from O'Reilly. The subtitle is *'Containers, Iterators, and Algorithms'*. There is no doubt in my mind that the STL is a suitable topic for one of these books. Few programmers are fluent enough to remember all the parameters of the algorithms in the correct sequence. And then there is the matter of which containers have some algorithms provided as member functions.

In general this little reference book does a good job though I find it a little quirky in places. For example it completely ignores the 'evil' `vector<bool>`. I think that even a small reference such as this one should not be ignoring specialisations that are so far removed from the template they specialise.

On the other hand the book finishes with miniscule (about six pages) coverage of the Boost libraries. That is pushing in the right direction but I would have hoped for a good deal more. I hope O'Reilly will consider a second edition with more comprehensive cover of 'C++ Containers, Iterators, and Algorithms' or perhaps a book with exactly that as its title.

If you are not fluent with using the STL and need easy access to a memory jogger this book will fit the bill.

Secure Programming Cookbook for C and C++ by John Viega and Matt Messier (0-596-00394-3), O'Reilly, 762pp @ £35-50 (1.41)
reviewed by Alan Lenton

It's a long time since I read a book that had me so enthusiastic after reading the first couple of chapters, only to be plunged into gloom later on in the book.

There are a number of serious problems with this book, although some may find it worth buying in spite of that. So what are the problems?

1. The level of understanding required to use different sections of the book varies wildly.
2. At least one of the techniques discussed is, in my opinion, highly dubious.
3. The inclusion of C++ in the title is a complete misnomer.

I'll look at these problems shortly, but first a little about the book.

The book is an attempt to provide practical advice and code on the security issues facing working programmers today. The first three chapters and the last chapter, together with parts of the networking chapter, provide an excellent look at these issues. They also provide some useful code. I suspect there won't be many C programmers who won't get something they hadn't previously considered out of these chapters.

It is obvious that the author's preference is for *nix type systems, but there is still plenty for Windows programmers. Topics covered in these chapters include initialisation, access control, input validation and error handling.

Chapters four to eleven are about cryptography and herein lies the first problem I mentioned above. There is absolutely no way that a programmer who didn't already have specialist knowledge of cryptography could make use of these chapters. At a minimum you would need to have read – and understood – something like Bruce Schneier's *Applied Cryptography*.

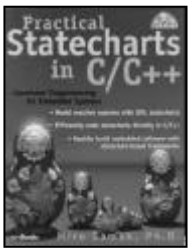
The authors are obviously really into cryptography. At one stage, in the chapter on random numbers (yes, a whole 75 page chapter on random numbers), the authors get so carried away that they discuss and develop code for statistically testing hardware random number generators!

At one level I don't really blame the authors, but what on earth were O'Reilly's editorial department thinking of to let through a book with such wildly different levels of experience needed? The weird thing is that the cryptography chapters would have made a good book in and of themselves. There's certainly enough material – over 500 pages of it.

Chapter 12 discusses anti-tamper techniques and it is here that, in my opinion, the authors stumble badly with a discussion of how to write self-modifying code. Apart from the appallingly bad programming practice that this represents, I would have thought it was just the sort of technique guaranteed to introduce bugs – including security bugs – into a program. Worse still, the technique is introduced as though it was a perfectly normal and acceptable technique, rather than something to only be used in extremis (if then).

Finally, there is the question of C++. The authors themselves concede that there is no use of C++ specific idioms, but argue that the C material is relevant. This is specious. The book does not provide C++ secure programming code, it provides C secure programming code and the title is at the very least misleading.

So, is it worth buying? Well that depends. If you are a C programmer with experience of cryptographic work, then you might well find this book useful. You should certainly consider it if you are looking for a crypto-programming cookbook. Otherwise, there simply isn't enough non-crypto material to justify the price. Either way, if you do consider buying it, make sure you go and look through it in a shop to see if it is what you need before buying.



Practical Statecharts in C/C++ by Miro Samek (1 57820 110 1), CMPBooks, 387pp + CD @ £33-99 (1.32)
reviewed by Chris Hills

This is not so much a book as a way of life. The book is part of the Quantum Programming way of life. It is based on UML state charts and active objects for multithreading.

The best way to discover if this is for you is to spend a few minutes browsing `http://www.quantum-leaps.com` where you will discover lots of information on Quantum Programming, resources and cook books, other books and magazine articles on QP, the QP community and discussion forum. They really have gone to town on this. The CD with the book is no exception. It is a joy to behold. One day *all* CDs with books will be of this standard! This is for both content and execution. It auto starts with a virtual web site logically arranged with links to all the software and resources with notes, links to other resources etc. So this is not 'just a book' on someone's wild ideas. The other important point is that

the web site is (almost) complete. Sometimes many authors promise a web site that never materialises, this one was there before the book.

By the time you have looked at the web site it makes my review of the book superfluous, as it contains much of the subject on the website. The good news is that whilst this system will work with most (they list eight) UML CASE tools it is designed to be a lightweight system that does not need a CASE tool. There is even a set of MS Visio Templates and a coding standard both downloadable so you could 'start with it on Monday' as they say in the book, without a major investment in tools.

For those interested in QP this book will be ideal as a reference. I know many have broadband Internet and could download the source and the information on the website but a book is still better than printouts (even when bound up) and does contain a lot of additional information.

The book itself is in two parts. The first covers UML state charts, real-time, etc. The second part introduces the Quantum methods and ideas. This is an event-message type system.

The examples in the book are designed to work on x86 (under Win9x-2K, DOS or RTOS-32) with VC++ and Borland compilers (all code is on the CD). The evaluation version of RTOS-32 is provided though it is claimed that the QF source has been 'carefully designed to work with nearly any RTOS'.

By coincidence, whilst writing this review there was a brief flurry of messages on `comp.arch.embedded` where someone asked about QP. There were only a few replies but everyone liked the method and had used it on at least one real world project.

The method works for both C and C++ though personally I have always had reservations about using UML for C. As for the book itself; I like it. It is logical, clearly written and the exercises use the code on the CD and work to a practical example. They are not designed as exercises for students where only the lecturer has the answers. If you are using or looking at UML then look at the QP web site first. If you think QP is for you then definitely buy the book. This book is recommended to all interested in QP.

C# & Java

Pure C# by William Robinson (0 672 32266 8), SAMS, 339pp @ £17-99 (1.39)
reviewed by Paul F Johnson

This is not a bad book, but then, it's not a good book and not a good book by quite a way. The code examples are fine, but they do seem to be based around interfacing with an SQL example. There looks to have been a fair amount of thought put into the layout and code examples.

The explanations of the code are adequate (for me, they don't go into enough detail). What the book really lacks though are self-contained examples, there are a couple, but not nearly enough. It is not always possible to create a noddy `main` function to try the code out. In at least one of the examples, the code failed to compile under mono (having been away all week, I've not had a chance to try it under Microsoft C#).

I have two large gripes about the book. Firstly, the appendices are half the size of the actual book (roughly 100 pages as opposed to 200 for the book). This isn't what I would call 'code intensive' – it's the sort of material you can pick up anywhere (descriptions of the various classes and key type references).

My second gripe is inexcusable – there is one of the worst examples of something purporting to be C++ on page 172. It is not possible to compile the code without messing around. It is being used as a comparison of ROT13 between C++ and C#. The C# version is fine...

Overall, it's a book that you would get for a stocking-filler, but that's about it.

Programming C# 3ed by Jesse Liberty (0-596-00489-3), O'Reilly, 689pp @ £31-95 (1.41)
reviewed by Paul F. Johnson

This book falls into the same category as the likes of Ammeraal's '*STL for Programmers*' and '*C++ for Programmers*' books in that the latter contains pretty much all of the former, but with lots more in the way of explanations and examples.

[*I have to say that I strongly disagree with this characterisation of Ammeraal's books.* ***Francis***]

Jesse Liberty's other O'Reilly C# book (*Learning C#*) is really contained in this book. Which is a bit of a gyp if you already have the first one as they cover just about the same core material, but with this one giving more examples, more explanation and with a few corrections in from the original.

Where this does come into its own is on the .NET material. It takes you through from a simple 'Hello World' to some quite complex mini programs. All of them are well explained, well thought out and rather well documented. Jesse has obviously put a lot of thought into the preparation of this area.

Past the .NET material, a lot of time is spent on streams, Internet related material (there is an example of implementing an asynchronous network fileserver, webstreams and asynchronous I/O – all of which are not simple things to implement, but the way the book is presented, the logic is easily seen and the book shows the theory as well).

