Contents

Reports & Opinions

Reports

Dialogue

Features

Reviews

Copy Dates

C Vu 15.6: November 7th

C Vu 16.1: January 7th

Contact Information:

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

Advertising: Chris Lowe
Pete Goodliffe
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

The Internet Isn't Working

How often have you been approached by a friend, colleague or family member telling you that "the Internet isn't working"? If you're polite, you'll work out a way to find out what is actually wrong without having to tell them that there is no single centralized thing called the Internet, and that most of it certainly is working just fine, thank you very much. I hope you'll do me the same courtesy when I say that, right now, the Internet is not working.

Decentralized as it is, there is one thing holding the Internet together: the distributed database known as DNS (domain name service). While DNS information is distributed, it relies on a known set of "root" servers to provide authoritative information on top-level domains. Maintenance of these root servers is contracted out to Verisign, a private company whose interests do not always coincide with those of other Internet users.

One thing Verisign are responsible for is answering DNS queries for machines in the `.net` and `.com` domains. Fortunately DNS is defined by open standards, and Verisign are (or ought to be) bound by various contracts and rules. So far, so good.

But then Verisign decided to change the way DNS works on the Internet, presumably in a bid to make more money. In effect, Verisign have registered every single possibly unregistered domain in the `.com` and `.net` domains. Yes, every single one. Any lookup for the address ("A" record) for a domain name that is not registered to someone else will now return the address of `sitefinder.verisign.com`, where it should return a result indicating that the domain is not registered.

In case you're thinking that this is harmless, let's look at some consequences.

- Spam filters that rely on checking that the domain of a sender's address is valid no longer block any spam, as almost any domain appears to be valid
- Programs which tolerate expired domain names by falling back to another name will find that the expired name now resolves, but to a machine not offering the service.
- Web browsers pointed at a site that no longer exists receive HTTP return code 302, defined by the HTTP standards as meaning " The requested resource resides temporarily under a different URI." Verisign are breaking the http specification in RFC2616 as well as breaking DNS.
- Mail for some domains is now undeliverable. Mail servers use the DNS database to determine where to send mail. Previously a bad mail exchange record would have been mostly harmless – domains usually have more than one mail exchange machine set up, and mail servers would try another if the first could not be resolved. With Verisign's unilateral change, most such names will now be resolved. That would still be mostly harmless except that Verisign is accepting incoming mail connections on the SMTP port of `sitefinder.verisign.com`, whose address is returned. The SMTP server on that machine rejects all mail, but largely ignores its input, thoroughly ignoring the SMTP standard. (Try telnet to port 25 of `www.verisign-are-not-team-players.com`, and hit enter a few times.)
- Simple tools that used to test if domains were in use or running web servers no longer work.

There is more, but space and time are limited. A quick web search (or a visit to Slashdot, `http://slashdot.org/`) will get the latest details.

There is an online petition to register disapproval with Verisign's actions to ICANN, the authority responsible for the appointment of Verisign and others, at

`http://www.petitiononline.com/icanndns/petition.html`

At the time of writing about 13,000 people have signed. Don't be passive – make your voice heard. It might make a difference. Verisign's current position is to claim (against all manner of evidence) that their change has not had any significant ill effects. It's necessary to make a stand against this behaviour. If the directory services of the public Internet can be subverted for gain by a private company, there is little hope for keeping the Internet remotely honest.

I'm hoping that cybersquatting rules will be bought into play. Maybe a $0.01 fine for each domain that Verisign are inappropriately redirecting to their site would suffice...

It has long been a cliché to say that "the Internet interprets censorship as damage and routes around it". It's pleasing to see that the Internet community has some ability to interpret abuse of power similarly, and to reduce its effect. Within days of Verisign's attack on the Internet's infrastructure an update was announced for BIND, the most common DNS server on the Net, to filter out the inappropriate A records returned by Verisign's root nameservers. I am in two minds about whether it is better to lobby our ISPs to install patches for their nameservers to limit the damage Verisign can do or to meet the problem head on. Opinions?

[Late breaking update: ICANN has formally requested that Verisign restore the previous functionality of the DNS system while an inquiry is conducted. It's possible that Verisign might do so – but they have not yet.]