All that in mind, why only a 'recommended' rating? Simple – it's only an extension of a previous book.

If you haven't bought Learning C# and don't mind paying the extra ten pounds, then this book is well worth the money. If you already have the Learning C# book, then this may not be as good value.

All of the examples compile fine under Mono (with Winelib) and MSVC#.NET 1.0 and 1.1. Recommended.

Programming in the Key of C# by Charles Petzold (0-7356-1800-3), Microsoft Press* , pp418 @ (s)
reviewed by Francis Glassborow

My main concerns with this book have little to do with the technical accuracy or readability of this book. The author is experienced both as a writer and as a programmer. His knowledge of MS Windows puts him at the upper end of the expert class. That means that you can largely take those aspects of this book for granted.

My first problem is that this book is clearly aimed at the novice programmer rather than the newcomer to C# who is already an adequate programmer in one or more other languages. The pace and content of the book would drive anyone with prior experience to skip reading. That would not be helpful. The more I have looked at books aimed at introducing a language the more convinced I have become that it is impossible to write a satisfactory book for those that have never programmed that will also be good for those with prior experience. This book is definitely for the former category but I have some reservations about the author's assumptions as to what such a reader will need to have explained.

I think he oscillates between the temptation to tell the reader everything and recognising that the newcomer does not need nor benefit from such comprehensive detail. Certainly the book does not cover the whole of C#, nor should it given the target readership. The author has some understanding of the needs of novices as is shown by his provision of 'Key of C#' as an alternative to using Visual C#.NET (which the author rightly suspects is too heavyweight for the typical reader). Unfortunately the reader has to fetch that program and, more importantly, the .NET SDK (over 100 Mbytes, which basically requires a broadband connection if you need to get it, which the typical reader probably will). Of course you will also need Windows XP or 2000. I would feel much happier if the required OSs were stated on the front cover and not just buried away in the Prelude. I am also certain that the book should have come with a CD that included all the necessary software. It is available free on the Web so there is no excuse for Microsoft Press not including it with the book.

The text of the book comes in 41 chapters + index and all the usual bits at the front. This tells you that the author has tried to reduce the exercise of learning to program into a large number of small steps. However, anyone with experience of teaching knows that learning requires consolidation, so where are the exercises? To follow through on the author's music metaphor – where are the scales? Theory without practice is no way to acquire a skill.

Please do not misunderstand me; this is a much better than average book for a novice.

However it is sadly marred by the things that are not there. The necessary software should be on a CD and most chapters should include exercises and clearly stated practice pieces for the reader. Without such things the book is only half what a reader needs.

Just Java 5th ed by Peter van der Linden (0 13 032072 2), Prentice Hall, 1094pp + CD @ (no listed UK price) $49.99
reviewed by Pete Goodliffe

There are plenty of books that will teach you Java. What makes this one worth taking off the shelf (besides the nice hologram on the spine)? It's the fifth update of a very popular introductory text that has been 'thoroughly revised' for Java 1.4 and is a part of the Sun-sanctioned 'Java Series'.

The book is vast. At 1098 pages you could use it to prop up your dining table. This huge amount of paper covers far more than the core language. Topics include key library features, server-side Java (servlets and JSP), client side Java (Applets), the UI framework (Swing), database connectivity (JDBC) and more. Let's just say it's comprehensive.

OK, but is it any good? It takes a long time to read from cover to cover, although most readers will probably focus on the language introduction and core library sections, only dipping in to the specific topics if they need them.

The author's style is light and readable, although the layout is decidedly clumsy. The book contains plenty of Java evangelism and the author enjoys Microsoft bashing. I was confused by the random margin graphics appearing throughout and I've still not entirely worked out what they're for. (The dog is cute. Well I presume it's a dog.)

The book lacks a preface, so I'm missing some important information; who is it aimed at? In order to provide a fair review you need to know. The back cover says 'for programmers of all levels', but no book can realistically be written to appeal to every kind of programmer on the planet.

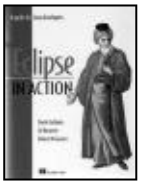
- I don't think this is suitable for totally novice programmers. There are no basic descriptions of the art of programming.
- The book is strongest for 'intermediates' – there is an adequate OO primer, but I'd definitely not recommend this book alone for getting into OO programming. There is an adequate Java language tutorial, although I think there are better ways of presenting this information.
- For the advanced programmer, there are other books that provide a better-paced route into the language.

Linden provides an overview of many more advanced Java topics, but doesn't go into enough depth to allow you to use any of them confidently in production code. Just Java 2 does cover the latest Java language developments. It also explains some of the curious design decisions and historical baggage of the language.

Each chapter has a 'light relief' section, some random 'coffee break' reading that seldom has anything to do with the chapter content. It's amusing, but of little real value. Some of the author's personal opinions in these sections might have been best removed.

The book comes with a CD that includes quite a lot of genuinely valuable content, but doesn't contain any version of the JDK. This is just plain nuts; the book is written for JDK 1.4 and readers with slow dial-up Internet connections do not want a URL reference and a two-hour download when they've paid for an expensive book.

In summary, not a terrible Java book by any means, but at the price I'd hesitate to recommend it over some others.



Eclipse in Action by David Gallardo et al (1-930110-96-0), Manning, 383pp @ (no listed UK price) $44-95
reviewed by Christoph Ludwig

'*Eclipse in Action*' offers a smooth introduction into the efficient use of the Eclipse IDE assuming the reader is already familiar with Java. The authors demonstrate typical tasks by documenting the development of sample applications step by step, but they succeed in conveying the broader picture too. Even if the settings chosen for the examples don't fit the reader's next project the explanations of Eclipse's design enable him to look for the relevant switches in the correct dialogue.

In their description of the various plug-ins, Gallardo et al. manage to explain the very basics of CVS, Ant, unit testing, code refactoring, the logging library log4j and web development. The short introductions cannot replace further reading, of course and a bibliography is among the few things I missed in the book, but the explanations are sufficient to allow even inexperienced programmers to follow the examples.

In the second part of the book the authors turn to a topic that is likely to be of interest to a much smaller group of developers; the development of Eclipse plug-ins. Since Eclipse is completely composed of plug-ins, except for a small core, the background knowledge presented in this part may come in handy if you ever encounter problems with the IDE.

Although three authors wrote the book there is no noticeable break in the language. They manage to keep an unobtrusive and easy to read style. Numerous screen-shots make it easy to follow even if you are not in front of your computer. The book's editorial quality is exemplary.

'*Eclipse in Action*' is not exactly low priced, but if you can fit it into your budget and you want to familiarise yourself with Eclipse then this book is certainly worth reading. Long time users of Eclipse may still find tips they did not know about, but unless they'd like to extend Eclipse themselves – in which case the second part will considerably help them to get started – I doubt the insights the book has to offer them justifies the expense.

Java Performance Tuning, 2ed by Jack Shirazi (0-596-00377-3), O'Reilly, 570pp @ £31-95 (1.41)
reviewed by Silvia de Beer

The book's contents do not exactly correspond to what I was expecting from the title. I was expecting more than only one chapter on profiling tools. The chapter on profiling tools only covers some basic profiling options for the various JVMs. I hoped to find a solid method description of techniques to apply when you encounter performance problems in your Java application. Proprietary profiling tools and their features are not described at all.

If this book is not about profiling tools, what about is it then? Well, it describes potential bottlenecks in your code and gives ideas about how you could optimise your code. Topics include object creation and garbage collection, strings, exceptions, casts, loops, I/O, sorting, threading, collections and distributed computing.

In this second edition, new chapters have been added on J2EE performance tuning, JDBC, Servlets and EJBs. It is a good thing that those topics have been added, but it makes the book rather long if you want to read it from cover to cover. However, these single chapters would not provide enough information if you would have an application heavily relying on one of those technologies; you would need to consult more detailed references.

Some suggestions for how to improve performance are in conflict with good programming practices. I think one should only replace the use of standard Java classes and interfaces with proprietary ones if no other solution is possible to improve performance. Luckily, the author does agree here, but still, in the chapters he gives various examples of how standard Java implementations could be replaced by proprietary ones if needed.

The book is interesting to read, but in my opinion the phrasing could have been more compact. On the whole I found that the book contained useful information and is worth reading.

Bug Patterns in Java by Eric Allen (1-59059-061-9), Apress, 234pp @ £28-50 (1.23)
reviewed by Silvia de Beer

I am very disappointed in this book. The title sounded very promising, a useful and interesting topic. However, the contents do not go into sufficient details. The book is filled with very simplistic examples and advice and it repeats itself too often. For example, the advice that one should use static typing whenever this is possible is repeated in almost every other chapter. I have the impression that the author has decided to write too early on this topic and that he did not take time enough to crystallize out the useful ideas.

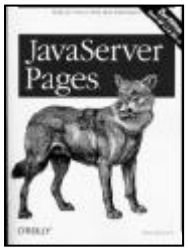
The cover claims that this book is of an Intermediate to Advanced level. I would agree with this, because references to design patterns are made almost without any explanations, which would mean the book is not for complete beginners. On the other hand, some of the

advice is so simplistic, that I would not say this book is aimed at advanced Java users.

One useful piece of advice that I have retained from this book is the warning to be careful with the use of `null`. Methods which return `null` as a flag instead of a typed object have a chance of being improperly used. If the caller forgets to check on the return of `null`, this might cause a `NullpointerException` that is usually very difficult to trace.

The examples that are given do not inspire me, they are too simplistic and seem only to be a filling of space. A few examples extend the class `List` with a new class named `Cons` and it is still unclear to me what the meaning of `Cons` is.

Concluding, the topic of this book is a topic worth describing, but the contents of this book would need a bit more maturity. If you are interested in bug patterns, in my opinion, rather buy a book about refactoring and try to improve your understanding of the Java language. That will bring you more than reading this book.



Java Server Pages 2ed by Hans Bergensten (0 596 00317 X), O'Reilly, 657pp @ £31-95 (1.41)
reviewed by Alan Barclay

This second edition book describes itself as a completely updated and comprehensive guide to Version 1.2 of the JSP specification with detailed coverage of the JSP Standard Tag Library (JSTL) Version 1.0. It claims to be targeted specifically at 'Page Authors' and 'Java Programmers' and anyone who is interested in using JSP technology to develop web applications.

As an experienced Java programmer I am certainly interested in learning about and using JSP technology, but I struggled to get to grips with it from looking at this book. I do not doubt for a moment the credentials and technical expertise of the author but it was a quite long and difficult book to read. It is a very technical book, which I feel needs to be read cover to cover to fully comprehend the subject. It does not attempt to be a comprehensive reference manual or cookbook that can be dipped into.

Primarily the examples are overly complicated at the beginning and rely too heavily on the author's own custom tag library to be easily understood. I don't think that the book provides an easy to understand introduction for newcomers to simple straightforward JSP.

The book does have an associated web site containing the example source code, which I consider essential for anyone wishing to actively work through the book.

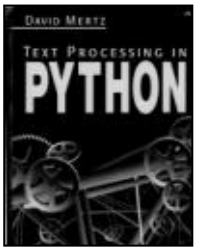
Inside the JavaOS Operating System by Tom Saulpaugh & Charles Mirho (0 201 18393 5), Addison-Wesley, 184pp @ (no listed UK price) $29-95
reviewed by David Alejandro Caabeiro

You might wonder why you should be interested in a failed project (from the market point of view) such as JavaOS. Well, even though the thin-client idea didn't entice in the end, I found this book to be quite a good read for people not only interested in JavaOS itself, but in modern operating systems design.

The authors narrate the evolution of JavaOS from a simple JVM to a microkernel design, describing concepts such as event handling, bootstrapping, memory, interrupts, networking etc. and how they were designed.

The information you'll find is quite different from the one found in classical texts about OS design. For instance, the chapters on device drivers and the JavaOS System Database (which could be described as the portable counterpart to WinNT's registry) are quite interesting. The book is well organised and contains clear diagrams and code snippets to illustrate the concepts discussed.

Python & Ruby



Text Processing in Python by David Mertz (0-321-11254-7), Addison-Wesley*, 520pp @ £37-99 (1.32)
reviewed by Ivan Uemlianin

This is an intermediate-level book discussing text processing. As such it may be interesting to non-Python programmers who do a lot of text-processing and would like to see what Python can do. Its main audience, however, is Python programmers who are comfortable with the language and want to take their programming up a level. I highly recommend this book for these readerships. In fact, anyone who programs in Python would probably do well out of this book.

Apart from a final chapter on the Internet-specific applications, the book's chapters survey, in order of increasing complexity, the variety of text-processing tools available in Python: beginning with 'Python basics', through string methods and regular expressions and up to EBNF parsers. Many sections contain problems and exercises, which can be used either as part of structured learning or just to open out some of the discussions in the text. Appendices give short but informative introductions to Python itself, Unicode and data compression.

The whole book, as well as the code itself and a range of Python text-processing utilities documented in the book, are all available at the author's website, `http://gnosis.cx/TPiP`. Note however, the version on the book at the website is in a custom mark-up language called 'smart ASCII'. This format is quite readable and the site (and book) have code to convert it to HTML (but not to the LaTeX which was used to produce the hardcopy).

The book, including the code, is well written in a clear, unfussy and readable style. Over and above its documentary function, the commentary and examples are lively enough to act as a stimulus.

The book knows its target audience well and sets a brisk pace. There are no long padding sections on 'how to install Python' or on the interactive shell: the appendix on Python is aimed at programmers who use Python rarely; the opening chapter on 'Python basics' is a fresh look at some facilities the Python programmer might be taking for granted. For example, section 1.1.1 on page 1 is called, "Utilizing Higher-Order Functions in Text Processing" (a higher-order function is a function that returns a function). Every topic in the book gets all and only the space it deserves. There are two corollaries to this:

First, no topic is treated shoddily – as if it was only put in to fill space – and so even the most elementary topics are worth reading even for the experienced programmer, as fresh light will be shed or a piece of handy code will turn up. This is even more the case for the 'improving' programmer.

Second, the reader is treated will similar respect. The head of each section tells the reader which topics are treated in depth, which briefly and which not at all; in the latter cases references to documentation are given (e.g., in the Python Library Reference).

As usual with this kind of documentation, facilities are illustrated with a code snippet or two. Unusually, the code snippets here are 'living code': illustrative rather than merely documentary, in that they illustrate the facility in action and suggestive in that the code does just enough to encourage you to experiment with it and turn it to your own ends.

For example, the chapter on Regular Expressions uses the functions below to help the discussion of the `re` module:

```
import re
   def re_show(pattern, s):
      print
          re.compile(pattern,
          re.M).sub("{\g<0>}",
          s.rstrip()), '\n'
   def re_new(pattern,
          replacement, s):
      print re.sub(pattern,
          '{'+replacement+'}',
          s)
```

These two functions highlight (wrap in braces) a match or a replacement in a given text. As well as making the text more readable, they are useful in themselves: handy for learning how regexes work and/or for getting a regex just right before putting it in your script.

The illustrative code given in the section on the Python Standard Library's HTMLParser builds a stack of HTML tags and gets you started on context-dependent interpretation of HTML (or XML) elements. Contrast this with the HTMLParse examples in the Python Library Reference or in [PIAN] which (a) concentrate on the 'tag collection' side of HTML parsing and (b) are difficult to generalise or reuse.

Apart from the inevitable occasional typo, there are three caveats I'd like to raise:

Firstly, the author is a fan of Functional Programming (FP) and uses it (though not liberally or gratuitously) throughout the

book. This might put those who don't like FP off. On the other hand, FP techniques are introduced and motivated very well in the appendix on Python and in the opening chapter. Finally, the reader is not obliged to adopt FP in order to use the ideas from the book: apart from anything else, FP style can easily be converted to more traditional Python style, e.g.:

```
add = lambda x, y: x + y
```

is equivalent to

```
def add(x, y): return x + y
```

Secondly, given that this book was clearly typeset with LaTeX, I was slightly disappointed to see nothing on LaTeX processing with Python. As the author says on his website, this may be to discourage too many downloads from his website. Notwithstanding the 'need' for DRM, the absence of any discussion of scripting extensions or wrappers for typesetting languages like LaTeX is a flaw.

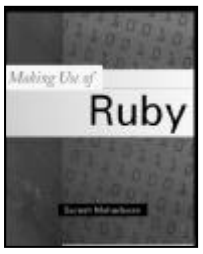
Finally, the book has a slight bias toward text processing understood as 'processing textual input'. This is understandable of course and no doubt correct. However, the book is correspondingly weak on 'processing textual output'. For example, section 5.2.2 "Parsing, Creating and Manipulating HTML Documents' actually covers only 'parsing' (the sections on XML are also limited to parsing). Similarly the discussion of email concentrates narrowly on reception rather than generation. While it may be fair to say that generation is simpler than comprehension, some discussion would have been useful.