OS Wars

I have used a fair range of different operating systems in my time, from the unnamed OS powering my first 8-bit micro through Acorn's amusingly named "Arthur" (presumably a reference to the fact that it was Arthur, sorry, half a, operating system, a stopgap until RISC OS was released), a Unix from a company using the now largely disgraced name SCO, IBM's OS/2, more flavours of Windows than I choose to admit (even to myself), Mac OS and a host of others including RTOSes from embedded systems. My latest venture in the OS world has been to run a system using Apple's latest offering, Mac OS X. With Mac OS X, Apple have left behind their legacy and moved into the 20th, sorry, 21st century with pre-emptive multi-tasking, protected memory and many of the other things that other "grown up" OSs have been offering for many years. They continue to offer an OS heavily influenced by aesthetic factors, and intended to be more user-friendly that the competition. Most importantly for me, it's a Unix (or at least, Unix-like) system that integrates smoothly with my laptop hardware (because it's a one vendor show) and allows me to run Microsoft Word natively. So far, so good (again).

There were several reasons for me to give Mac OS X a try. Firstly, as with many programmers, I like to work with anything new and different. Secondly, it was guaranteed to go down well with my Mac-loving wife (who was not my wife when first I bought a machine with Mac OS X pre-installed). Those two are not necessarily in order of importance. Thirdly, with Mac OS X Apple appear to have moved on from previous battles with the Free Software and Open Source worlds a little, and now ship a software which is based on an Open Source core, Darwin. Indeed, Darwin is essentially Mac OS X without the pretty graphical user interface, and is freely available – which can come in handy, as when recently I wanted to see how to make system calls without using the supplied C library.

An aside: that was part of a hobby project, implementing the standard C++ library for myself – a fun exercise I'd recommend to anyone who wants to get more familiar with the library. I'd also recommend that for most purposes you use a tried and tested implementation rather than a home-grown one. `std::list`, for example, may be nice and easy to use, but in terms of implementation difficulty it's by far the most complicated linked list specification I've ever come across! Now where was I? Oh, yes: back to the OS talk.

My first real experience with Mac OS X was with version 10.1, and I had mixed feelings. As a Unix it's rather non-standard in terms of directory structure, startup operation, and many other aspects. As a system with a much hyped graphical shell, it felt rather incomplete – things that should have been simple weren't possible without dropping back to command line tools that were specific to Mac OS X. One small hurdle was the fact that Mac OS X does not, by default, permit root logins, and the option to allow them is well hidden. Using `sudo` is a good idea, but sometimes I just want to be root for a while.

The upgrade to Mac OS 10.2 filled in 90% of the holes I saw in 10.1. Stability issues still exist, and Apple's "It just works" slogan annoys me, because I find that less true of Mac OS X than of the other platforms on which I work and play: with Mac OS X simple things intermittently stop working, and then start working again. The `ping` utility, for example, will stop being able to resolve names even when `nslookup` can – and then minutes later `ping` will recover its senses. Come on Apple, the basics should "just

work". I do have to reboot several times a month, a level of instability I haven't seen since Windows NT 4. At least I got a free upgrade to 10.2 as I bought 10.1 only weeks before the launch of the later version.

But now Apple are set on losing whatever loyalty I might feel towards them. We are promised that the next release, 10.3, will cost US$129 whether you are upgrading from a 5-year old OS or from a 5-month old copy of Mac OS X. Should a company with a market share smaller than 5% of the desktop be so hostile to its existing users?

Maybe it's time for me to move on and try another exciting new OS – any suggestions?

James

From the Chair

Ewan Milne <chair@accu.org>

The past couple of months have seen a great deal of activity behind the scenes in preparation for next year's ACCU Conference. Last issue I mentioned our search for a new organiser: I'm happy to be able to welcome on board Julie Archer of Archer-Yates Associates. Julie will be handling the commercial side of things, and of course dealing with all your bookings, and I hope this will be the start of a fruitful relationship.

The programme is currently being finalised, and with a strong line-up of speakers both familiar and new to the conference already in place, I think I can predict that next year's will be as strong as ever. I prefer to wait until the ink is dry before mentioning names or details, but trust me, you will not be disappointed. There really is a lot of exciting stuff coming up.

The final element to be put in place are the conference dates and location. The conference will take place from the 14th to the 17th April. And the hot off the press news is that we will be making a return to central Oxford, to the prestigious Randolph Hotel. In this way we will combine central convenience, greater space for the keynote talks, and a location with a great deal of atmosphere. So put those dates in your diary now: this is the conference you can't afford to miss.

Membership Report

David Hodge <membership@accu.org>

At the time of writing in early September about 50% of the membership have renewed. Still to be entered, when the bank send us the statements, are the increasing number of members using the standing order facility. If you would like to do the same please email me. If you have not renewed you will have had an e-mail reminder in mid September and a letter in mid October. The October journal is the last one you will receive without renewing your membership. If you haven't already thrown away the plastic packaging, the address label will show the expiry date of your subscription.