The remit of this book is broader than its title might suggest: first, text processing covers a lot of things (e.g. handling markup languages, Internet protocols, even bioinformatics); second, the way the book is written encourages the reader to explore the ideas. Like a good monograph, it uses its focus (text processing) as way to illuminate the whole field (programming in Python).

References:

[PER] Python Essential Reference 2nd Edn. D. M. Beazley. New Riders. 2001.

[PIAN] Python in a nutshell. A. Martelli. O'Reilly. 2003.



Making Use of Ruby by Suresh Mahadevan (0 471 21972 X), Wiley, 216pp @ £25-95 1.35)
reviewed by Giles Moran

This is a gentle introduction to the Ruby scripting language. For those who haven't heard about Ruby, it's an object oriented scripting language developed by Yukihiro Matsumoto of Japan in the mid nineties. Ruby is very popular in Japan, but there's not a lot on the web about it when compared to say, Python. The elegance and clarity of the syntax is much vaunted but it still has its share of odd syntax. More information can be found at `www.ruby-lang.org` or `www.rubycentral.com`.

The book is very short, standing at 216 pages (including index and appendix) and employs a very large font, which reduces the amount of information on each page. There are enough typos to make reading hazardous (the regular expression section suffers especially) and indentation of code examples is inconsistent at best. The book has an ongoing theme of implementing a web package for a hypothetical book company augmenting chapters on classes, io, etc. The trouble is that half of the chapter is taken with this single example case and there's not enough detail on the syntax given.

Once you've read this book, if you like the look of Ruby, you will need a reference straight away. There's not enough information in the book to make it remotely usable. The book also dwells on the simple stuff like loops and if statements for too long, while skipping a lot of the more demanding/interesting detail in later chapters.

You can do some fairly interesting stuff with Ruby. Access to threads seems straightforward. You can build GUIs using Tk.

You can also embed Ruby into HTML using eRuby. I'm sure there's a lot more, but I need a better book to enlighten me.

Embedded & Real-Time

Embedded System Design on a Shoestring by Lewin Edwards (0 7506 7609 4), Newnes, 232pp + CD @ £35-00 (1.43)
reviewed by Chris Hills

The title is a little misleading as it is not a general embedded book. It is specifically on the ARM7 using Linux and Gnu. To use this book you will also need the Atmel AT91EB40 evaluation board. Actually this book will probably end up being supplied with the Atmel kit so check with your Atmel supplier before buying the book. That said, the first two chapters, about a third of the book, are very good advice for all embedded developers. After that it goes in to Cygwin and the Gnu tool chain for the middle third of the book, which restricts its usefulness a little, and the final third of the book is debugging on the ARM7 AT91EB40 board. My resident ARM engineer said it was not bad as an introduction to the Gnu tool chain.

It is a relatively short book; I have longer texts on my web site (but not on this specific subject). I was not taken by the book when I first looked through it, since much of the information is in the Gnu tool chain manuals and other freely available places; but then again so is almost everything else if you look for it. I warmed to it more when I read the first two chapters as these have a lot of useful information for the student (and indeed new working engineers) on how to make decisions in industry. This is not the same as the theoretical choice you might make in academia. There are some very useful tips on some pitfalls that could kill a project.

The author bought all the equipment personally without telling the suppliers it was for a book. Thus he knows the real costs and the level of support available to the average user. If the author could add more meat to the book and some more advanced topics it would be a good book. That is not to say there are not some useful things in it now, writing to flash for one. That is always one of the first problem areas to be encountered after you get the compiler to actually compile something.

From the comments in the initial chapters the author certainly understands the industry. Also whilst this book (written in the US) is, like most, US-centric, the author (an Australian) has made the book accessible to others. I like the style and way the book is written. As a plus point the full text is on the CD as a PDF. A very good idea and a brave decision by the publishers.

However I do not think that there are enough plus points to warrant buying this book unless you have the Atmel kit. As an addition to the Atmel kit it would be perfect. So on the whole, sadly, only recommended for Atmel EB40 users. Note to author; please do an expanded second edition!

MicroC/OS-II: The Real-Time Kernel 2ed by Jean Labrosse (1 57820 103 9), CMP Books, 606pp + CD @ (no listed UK price) $74.95
reviewed by Chris Hills

This is not so much of a review of a book but a review of a book, an RTOS and a web site. It's the second edition of the second version of a 'free' RTOS. It is the book worth buying? **Yes**.

MicroC/OSII is an RTOS with the entire source available as part of the package. Whilst it is free for most of us, including colleges and universities, there is a licence if you want to use it commercially, which seems fair. See `www.micrium.com` for licensing.

So why use this instead of Linux or any of the others? MicroC/OS has been designed as a small footprint real time, pre-emptive OS that was designed for embedded use on 8 bit platforms upwards. It has also been approved for use in a DO-178B aerospace system and is (apparently) MISRA-C compliant.

Accompanying the book is a web site where people post ports to other CPUs, the executable in the book is for x86. There are many ports and you will often be faced with a choice of ports to any processor you choose (8051 to ARM). These are all free and donated by their authors. Due to the modular form of the design only a few well-defined files need changing for most ports.

The book itself is a heavy 600-page hardback. Much the same size as the 500-page first edition but there are a lot of changes. Most of the source code has been dropped, as has the chapter on porting from the first version of the RTOS. There has been a major expansion on inter-task communications and

synchronisation. The software itself has gained mutexs.

In explaining the API and rtos the book does a good job of explaining how, in general, an OS works. This is achieved via diagrams and extracts of code. The main manuals for the OS are on the CD in pdf form. The book itself has a guide to the chapters showing what to read for Concepts, Structure, synchronisation, communications, porting and the user manual. In short it is a complete RTOS kit. Students and lecturers will find this book very useful. That said I know of a least one serious commercial development using MicroC/OSII in a serious embedded system.

The book is an excellent entry into operating systems. You will need another book if you really want to go deep into the theory of operating systems, but for most users who want an OS to use and experiment with this book will fit the bill. Highly recommended

Software Engineering for Real-time Systems by Jim Cooling (0 201 59620 2), Addison-Wesley, 787pp @ (no listed UK price) $71.00
reviewed by Chris Hills

Some five years ago Jim wrote a book with a title similar to this one. It was a good book but over the years he thought about a second edition to bring it up to date. The first book had 450 pages, this one is just short of 800; it has been more than just updated, hence the new title.

This book is about software engineering and its processes, tools and methods, not writing software as such, so there is very little source code in the book. Example software is usually on the web sites of the silicon and tools vendors anyway.

The book covers all the areas needed for developing and testing real-time (and embedded) systems. It is split into three sections, foundations, designing and developing, and finally implementation and performance issues. It follows much the same lines as many software engineering books but it does it from the perspective of a system that controls real moving machinery. This becomes more apparent in the section on development and debugging tools where things are very different to desktop systems.

The design sections highlight some of the unusual and problem areas in embedded and real-time work, also some of the blatantly obvious ones that are overlooked. This is done from several overlapping views, which highlights that there is no one true method and why you may need more than one view of a project.

I was pleased to see the chapter on 'why diagrams' discussing the use of diagrams and graphic designs systems per se, is still in the book. This looks at everything from electronic schematics to use-cases and assembly diagrams. Another discussion chapter is the one on metrics (part of testing) where again a pragmatic common-sense approach is used.

One of the surprises to some will be the inclusion of structured methods (Yourdon). This because a large amount, probably a majority, of real time development is done in C. Some is still done in assembly! OO methods are also covered as well. The useful thing with this book is that it does not push any one method or language but explains the relative merits, when you would use the methods and why.

This book will not teach any of the methods in depth, but puts them into perspective. It gives overviews and shows how they all fit together. Some of the overview diagrams would go well as posters on the wall (especially for students).

This book seems to cover everything. You will of course need books on the language, design methods and specific protocols and algorithms you are using. This book will be invaluable to engineers (and students) moving into embedded and real-time work or as a reference for those already working in the field for inspiration and ideas. I shall keep my copy close to hand. Recommended

Building Embedded Linux Systems by Karim Yaghmour (0-596-00222-X), O'Reilly, 391pp @ £31-95 (1.41)
reviewed by Michael J. Pont

In the preface to Building Embedded Linux Systems the author Karim Yaghmour states that '*This book is intended first and foremost for the experienced embedded system designer who wishes to use Linux in a future or current project*'. Does the book match these intentions? Not completely, but it is – nonetheless – a useful book.

Prospective readers should be aware that Yaghmour is concerned with what are (from the reviewer's point of view) fairly large embedded systems, with a 32-bit processor linked to at least 2MB of ROM and 4MB of RAM. X86, ARM and PPC hardware are the main platforms that are discussed.

It should also be noted that the book is primarily concerned with 'soft' real-time designs and issues such as scheduling mechanisms in Linux are not discussed. Furthermore, issues of interrupt latency are only considered right at the end of the book and here no figures are given (instead some simple techniques for measuring such latencies are presented).

In the discussion of interrupt latency, Yaghmour suggests that if the interrupt response of Linux is not sufficiently fast, one of the real-time derivatives of Linux should be chosen as an alternative. These variants are discussed (briefly) at the start of the book, in a section focusing on licensing (and patent) issues. If you intend to use Linux in a commercial project, then Yaghmour's discussion of the legal implications of doing so may prove to be valuable.

Another issue that is covered towards the end of this book is the tracing facility provided by Linux; this is not widely used on desktop systems, but it provides a useful way for embedded developers to explore how their applications interact with each other and with the kernel.

Overall, if you have some knowledge of Linux, some experience with embedded systems and wish to use a 'standard' OS in a comparatively large system with soft timing constraints, then this book will prove very useful. Recommended.

Games Programming

Game Coding Complete by Mike McShaffry (1-932111-75-1), Paraglyph Press*, 580pp @ (no listed UK price) $39.99
reviewed by Paul F. Johnson

Ever wanted to know how to write a game, the processes behind it, the planning and just about anything else you could need to know? Then this is the book for you. Unless you're not interested in Win32 and DirectX that is, in which case, it's still a great book – you just can't run the software.

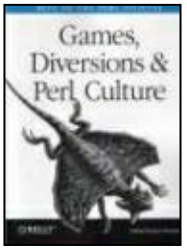
The author is one of the team behind the Ultima series of games and has been on the series since the start, so should know a thing or two about what goes on behind a game.

This book is well presented and contains a lot of helpful information on getting the most out of a game code. It is refreshing in today's environment of bloat acceptance and games needing the highest spec'd hardware available that the approach taken is for general machines and tight code. The author is to be commended for taking this route.

Too often those using early P3 Windows machines/late P2 machines (or even those with moderately high spec Linux boxes and run WineX to play games on) are left out due to someone deciding early on that the game will only run at a high bit-depth resolution, with >256Mb memory and the latest nVidia graphics card and Soundblaster technology (some of which may not be currently available to the general public).

Another refreshing approach is in the actual writing style – it is very down to earth and treats the reader more as a friend than anything. You could actually imagine part of it taking place down the local pub on backs of envelopes! It is yet another reason to get the book that even with this style, it is informative.

Where the book does fall down on is that it doesn't cover in any depth the use of extra applications used in modern game development (such as 3D Studio Max and sound software). I can understand why the author has done this – it would increase the page count from a respectable 563 pages to something approaching the OED or would have to be seriously hacked down in order for the page count to be something sensible. Highly recommended.



Games, Diversions, and Perl Culture by Edited by Jon Orwant (0-596-00312-9), O'Reilly, 569pp @ £28-50 (1.40)
reviewed by Mathew Davies

This book is the third in a series containing collections of articles from The Perl Journal.

Different chapters are written by different authors in slightly different styles. The editors have cunningly pulled together a diverse range of articles under the three headings in the title.

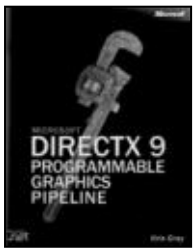
To begin with, I was a bit apprehensive about reviewing this book; I am no Perl guru, though I do occasionally use Perl to search or manipulate text files. I needn't have worried. A lot of the book requires little detailed knowledge of the Perl language. Moreover, a lot of the points that the authors make apply to programming in general.

To begin with, I made the mistake of trying to read the book sequentially, starting at the first chapter and working my way through to the last. That was a bad move. I soon realised that it was far more enjoyable to jump around randomly between chapters, as the mood took me. I was able to do this because the forty-eight chapters are largely independent of one another. Also, most chapters are quite short, falling well within the reading time available on my daily commute back and forth to work by train.

I suspect that no two people would agree on their ranking of chapters in this book. My own favourites were a delightful account of the Infocom z-machine, an intriguing description of how Perl was used on the Human Genome project and the somewhat bizarre collection of Perl one-liners. Chacun a son gout!

I don't think that I'd have bought this book before reading it, but I'd have been very happy to have received it as a present. In summary, it's a charming coffee table book for the computer literate reader with a little time on their hands!

Other Programming



Microsoft DirectX 9 Programmable Graphics Pipeline by Kris Gray (0-7356-1653-1), Microsoft Press*, 454pp + CD @ £36-99 (1.35)
reviewed by Daire Stockdale

Written by a programmer from the DirectX team, this book covers programming pixel and vertex shaders using DirectX 9 (DX9). Assembly and High Level Shading Language (HLSL) are covered equally; we are introduced to assembly vertex and pixel shaders with quite straightforward examples and then are shown HLSL language versions. The book comes with a CD, which includes the DX9 SDK and a set of tutorials and has colour plates illustrating the examples used.

My initial impressions of the book were very high; it's concise and clearly written, something I always look for in a technical book. It assumes a reasonable minimum knowledge on the part of the reader. The book is for a programmer with experience of both Windows and DirectX programming and it concerns itself only with the details of getting the shaders working. The book is informative and definitely advanced my knowledge and confidence in the subject. All the shader versions are covered too.

I was disappointed to find several chapters on using DX 'effects', as I do not consider these to be true components of the DX graphics pipeline and I would have preferred deeper coverage of the uses shaders can be put to. Other little niggles about the book include some typos in the code, which highlight the fact that the code contained in the book has not been compiled.

A large portion of the book is made up of its appendices. These cover an overview of the pipeline's vertex processing and references for the asm shader languages, the HLSL languages and the effects. If these references were thorough, this alone would make the book a useful desktop companion. Instead, many of the references that I consulted gave a short description of the term and then said 'See SDK Reference page for details'. I was also unable to find the details of pairing for the assembly language commands.

In summary, the chapters of the book that are good are very good. Kris Gray covers vertex and pixel shaders written in assembly and HLSL well enough to give a reader with some experience of DirectX and graphics terminology the confidence and understanding to put this technology to use. However shaders are not covered in the depth that they deserve and much of the book is reference material that can be found in the free DirectX 9 Software Developer Kit.

XML Primer Plus by Nicholas Chase (0-672-32422-9), SAMS, 992pp @ £32-99 (1.36)
reviewed by James Roberts

My opinions of this book varied wildly as I ploughed my way through its 1000+ pages. It started well with a very readable coverage of Basic XML document structure. The explanations were clear and the book handles examples well, by building up complex examples step by step, using bolding to show the changes between consecutive examples.

The chapter on DOM (number 3) is where my main criticisms started. There is a reasonable description of the technology buried under a pile of examples. Each point tends to be illustrated in Java, VB, C++ (oops, Microsoft Visual C++), Perl and PHP. I felt that these examples might be useful, but I would have liked to see most of them relegated to a website. To take their place I would have liked to see some of the API definitions.

These issues annoyed me (and annoyed me throughout the book). However, I persevered, skipped most of the examples and skimmed the rest of them. This (for me) improved the book and turned it into an interesting (if over-heavy) summary of XML and a vast array of associated technologies.

Refactoring Workbook by William C. Wake (0-321-10929-5), Addison Wesley, 235pp @ £26-99 (1.30)
reviewed by Francis Glassborow

I like the idea behind this book, to provide an understanding of refactoring by example and exercises and I think many programmers could benefit from such a study but the book has too small a target readership. The problem is that many of the examples come from XP (Extreme Programming) sources. That is irritating, but more to the point is that they are Java ones. I think that programmers from other languages (such as C, C++ and C#) will be constantly distracted from the main thread by the examples and exercises being in a language that is subtly different to that with which they are familiar.

I completely understand that it is very difficult to provide multi-language coverage in a practical book but there are ways of addressing this. Many of the examples/exercises could have been at function rather than class level and thereby have a far greater degree of commonality. I think that it could be possible to provide alternative examples in other languages even if those were mainly in an electronically readable form (CD or website).

On the positive side the author's classification of code smells is useful and helps the programmer to focus on particular aspects of code that may lead to either simple, in place, changes or to refactoring.

If you program in Java working through this book will almost certainly improve your code. If you are not reasonably fluent with Java I think you should be encouraging the publisher and author to revisit the issue with an alternative edition. I think the loss of value for those who mainly program in another language is too great to justify the cover cost.