Standards Report

Lois Goldthwaite <standards@accu.org>

Standards are good. Everybody knows that. Without standards, life would be a constant headache of coping with products that don't fit, that don't work together, that aren't reliable, that may not even be safe. I have lived in a place where broadcast television signals came in PAL, NTSC, or SECAM format, depending on which company had supplied the transmitter. It was also common there for houses to be wired with both 110 and 220 volts service, to accommodate appliances which might have come from anywhere in the world. (And it did not help that the builders had fitted the same kind of power points to both sets of wires! – here we see problems with having too many standards and not enough standards, simultaneously.)

It is widely recognised that ISO standards are the 'gold standard' of standards, especially in the information technology fields. Many standards which are developed elsewhere, whether by national bodies or industry groups, are subsequently submitted to ISO for confirmation as international standards, to gain the recognition which makes them widely useful.

But although the ISO imprimatur is highly respected, the ISO process unfortunately may not be. More and more, standards development activity is going to other standards bodies (such as ECMA), to industry groups (such as OASIS or W3C), and to ad hoc bodies which are chartered for the purpose (such as J-Consortium). Rightly or wrongly, the ISO committees and their member national bodies are perceived to be ponderous, bureaucratic, and exclusive. The current action is in areas like web services, e-commerce, XML, and not one of those standards is being developed in a committee under the ISO/IEC umbrella. In SC22, which is responsible for standards on programming languages and environments, the youngest working group is that for C++, founded in 1991.

Furthermore, SC22 is about to disband the Posix working group, WG15, and delegate responsibility for maintaining and extending that standard to the Austin Group. (The same Posix standard is jointly endorsed by the Open Group and IEEE as well as WG15; the cooperative effort is known as the Austin Group.) At the same time, SC22 is encouraging the Free Standards Group to submit their Linux Standard Base documents, when finalised, for ISO fast track approval. Of course Linux and Posix/Unix share a common heritage and purpose. There are some differences between them, but harmonising such differences is why standards committees exist.

The danger, and I am not the only person who sees it, is that ISO is increasingly dwindling into an organisation which wields a rubber stamp. The value of the ISO endorsement is that approval is based on the international consensus of technical experts. But it is impossible for national body technical experts to hold informed opinions if they are not able to participate in the deliberations which produce standards documents.

ISO needs to make some changes if it is to regain credibility as a developer of useful standards. First, it must find ways to work together with other bodies committed to open standards, jointly developing documents which all can adopt. This is the best way to build international consensus, and the Austin Group is a shining example of how well such a cooperative process can work.

Second, it must broaden participation in the ISO/IEC standards committees themselves. Currently only experts sponsored by a national body are eligible to attend official meetings. There should be channels for businesses, professional organisations, industry consortia, and individuals to take part and contribute their expertise. Plus unofficial channels too – the C++ committee already monitors Usenet newsgroups like `comp.std.c++` and `comp.lang.c++.moderated`, looking for possible defect reports and trends in using the language. The national standards bodies also need to broaden their appeal, especially the ones which now charge high fees for the privilege of volunteering to do unpaid hard work.

Third, ISO standards must become easier to find and more reasonable to acquire. Standards from many other sources are freely available, while ISO documents are expensive and difficult to get hold of. What good is a standard if hardly anyone gets to read it?

As a general trend, today's world is gravitating to embrace open standards and avoid proprietary ones. Open standards organisations must start seeing each other as collaborators, rather than rivals scrapping to 'own' the juiciest subjects ripe for standardisation. The real competition is with proprietary standards which seek to lock customers into expensive, restricted solutions. It would be a tragedy if the world's most prestigious open standards organisation committed suicide by hanging itself with red tape.

On the subject of ISO standards becoming more available, by the end of October, you should be able to walk into a good bookstore and buy printed, bound copies of the current C and C++ standards, published by John Wiley & Sons, Ltd. Each document incorporates the updated text of its respective Technical Corrigendum, so these are the most current documents available (unless you are a member of the C or C++ standard committees with access to working drafts – and if you want to become one, please write to `standards@accu.org`). The books hardly qualify as light entertainment, but if you care about writing portable programs, they are a handy reference to keep around. And a useful companion volume for all the other technical books on your shelf.

Dialogue

Student Code Critique Competition

Prizes provided by Blackwells Bookshops & Addison-Wesley

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

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

Student Code Critique 23

The problem

What would you do to help this student with his programming? Note that this time the student knows enough about C++ to write a suitable program, so the problem is in the mind to the extent that I think he is even confused by what is needed as output.

I am working on a program that will tell someone how old they are in years, days and months. I.e. You are 27 years, 200 months, and 1500 days old.

```
#include <iostream>
using namespace std;
void main() {
  int currentyear = 2003;
  int dob = 0;
  int result = 0;
  int day = 0;
  int month = 0;
  cout << "Enter your year of birth: ";
  cin >> dob;
  result = dob - 2003;
  cout << "You are " << result
       << " years old" << endl;
  cout << "Enther you month of birth: ";
  cin >> month;
  for(result=0; result <=12; result++)
    if(result <= 12) result = month+12;
  cout << "You are " << result
       << " months old"<<endl;
  cout << "Enter you day of birth: ";
  cin >> day;
  for(result=0; result <= 365; result++)
    if(result<=365) result = day+365;
  cout <<"You are " << result
       << " days old"<<endl;
}
```

Of course remember to comment on the program defects but your main focus should be how to explain the need for clear thought when producing a solution.

Introductory Comments

This time I had just two entries that took very different approaches. In my opinion neither managed to display clarity of thought in the context of code. Both provide ideas about clear thinking before you start coding. I wonder what you think?

I also wonder why you did not enter. Yes I do mean you, the person reading this now. Perhaps you thought about it but never actually got round to doing it. Perhaps you did it but thought it not worth sending in the result. Well that certainly makes my life easier and it also makes it easier for James to choose a winner. However that would be to miss the point: the real benefit from this column is not in reading it but in participating.

If you cannot sit down and bash out a reasonable submission to most of these code critique exercises in a couple of hours then you have a great deal still to learn. What is that I hear you say? OK so you are an expert, but how do you intend to give constructive help to the struggling newcomer to your team?

Please think about it and aim to try at least one critique every year. If you are competent you should not fear the criticism of your peers, and if you do fear such criticism you already doubt your own skills and should welcome the chance to learn a little more.

From Colin Greenock

Francis asked for a critique of a short programme derived from the following requirement, "I am working on a programme that will tell someone how old they are in years, days and months. I.e You are 27 years, 200 months, and 1500 days old."

There are two problems to be solved for this critique. The first is to try and explain the need for clear thought when trying to produce a solution. The second to identify defects in the code as supplied with an unwritten requirement for a sample of code that works,

Clear Thinking

So how on earth do we explain or demonstrate how to think clearly? Perhaps it is worth taking a step further back from the impossible and showing how a clear problem definition can make it easier to think clearly about the problem in hand. It doesn't remove the need for clear thought, but it provides a sieve that you can use to filter out the rubbish that might be cluttering your thoughts. The better the problem definition the finer the sieve and, one hopes, the clearer the resulting thoughts.

Let me try and demonstrate the process.

Taking the original summary of the programme's purpose it *looks* like the programme should report the user's age in three different formats rather than as a single, "You are 27 years, 6 months and 21 days old" result. Why? Well, normally you would not report month values greater than 11 or days greater than 364. If that is the case then the sample data doesn't make this clear as the figures yield 3 different ages, 27, not quite 17 and a little over 4 years. However looking at the code, well I'm really not sure what it's trying to do, but I think it may in fact be trying to report the age as a single, "You are x years, y months and z days old." statement.

So how are we to solve this problem? Which problem? Good point, let's step back a bit and see if we can identify what problems we have.
In no particular order:

- How are we to calculate the user's age?
- *What* calculations are we to perform? What are the "business" rules?
- How is the result to be presented to the user?
- Who is going to be using the proposed programme?
- Why is a programme required? Do we really need a computer to achieve a result?

Well the first problem on the list is probably the least important of them all, it will come down to (ha ha) a SMOP (simple matter of programming) and providing the rules used to calculate ages are followed correctly any one solution is as likely to be as good as any other.

The last question is always worth asking. Is this a one off task where the result can be obtained using a pencil and paper with, perhaps, the backing of an electronic calculator?

Correct answers to the remaining questions are vital to the successful outcome of our project. So how are we going to answer these questions? Simple. We're going to have to ask someone and more to the point someone who knows about the problem area.

The *really* difficult part of our trade is that there's only one way of finding answers to the questions above. That is talking to other people and extracting information and decisions from them. This is a skill every bit as important as being able to think clearly or knowing how to write the leanest, meanest most efficient code ever in the most fashionable language of the moment.

So how might this go in the commercial world? Well let's imagine....

"Good morning Sir. I'm from the I.T. dept. Have you a few minutes, perhaps half an hour to talk about your request for a programme to report someone's age?