Beta Testing for Better Software by Michael Fine (0-471-25037-6), Wiley, 284pp @ £29.95 (1.34)
reviewed by Paul S Usowicz

This book is about implementing a beta test program for your software. The author used to be the beta test manager for 3COM and his knowledge and experience of the subject is obvious throughout the book. No specific software applications are discussed with all the information being relevant to most software regardless of size, language or platform. The book covers all aspects of the process and is cleverly divided into three separate sections labelled 'Understanding Beta', Building a Beta Test Program' and 'Making the Results Work'. Each section covers a distinct area of the testing and is aimed at slightly different readers. It would be unfair to say that it is three books in one as they all really need to be read, particularly if you currently have no beta test program in place.

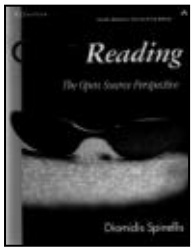
Section 1, 'Understanding Beta', gives a good overview of what a beta test is and what you should expect from one. This section is aimed at people who either do not currently have a beta test program in place or for people who have no intention of implementing a program but wish to understand what it is (your manager?). I found the information presented here got me

up to speed so that the next two chapters did not overwhelm me.

Section 2, 'Building a Beta Test Program', is where some meat is placed on the knowledge gained in the first section. This section is also the largest taking up about half of the book. Topics covered include the step-by-step process of creating a beta program, selecting participants and gathering the data. If you have never carried out a beta test before then the information here is invaluable. No stone is left unturned and there will be no guessing as to what to do next or who to do it with. My favourite chapter in this section is simply called 'The Bug'. This chapter can be summarised by the author's own definition "A bug is defined as anything that could potentially have a negative impact on the customer's experience with the product'. Now that is what I call common sense.

Section 3, 'Making the Results Work', is a very small section but nonetheless essential for a full understanding and implementation of a beta test program. This section covers what to do with the data once you have amassed it. Various topics are touched upon including how the data can be used by various departments and how cost savings can be achieved.

Overall I found this book easy to read and very informative. If you are planning to introduce beta testing or find your existing beta tests lacking then this book definitely deserves a place on your desk.



Code Reading: The Open Source Perspective by Diomidis Spinellis (0-201-79940-5), Addison-Wesley*, 494pp + CD @ £37.99 (1.32)
reviewed by Tim Pushman

As stated in the introduction, how many novelists, painters, surgeons, even airline pilots, have learnt their trade by studying the work of others before them? Yet, in computer science, we are usually taught how to write programs, libraries and so on, with barely a reference to the skills of the masters before us. When much code was proprietary it was sometimes difficult to find code to guide us, but in this age of open source software, we now have a wide range of code to study.

The author suggests that reading code could be a standard part of a Computer Science curriculum and this book is an attempt to formalise the process, based around some samples of open source software.

The book can be broken down into four basic parts.

The first part gives an overview of the C language (and to a lesser extent C++) as used in open source software, with examples taken mainly from the source of NetBSD. It provides a brief synopsis of the various paradigms used in open source software as well as a short explanation of the syntax of C as it is used in the source base. C programmers will probably skip this section, although it is probably worth browsing through just to see how the Gods of BSD have tackled things.

The second and main part of the book covers the overall architecture of project, directory structures, build processes, coding standards and documentation of large, open source projects, wrapping up with a brief overview of the tools a programmer will use to find his way around a complex software project.

Of necessity, parts of this section are all too brief, which is a pity as this is really the meat of the book. It may have been better to skip the short overviews of object oriented programming, remote procedure calls and event based architecture and concentrated more on the open source way of approaching these subjects.

The third (and shortest) part is the fun part, Spinellis walks us through the process involved in adding functionality to an existing project ('hsqldb'), from first unpacking the code base through to testing and documenting the changes made. Anyone who has done this sort of thing will be nodding their head at some of the blind alleys and mistakes to be made on the way.

The final part is the usual set of appendices, which in common with many computer books these days, seem to serve no useful purpose other than to bulk the book out (in this case by 150 pages). Do we really need all the software licenses from the CD printed out?

Some areas are weak in the book, for instance, more on handling inter-library dependencies to get the projects working in the first place. There is very little on the KDE or Gnome frameworks, which is a pity because many programmers will start with these kind of projects, rather than network drivers and the like. There is very little coverage of the autotools system, which is the basis for most new open source projects (automake does not even appear in the index) and these omissions give the book a somewhat 'old-fashioned' feel, covering systems that were put together in the days of 'roll your own' build systems.

On the physical side, the book is well made, lies flat on the desk and the text is clear, well laid out and well organised, with a good index.

So who is this book useful for? If you are considering putting together an academic course on computer science and want to include the art of code reading, this book would be an excellent starting point. Each chapter includes a set of exercises that are obviously designed to be used in a classroom situation and the book follows a solid logical path through the subject. The course may want to skip the C overview and move the section on tools to an earlier part of the course.

Another group that would gain immensely from reading the book would be Windows programmers making the change to *nix systems, or who simply want to understand the code. The book provides an excellent overview of the whole system of open source software, from checking out of cvs through to the release process and would make a good primer before heading off into the wilds of 'apache.c'.

So, if you belong in these two groups, this book comes recommended. For all others, an interesting read but not essential.

Lessons Learned in Software Testing by Cem Kaner (0 471 08112 4), Wiley, 286pp @ £29-95 (1.34)
reviewed by Chris Hills

This is a somewhat unusual book. It is made up of 293 'lessons learned' in eleven chapters. Each lesson is self-contained. The chapters range from 'The role of the tester' via 'Bug Advocacy' and various types of testing to 'managing a testing group' to an interesting section on 'Your Career in Software Testing'. Yes, strange as it may seem to programmers, some people do have a career in testing software (and I don't mean they are end users).

The lessons range from three lines to a page and a half. Though, as with source code, their worth cannot be measured by line count. Lessons vary from specific points to interesting techniques to philosophical approaches, quite a mixed bag. It is clear that the three authors have 'been there', in fact there is a lesson on what to do when a testing professional meets a manager with a deadline who wants to skip things. The format of the book means that it is pure information with very little waffle. There is not the usual narrative between points, though each of the 11 chapters does have a brief introduction.

The "software testing" is general software testing and documentation. It does not cover embedded software or related hardware testing techniques, which are a whole different ball game.

My only problem with the book is its usefulness or how to use it. I have similar books on Chinese and oriental wisdom, they too are fascinating and contain much wisdom. The problem is remembering the right proverb at the crucial moment. This book, that has many very useful lessons, will make fascinating reading but will you remember the right lessons when you need them?

I think that you need to go through the book as you start a project and plan the testing. Noting down in your notebook the number if not the text of those lessons you want to use and at which points in your project. I would also advocate keeping this book to hand during the project to read in those quiet moments for inspiration and confirmation that you are not the only one who had that last problem. Highly recommended.

RTF Pocket Guide by Sean M. Burke (0-596-00475-3), O'Reilly, 146pp @ £8-95 (1.45)
reviewed by David Ross

I saw Francis' review of this book in the last issue of C Vu and thought I'd add my

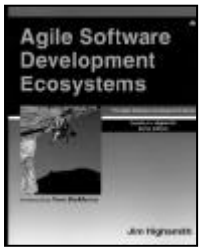
comments. First I agree with Francis that this is a short book and in part this is because there really isn't much to say.

There are 126 pages, not counting some reference tables and indexes, split into three sections. The first covers the RTF format. The book struggles in the sense that it quickly becomes clear that the format is not the most structured or even that rigorously adhered to by many applications. The way I used the book was to take some of the examples and slice and dice them to get what I wanted (and I suspect that's pretty much what most people will do). In this way I was able to generate some RTF reports for my application that could be rendered by the rich-text box component in .NET and in Microsoft Word.

The second section covers the generation of pretty basic Windows help files using a slightly modified RTF format. The final section provided some example applications, which demonstrate the concepts discussed in the first section. To be honest I'm not sure of the actual value of these last two sections. Help files are probably best produced using appropriate tools and the applications are fairly basic – but they do provide more examples to slice and dice.

That said, this is one of the few books out there for RTF. It is readable and has the advantage that you can read it very quickly, which makes a change these days. More importantly, you end up producing working RTF pretty quickly. Any weaknesses in the book seem more down to the subject matter than anything else.

Methodologies



Agile Software Development Ecosystems by Jim Highsmith (0 201 76043 6), Addison-Wesley, 404pp @ £34-99 (1.29)
reviewed by Mike Higginbottom

This book is not a prescriptive list of bullet points. It is a gentle and sometimes rambling and repetitive journey into the philosophy and attitude of the proponents of agile development methodologies. Jim brings us into this world, in a very real and balanced sense, by coming at the subject from a huge variety of angles.