What!? Can't you read? I sent a memorandum to your head chap detailing my requirements last week!
Ah, yes Sir. I have a copy here. I have read it, but there are still a few grey areas that we need to clear up with you and...
Oh, very well then. If you must, you must, but I'm a busy man so it had better not take too long.
Well, shall we start with how the programme is to find out how old the user is...
Isn't it obvious? He or she will enter their date of birth!
Ah, yes. We did rather expect that, but how will they enter their date of birth? Do you want them to be able to enter a date as they would on a paper form or is the user to enter each part separately?
Don't play silly beggars with me young woman, as you would on a form of course!
Ha hm. Just so. Ah, is this programme to be used in our overseas offices?
Now just what the blazes has that got to do with anything?
Well if we're going to ship it to all our overseas branches then the staff in each branch will probably want to use the calendar that they're used to..
What **are** you blethering on about?
Well the Islamic, Hebrew and Chinese calendars are in common use in some of our offices and I'm not certain but I think that one or two our branches in the Balkans are still using the Julian calendar..."

... and so it went on. Our heroine ploughed through a list of questions that she had, adding others that arose out of the discussion. Then, after a good deal of explanation that a programme written to a limited specification would be quicker to produce and above all would be far, far, **far** cheaper to produce than one that dealt gracefully with locale issues, she presented the following to ~~Colonel Blimp~~ her customer...

"A programme shall be written that will report the user's age as the number of complete years since the user's birth, the number of complete months since the user's birth and the number of complete days since the user's birth.
When the programme is run the user shall be prompted to enter first the year, then the month and finally the day of their birth.
The programme shall not attempt to calculate the age of the user until all of the date information is entered.

For example:

```
Please enter a year of birth  : 1904
Please enter a month of birth : 6
Please enter a day of birth   : 13
Age at end of 2003/09/02 was 99 years,
                          2 months, 21 days.
```

The programme shall apply the following validation rules to the entered data.

Years

From (current date – 130) to current date. Inclusive
For example in 2003 the range would be 1873 to 2003 inclusive.

Months

1 to 12. Inclusive

Days

1 to maximum number of days in month. Inclusive.

If a user enters a value outside a valid range he or she will be asked to enter a value that lies within the valid range.

For example:

```
Please enter your year of birth  : 1837
  Please enter a value between 1873 and 2003.
Please enter your year of birth  :
```

In addition:

- The Gregorian calendar shall be used.
- Users shall not be allowed to enter dates in advance of the current date."

All that work, for such a *simple* programme and we haven't even begun to suggest any solutions never mind cutting any code.

By the way there's a bear trap in the foregoing. The customer may not be, is rarely, the user and may just have led you right up the garden path. So in the ideal case we'd go and talk to some typical users as well before we went any further. That's the ideal though...

Well what we've got is not complete, first attempts almost never are, but it is a clearer statement of the problem that we can discuss with colleagues and it can be used as a basis for sensible design decisions and, just as important, we have the beginnings of something that we can test against.

So why would we want to discuss our problem definition further? Well, let's start with date entry. We want to prevent the entry of dates in advance of the current date. This might be simpler if the user entered the complete date at once, but then we have the complicating factor of format validation. Then there's the matter of valid year ranges. We've said "130 years before current date" and given an example. But do we really mean 'current date' or should we have said 'current year'? And what about that 130 limit itself? What happens if we want to know the age of an elderly Galapagos tortoise? Oh, and how do we define a 'complete' month and what about someone whose birthday falls on the 29th of Feb?

So, where are we in our attempt to show at least one way of reducing vulnerability to muddled thought?

- Define the problem. Unbounded problems will run away with you. How do you define the problem? Well...
- ...find out as much as you can about the problem area from those who know the problem area well.
- Try to consider the problem from the point of view of the person who posed it in the first place. Go through the problem definition slowly and ask yourself, "What if this circumstance were to change?" and "Why on earth was that value chosen?"
- Write your problem restatement/expansion down in simple, short words that anyone can understand. Get someone else – a colleague and especially the customer – to read it. If you can explain the problem simply to someone else and they understand what you tell them then there's a very good chance that you understand the problem well enough to actually solve it.
- One more time, *simple*, *short* words. Avoid jargon if you can.
- Never regard your first attempt at problem (re)definition as the last word in all but the most trivial cases. Of course you then have a supplementary difficulty, how trivial is trivial?

At this point you can then sit back under your (metaphorical) tree and try and feed candidate solutions through your sieve.

Sitting under the Tree

I haven't got much to say about this as, as I said earlier, how on earth do you teach clear thinking? So perhaps a list of points that work for me will at least give you something to start with…

- Don't try to solve the whole problem at once. If at all possible break the problem down into manageable pieces.
- Pick a section of the problem and work on it. Write things down. Nothing is likely to reduce clarity of thought faster than trying to maintain more information in your head than your mind can comfortably handle. Levels vary from person to person.
- When you are working on other sections of the problem be prepared to revise earlier work in the light of new discoveries. Revision might mean scrapping and starting again.
- Take breaks from the problem, clear your mind. Make a coffee or go for a short walk, do the washing up and if you are **really** stumped go and talk to your spouse/partner/cat. Anything just so long as it is completely unrelated to the work at hand.
- Allow yourself enough time to deal with the problem. Panic and its first-born muddled thinking are very often the products of a ridiculous deadline. If you do a good deal of commercial coding you will, more's the pity, find this out.

The Original Code

So what about the original code? Well as the original statement of requirements is a little vague there's not a lot that I can say and I have as many questions as criticisms.

- Surely it should be '`2003 - dob`'? If that's the case then it won't work for any year but 2003.
- Was '`currentyear`' meant to be retrieved from current date and the literal `2003` substituted for the sake of testing?

- There's no validation of input data.
- There is no explanation of the intention behind the arithmetic used to calculate month and day values and I cannot deduce the intent by reading the code.
- Why the choice of 'dob' for year of birth? Was the original intention to allow the user to enter a customary date such as 11/09/37?
- Was there an undocumented requirement to avoid using the standard library date and time functions?

Well having written all that the very least I can do is offer a possible solution to what might have been the original problem. It is flawed, but then you, dear reader, are the first reviewer of the work, the work was done without collaboration and (through no other fault but my own) I left it until too close to the deadline to start so poor design choices have been left as they were. Now, remind me, what was I saying earlier? :)

```
// Warning! Only (lightly) tested with g++ on
// FreeBSD 4.6.2 For student code critique 23.

#include <iostream>
#include <ctime>  // in g++ this is an alias
// for time.h - it may not always be so.
using namespace std;

// Assumptions -
//  Gregorian calendar is going to be used.
//  We don't expect to find anyone older than
//    about 125.
//  All sorts of assumptions about current
//    date, time and we had
//  to have a look at time.h to check what
      we'd get back.
const int MAX_LIFETIME_YEARS = 130;
const int SYS_BASE_YEAR  = 1900;
const int MONTHS_IN_YEAR = 12;
enum eMonth {
  Unknown,
  January = 1, February, March, April, May,
  June, July, August, September, October,
  November, December
};
bool IsLeapYear(int Year) {
// Leap years in the Gregorian calendar occur
// in years that are exactly divisible by 400.
// Non-centenary years that are exactly
// divisible by 4.
  if(Year % 100 == 0 && Year % 400 == 0)
    return true;
  if(Year % 100 != 0 && Year % 4 == 0)
    return true;
  return false;
}
int DaysInMonth(int Year, eMonth Month) {
/* Thirty days hath September, April, June &
    November, All the rest have thirty-one.
    Excepting February which hath 28 except...
*/
  int days = 31;
  switch (Month)  {
    case February:
      if(IsLeapYear(Year))
        days = 29;
      else
        days = 28;
      break;
    case September:
    case June:
    case April:
    case November:
      days = 30;
      break;
  }
  return days;
}

int main() {
  int Day   = 0;
  int Month = 0;
  int Year  = 0;

// We mustn't allow anyone to enter a date
// later than the current date.
  time_t NowInSeconds = 0 ;
  tm* pCurrentDate;
  int CurrentYear  = 0;
  int CurrentMonth = 0;
  int CurrentDay   = 0;

// Find out what the current date is.
  time(&NowInSeconds);
  pCurrentDate = localtime(
    const_cast<time_t* > (&NowInSeconds));
  CurrentYear = SYS_BASE_YEAR +
                    pCurrentDate->tm_year;
  int ApplicationBaseYear =
        CurrentYear - MAX_LIFETIME_YEARS;
  CurrentMonth = 1 + pCurrentDate->tm_mon;
  CurrentDay = pCurrentDate->tm_mday;

// Let's keep a track of where we are.
  cout << "Currently : " << CurrentYear << "/"
       << ((CurrentMonth < 10) ? "0" : "")
       << CurrentMonth << "/"
       << ((CurrentDay < 10) ? "0" : "")
       << CurrentDay << '\n';

  while(Year < ApplicationBaseYear
        || Year > CurrentYear) {
    cout << "Please enter a year of birth: ";
    cin >> Year;
    if(Year < ApplicationBaseYear
       || Year > CurrentYear)
      cout << "\tPlease enter a year from "
           << ApplicationBaseYear
           << " to " << CurrentYear << '\n';
  }
  int MaxMonth = MONTHS_IN_YEAR;
  if(Year == CurrentYear)
    MaxMonth = CurrentMonth;
  while(Month < 1 || Month > MaxMonth) {
    cout << "Please enter the month of
                                       birth: ";
    cin >> Month;
    if(Month < 1 || Month > MaxMonth) {
      cout << "\tA month from 1 to "
           << MaxMonth << " please.\n";
      Month = 0;
    }
  }
  int MaxDay = DaysInMonth(Year,
                  static_cast<eMonth>(Month));
  if(Year == CurrentYear
     && Month == CurrentMonth)
    MaxDay = CurrentDay;
  while(Day < 1 || Day > MaxDay) {
    cout << "Please enter a day of birth: ";
    cin >> Day;
    if(Day < 1 || Day > MaxDay) {
      cout << "\tA day between 1 and "
           << MaxDay << " please.\n";
      Day = 0;
    }
  }

  int WholeYears = CurrentYear
        - Year - (CurrentYear>Year? 1 : 0);
  int WholeMonths = 0;
  int WholeDays   = 0;
```

```
  if(CurrentMonth == Month) {
    if(CurrentDay >= Day) ++WholeYears;
    else WholeMonths = MONTHS_IN_YEAR-1;
  }
  else if(CurrentMonth > Month) {
    if(CurrentYear > Year) ++WholeYears;
    WholeMonths = CurrentMonth - Month- 1;
  }
  else {
    WholeMonths =
        MONTHS_IN_YEAR - Month + CurrentMonth;
    if(CurrentDay < Day) –WholeMonths;
  }
  if(CurrentDay >=Day) {
    WholeDays = CurrentDay -Day;
  }
  else {
    int PreviousMonth = (CurrentMonth +
          MONTHS_IN_YEAR -1) % MONTHS_IN_YEAR;
    WholeDays = DaysInMonth(CurrentYear,
    static_cast<eMonth>(PreviousMonth)) - Day;
    if(WholeDays < 1) WholeDays = 0;
      WholeDays = WholeDays + CurrentDay -1;
  }
  --CurrentDay;
  cout << "Age at end of "
       << CurrentYear << "/"
       << ((CurrentMonth < 10) ? "0" : "")
       << CurrentMonth << "/"
       << ((CurrentDay < 10) ? "0" : "")
       << CurrentDay << " was "
       << WholeYears << " year"
       << ((WholeYears != 1) ? "s " :" ")
       << WholeMonths << " month"
       << ((WholeMonths != 1) ? "s" :"")
       << " and "
       << WholeDays << " day"
       << ((WholeDays != 1) ? "s." :".\n");
   return 0;
}
```