He draws on considerable experience, both his own and via interviews with agile colleagues, that of others. He talks about agility in the wider business and economic sense. He brings in chaos theory to highlight the need to balance on the cusp between order and disorder in a fast paced world. He discusses collaboration, customer value, pride in the craftsmanship of the product, evolutionary design and generative rules. He talks about replacing process and control with freedom and discipline, but the pervasive thread running through all this is an emphasis on people and the way they think and behave. This is the central message of the book. Process won't save you; but your people just might if you let them.

The book is an effort to immerse the reader in agility. As a consequence, there is much repetition. I suspect this is entirely intentional and, although I found it irritating at times, it is an approach that works. As you read, the principles will diffuse into you and there are enough 'zen slaps' to keep the narrative flowing and the interest alive. My only other criticism would be the lack of explicit comparison between the alternative agile methodologies. Again, this would seem to be intentional. This book is not designed to lead you to the one true way but rather to open your mind to the possibilities that agility offers.

There is little in the way of detailed process description; this is a book about principles rather than practices. You will not be able to read this book and suddenly have your project 'go agile'. You will be able to read this book and be in a position to decide whether agile is the way for your project to go. The implementation is left to other resources.

This is a must read both for those new to agility and those who are already operating in an agile environment. Highly recommended.

Extreme Programming Refactored by Matt Stephens and Doug Rosenberg (1-59059-096-1), APress, 400pp @ £28-50 (1.40)
reviewed by Francis Glassborow

This book takes a long hard and completely irreverent look at the Extreme Programming methodology. I guess that it is going to cause some people to spit blood and others to dismiss it as bad technical writing but for this reviewer it is a long need breath of fresh air.

The authors' style incorporates satire, song writing, industry experience, as well as a solid analysis of the elements of XP. The writing style makes it easy to read and the analysis and experience make it worth reading.

XP is a particularly rigid example of Agile programming methodologies that was designed by Kent Beck. When he presented it in eXtreme Programming Explained (ISBN 0 201 61641 6) he was very firm that all the various elements were essential and must not be messed around with. Perhaps more significant is that the book was published in late 1999 (and largely written in the preceding two years) and the seminal project was the Chrysler Comprehensive Compensation or C3 project to develop a new payroll system to replace the existing one which was expected to have Y2K problems. The project was started in 1996 and had a clear absolute deadline of 1/1/2000. In fact the project was cancelled in 2000 having never handled more than a third of the Chrysler payroll.

I think that Kent Beck's original book was worth reading and certainly extended the range of possible discourse on programming methodologies. However it has never seemed like a silver bullet and initially masked the more general view of Agile methodologies.

One of the interesting things about this book is that it isn't simply a diatribe against XP practices but does take Kent Beck's work seriously if only to disagree with much of it. The early chapters take the various principles of XP and hold them up to the light so that we can better see the holes and faults in them.

In chapter 14 the authors tackle one of the major problems with XP, it simply does not scale. Actually I note looking back at my review of eXtreme Programming Explained that I did not think it would (interestingly the C3 project was larger than my estimate of where XP would work though I was unaware of that at the time).

In the final chapter (15) the authors carry out the process of refactoring XP by seeking to extract the good ideas from the monolithic process that Kent Beck described. While I would not agree with everything I find there, I do think that their 'defanged XP' recovers a good deal of the agility that I expect from an Agile methodology.

If you are interested in the software development process and are happy with technical writing that leans on satire to make its points then this is a book you will both enjoy reading and benefit from having done so.

Web & Internet

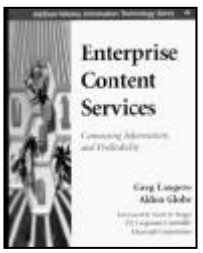
Learning Web Design 2ed by Jennifer Niederst (0-596-00484-2), O'Reilly, 454pp + CD @ £28-50 (1.40)
reviewed by Francis Glassborow

I tried this book because I needed to work on a website to support my own book. I had hoped that it would be helpful by pointing me in the right direction for many of the things I would need. In the end I just became increasingly irritated by the things it failed to tell me or the places where it gave either wrong or gave clearly out-of-date information.

There seems to be an assumption among writers of books on this subject that either the reader wants to know nothing more than how to produce a simplistic home page or will want to lash out on a copy of Dreamweaver at over £200.

In the end I decided that it must actually be possible to write a book about setting up a good website from scratch but this definitely was not it.

Management



Enterprise Content Services by Greg Laugero & Alden Globe (0 201 73016 2), Addison-Wesley, 180pp @ £22-99 (1.30)
reviewed by Christopher Hill

This book is written to challenge the view that 'Managers have a deeply held belief that technology is about automation and that automation replaces management'. In fact we need new ways of managing to get the most out of the technology – to let strategy drive technology.

Rather than simply throwing the latest search engine or document management

system at the problem, take a step back and consider what the enterprise (business) actually requires.

Each department producing its own 'helpful' information as little islands of knowledge fails to provide useful returns on the effort because they are just reproducing the paper-based system.

Better to consider Use Cases (the book calls them lifecycles and storyboards). By focusing on the lifecycle for a key business entity (customers, products, staff and services) the discussion stays at business level rather than descending to department or technology level. Once the lifecycle is firmed up, then tasks can then be assigned to departments and technology – the dog is now wagging the tail.

First a common language must be agreed to enable to identify and later find content. Versions and editions of documents need to be handled consistently across a range of media. Entities need to be identified consistently across the company. Security of access is also required; who can edit, who can authorise, who can read.

The final section of the book considers the roles within this vast project; the champion, the decision maker, the solution designer. This section would be useful in understanding any larger project.

The book addresses the need for an overarching vision, an enterprise taxonomy and standards for navigation to avoid the messy collections of disconnected web sites that beset many large companies.

Whilst the book is aimed at large companies, the issues raised need to be faced at all levels of content management. Recommended.

Project Management: A System Approach to Planning... by Harold Kerzner (0 471 22577 0), Wiley, 1192pp @ £51-95 (1.54)
reviewed by Chris Hills

This is not a software engineering book but a generic project management book that has an engineering bias. Any book that survives to a seventh edition must have something going for it. No, wait, there is an eighth edition that was published a month ago! I picked up the review copy of the 7th edition at a show a couple of months back. It appears that editions seven and eight were printed just two years apart. I discovered this looking on the web, as the book I have (the 7th edition) has no history of the previous six editions at all. I find this strange, that a book on project management has no revision history.

This is designed in part to be a textbook and there is a separate lecturers' resource and workbook. This is (it appears) on free download from the Wiley web site, via password protection. (see `www.wiley.com/kerzner` for information). There are also questions and problems at the end of each chapter. There are no solutions for these in the book. The author maintains that the book and the workbook together make an excellent self study guide for the US Project Management Institute's Certification Exam, though how you are supposed to do this when the workbook is only available to lecturers I am not sure. That said, the questions at the end of the chapters are often there to provoke thought rather than a single quantifiable answer.

It is a solid book and not just for its 1200 pages and hard cover. There are plenty of illustrations, diagrams and cartoons; many I suspect are part of the lecturers' resource kit. They certainly look like part of the Project Management Courses the author runs, but this is not a picture book, there is much detailed and highly informative text. However, I think that this book will stand on its own without the additional courses.

That this book is used for courses shows in that things are explained clearly and with diagrams. Therefore the average reader, new to project management, will be able to follow the techniques and numerous case studies from many areas of industry; between them they highlight many things that are not obvious and are quite thought provoking. As the book covers virtually every area of project management it covers areas that many project managers will not be directly responsible for, which is useful as it will give an appreciation of the problems of other people that the PM has to interface to.

There are some useful comparisons between methods and the pros and cons of each. As with most things there is no silver bullet. This book will supply you with all the information and arguments. You still have to decide which techniques to use and how to run your project!

Overall I think that this is a good book. However it is more applicable to the US than to Europe. One or two illustrations would never (ok, are highly unlikely to) occur this side of the pond. So with that in mind this book is recommended for US readers and recommended with reservations for Europeans, but check out the 8th edition first.