Francis' Comments

Colin's code caused me some problems because there is a relatively large amount of it and in the original he had added a considerable amount of comment. Space is always at a premium in C Vu so I removed most of the comments. The result may be less readable but it is almost a page shorter, which, in this context, matters.

There are several points that I want to comment on before moving on to the next entry.

To my mind `main` is clearly far too large and includes far too many nested structures.

While Colin has carefully validated the input in regards to the numerical values he has not checked for successful input. In real life, in my experience, the latter problem is far more common.

A large proportion of his code (and most of the nested conditionals) is the result of the way he has approached the problem. How could he have done it differently? Consider:

```
int years = current_year birth_year;
int months = current_month – birth_month;
int days = current_day – birth_day;
if(days < 0) {
   --months;
   days += days_in[leap][current_month-1]
}
if(months < 0) {
   --years;
   months += months_in_year;
}
if(years < 0) {
   cout << "You haven't been born yet.\n";
else
   cout << "Your age is:"
// etc
```

Oh, that `days` is in because I would always use a look up table for this kind of irregular data. In this case I would store the number of days in December as both the first and the last entry. Check that out, it ensures that January works correctly even when the current date is earlier than a January birthday. The other advantage of a look-up table is that it makes the validation code for `birth_day` much simpler. `leap` will have been evaluated earlier.

One thing always worth considering is distinguishing between unlikely data and impossible data. The claim to have been born in 1503 is (extremely) unlikely but the claim to have been born in the 15th month or the 33rd day of the seventh month is impossible (for a Gregorian Calendar). I tend to favour accepting unlikely data but perhaps verifying it.

(Editorial note: I would not use a lookup table here as – to my mind – it adds complexity rather than clarity. But that just shows that simplicity and elegance is hard to pin down, and even experienced programmers can reasonably disagree sometimes. James)

from Walter Milner <w.w.milner@bham.ac.uk>

We will firstly check through the program as you have written it, then take a broader look.

The program as written

Firstly

```
void main()
```

might often be written as

```
int main(void)
```

together with a

```
return 0;
```

at the end of function `main`.