Configuration Management Principles and Practice by Anne Mette Jonassen Hass (0-321-11766-2), Addison-Wesley, 370pp @ £37-99 (1.32)
reviewed by Chris Hills

Configuration management, or version control as it is sometimes known, is an area that is often overlooked by programmers. It is regarded in the same light as documentation, procedures and testing. However it is a very important part of any project. This has become more apparent with the Linux community; which has many people independently working on many versions of the system. Though the principles also apply to a one-man one-computer project. Comprehensive is the word I would use to describe this book as it covers the whole subject almost to the point of project control. I don't think the book is aimed at the one-man project unless he has lots of small projects.

Although I am a great fan of configuration management I found this book surprisingly heavy going. Whilst it defines everything in minute detail, the definitions are wide and often completely open. There is a lot of it may be this or this and this and additional things may be added. Some things appeared to be stating the obvious. I found myself sinking into a morass rather than a clear direction.

For example, one area of confusion is that version control is not configuration management. CM is a lot more than just version control. This is fine if the book stopped there and laid down some rules. The problem is that the book then goes on to say; it is up to you how you define the two terms and that in some companies they are used interchangeably without any problems. The author should have clearly stated a method and stuck with it.

I was surprised to find the 'general principles' section was chapter 15! I did try to like this book and did find useful information in it, the W-model for one thing, though this was a just a diagram with virtually no explanation.

On reflection this is more of an academic (not teaching) discussion on CM rather than an engineering book. All the information is there, it is jut not clearly and easily accessible. I think this is a pity. This book could do with a re-edit to make it more accessible to team and project leaders. I would hope that the author does this as a good [readable] book on CM should be on every software engineer's bookshelf.

Tools for Success: A Manager's Guide by Suzanne Turner (0 07 710710 1), McGraw-Hill, 200pp @ £9-99 (1.50)
reviewed by Chris Hills

This is a strange book. Basically it is an annotated list of methods (or tools) and a tool in itself. What it does is list 95 tools or methods used in management with the essential notes. Note these are methods in the sense they are a way of doing things. There is not one of them that actually require you to buy any software packages or other tools bar flip charts, marker pens, paper etc. Though for some something like MS-Project or Visio might be useful but not essential. No software is recommended or mentioned in this book. It is not a veiled attempt to get you to buy software, tools resources packages or anything else for that matter.

The format is startlingly simple; the book starts with the usual introduction on the format, how to use the book, etc. It then gives a matrix of project type against stage, e.g. business strategy, IT project, sales/marketing, design, etc. against defined objectives, monitor and review, implementation, etc. It then lists the rules that fall into each category. This will help guide you to the right group of tools or methods for the job.

There is also a matrix listing all the methods against various categories (analysis, creativity, problem solving, planning, etc.) to show where they can be used.

The methods themselves take two pages each. I wondered about this but I think it is a good idea. Had it been a flexible size then there would have been the temptation to go into details and get away from the bare essential ideal.

The format is the same for all the methods making comparison easy. There are sections; when to use, what you get (result), time (how long it takes), number of people (some are group/team activities), equipment required (usually pens, paper, flipchart and a room). These sections are a sentence or two at most.

You then get the method itself. Usually with a diagram, chart, table, etc. as required. This is more than two sentences but not more than half a page. It is the essential points. It is followed by a short example of a couple of sentences and then a suggested exercise, which is normally something very simple that will quickly show how to apply the method. This is followed by a short list of key points and then additional comments.

Following this is where to get additional information if you really do feel the need. Some companies want more detail and references before they implement new procedures. The final item is usually a chart, table or whatever, that is large enough to photocopy out of the book to start using as a blank for your own experiments or on the wall as appropriate.

The useful thing about this book is its brevity. You won't get bogged down in reading vast amounts of information. You get the essence of the method without getting bogged down in volumes of opinion and trivial detail. Most ideas are simple but you have to put in a lot of padding to get a book out of it. This book is 94 methods stripped to the essentials and laid bare.

This book will be indispensable to managers at all levels. When you have a problem, a project to do, etc. then ten minutes with this book will give you enough information to find a labour saving system that will help organise your thoughts. Highly recommended

Managing the Testing Process by Rex Black (0 471 22398 0), Wiley, 528pp @ £25-95 (1.54)
reviewed by Chris Hills

This is a brave title for a book. If there are any words that programmers run from they are 'managing', 'testing' and 'process'! Yet I would hope that all software engineers regard (properly organised) testing as an essential part of software development. Formal testing and proof of testing, is becoming more relevant as software is now in far more areas of our lives; areas where people sue when things go wrong.

This book is aimed at the manager and team leader explaining, as the title suggests, how to manage the testing processes to get the test team to do the work. It highlights some of the mistakes that can be made, some made by the author in previous projects where he had to dig himself out of a hole. This is a real world book not a set of academic lecture notes.

There are numerous diagrams, some, I am sure, will find their way into your own in house presentations to managers and engineers. As this is 'managing the process' it is orthogonal to most types of testing and does not look at specific tools or languages but stays generic. This is a plus point for me, as this book will survive changing fashions of languages and tools. That said it does look at the classical models, V and waterfall. Also the pros and cons of automated test tools with some interesting points.

For me the whole book is worth its price just for the pages that show how to calculate the ROI (Return on Investment) for testing. It shows, in financial terms, that the time and money spent on testing is well spent and that not testing (properly) is a false economy. It shows you how to beat the company accountant at his own game.

This book is as much anecdotal and team managing as methods and systems. The author has a lot of experience and enthusiasm for the subject, making it his life's main work. This can be seen from the author's web site. This book is clearly a culmination of many real projects and technical papers presented around the world. Most of the papers are on free download on the author's web site. You may ask why buy the book? For the same reason why people buy Red Hat Linux rather than downloading all the sources, all the information has been distilled into one volume that is easy to use. In fact I would say go to `www.Rexblackconsulting.com`, look at the papers and then buy the book from the author (it comes autographed). You will then have the slides to use when you need to do a presentation to convince the engineers or management.

I think that some parts are more applicable to the US reader than the European. However, most of it is generic and applicable to teams anywhere. I shall keep this book on the shelf as a reference and Rex Black's URL is on my favourites list.

Other

DNS and BIND Cookbook by Cricket Liu (1-596-00410-9), O'Reilly, 221pp @ £24-95 (1.40)
reviewed by David Ross

I'm fairly new to the Cookbook format, but have found it very useful since it's task orientated, which means that you can quickly tackle your problem (assuming it's covered by one of the recipes) rather than having to read a whole book cover to cover. Of course as a reviewer you're forced to read the book in a more traditional sense.

This book contains over 150 topics, which range from the rather simplistic 'registering a domain name' right up to 'setting up a stealth slave name server'. The book is biased towards the more complex topics however and I have to say that I was out of my depth in many of the topics since I'm really a BIND beginner. It is here that the cookbook format starts to fall down. There is never a real sense that you have the basics under your belt. The author seems to acknowledge this and points the reader to DNS and BIND, co-authored by Liu and also published by O'Reilly.

Overall, the book is written in an accessible and readable style. I'd have no trouble recommending it, but I think that you should first have read the companion book to get the basics before you consider tackling this one.

Linux in a Nutshell, Fourth Edition by Ellen Siever et al. (0-596-00482-6), O'Reilly, 928pp @ £28-50 (1.40)
reviewed by Giles Moran

This is the 4th edition of the popular desktop linux reference. The book covers most of the commands users will want to use for most tasks on a linux system. It contains several references in one, covering CVS, RCS, sed, gawk, vi, emacs, bash, tcsh, regular expressions, bootloaders, package management, desk top environments and some system administration commands. It's written in a fairly terse style, but gets the information over in a clear and concise manner. Examples are given, although more would have been useful. The index runs to 31 pages which gives some indication of the depth of reference material available.

After a quick introduction, the book dives into a system administration overview. This is a welcome chapter as I can now decipher most of the TLA that the local sysadmins throw at me with a moment's notice. The overview of networks and TCP/IP is clear and informative. Chapter three present the user, programmer and systems administration commands. This section is very large (464 pages) and seems very complete in its coverage. Chapter four covers boot methods covering both LILO and GRUB boot loaders. Package managers for RedHat and Debian linux distributions are described in chapter five. Chapters on shell, bash and tcsh, sed, gawk, emacs and vi follow. RCS is covered and then CVS. The chapter on CVS is very good and gives a concise overview of what CVS is and the commands. The book then shifts away from the terse command style to cover window managers. GNOME, KDE and fvwm2 are described in some detail.

This isn't a book to read (believe me I tried), but it is a good reference for those who regularly use linux and haven't already got a decent reference. I use CVS, tcsh, vi and emacs on a regular basis so this book is very useful. What this book won't do is to teach you how to get started with linux, other books in this series will do a much better job of that. Very handy and recommended.