[*Not strong enough. It should be written with an* `int` *return type unless you are happy to write non-portable code relying on a vendor extension. Francis*]

Then you declare

```
int currentyear = 2003;
```

This works this year, but you would have to rewrite the program next year. Ideally one would get the program to use the system to tell you the current date. However you seem to have forgotten what you've done when you write

```
result = dob - 2003;
```

using the 'magic number' 2003 rather than

```
result = dob - currentyear;
```

However suppose `dob` is 1983, then `result` = 1983 - 2003 = -20. You should have said

```
result = currentyear - dob;
```

This raises the question of naming variables. I suppose `dob` is 'date of birth', but it is used in the program in the sense of 'year of birth'. Does the difference matter? Yes. There are a variety of naming conventions, but it is generally accepted that variable names should reflect as closely as possible the 'meaning' of the variable, and this often prompts you to ask yourself 'what *exactly* is this variable about?' which is often a very useful question. You can probably now see the problem with the name 'result'.

We then have

```
for(result=0; result <=12; result++)
  if(result <=12) result = month + 12;
cout << "You are " << result
     << "months old" << endl;
```

Let's see what this will do. The value of `month` will be between 1 and 12 (if the user entered a valid value). The first time into the loop, `result` will be 0, so

```
result = month + 12;
```

will give `result` a value between 13 and 24. Then `result++` will add 1 to this, and since `result` will be 14 or more, the loop will terminate. If I was born in January it will tell me I'm 14 months old.

So what did you think it would do? You have a ' + 12' inside a loop – so are you adding 12 for each year of the person's life? If so you would have needed to say

```
result += 12;
```

then added `month` in once after the loop for the incomplete year. But

- you loop through this 13 times, not for each year of life, and
- you use `result` both as the loop counter and to hold the required answer.

Using more precise variable names, such as `yearOfLife` or `monthOfLife`, might have made the problem slightly more apparent to you.

You repeat the same code pattern in

```
for(result=0; result<=365; result++)
  if(result<=365) result = day + 365;
cout << "You are " << result << " days old"
     << endl;
```

Before you repeat a code pattern – make sure the original works.

The broader view

We start from the question –

'What is this program supposed to do?"

You say it should 'tell the user how old they are in years, days and months i.e. 27 years 200 months and 1500 days old'

I think you mean "e.g." not "i.e." Now this might mean how old in *complete* years, *complete* months and days. Alternatively it might mean how old in years (ignoring part years) or in months (ignoring part months) or in days (ignoring... you've got it). Your example seems to suggest the latter, but not clearly, since someone who was exactly 27 years old would have be 324 months old, not 200. Before writing the program, you must be clear as to what it is supposed to do. Let's take the second interpretation here.

Einstein is supposed to have said

Keep it simple

which has been made into the acronymn KISS, keep it simple, stupid. However Einstein added the more useful caution

But not too simple.

In the context of calendars, dates and times, 'not too simple' is really quite complicated. For a start, what do you mean by month – calendar or lunar? Well calendar is a lot easier here, so we'll do that. As another example, calculating the number of complete years involves not just the year of birth and the current year. For example, the current date is 21 August 2003. If my date of birth was say 1 November 2002, my age in complete years is not 2003 - 2002 = 1. It is zero – I have not yet lived 12 months. Similar ideas apply for months and days. In this way we have a more precise specification for the program:

input: date of birth (day month and year) and current date (day month and year)

output: age in complete years, age in complete calendar months and age in complete days.

How do we do it? Well, you need to re-submit this assignment so I will not do it for you, just outline two approaches. The first is to use calendar and date library functions which will essentially do it for you. However that is not what you are expected to do here. The second approach is to do it yourself, as follows:

Firstly calculate the complete years. You would need to compare the current month and the month of birth, and if they are equal, the birth day compared to the current day. It might help to draw some diagrams, or work out some examples on paper.

Then calculate the complete months. This is the complete years times 12 plus the difference in the months.

Then the days. The simple approach is the complete years multiplied by 365, plus the days in the months – which depends on which months they are.

The correct approach takes leap years and leap centuries into account – and this is what makes libraries so nice.

Francis' Comments

Interesting that both submissions religiously use `endl`*. What is wrong with* '`\n`' *which does exactly what you want unless you really do need to flush the output. I actually edited most of Colin's uses of* `endl` *but left Walter's in so I could make this comment.*

The Winner of SCC 23

The editor's choice is: **Colin Greenock**

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

Student Code Critique 24

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

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

Francis' Scribbles

by Francis Glassborow

Spam

If you are reading this you are probably a programmer and that means that you are both a computer user and, almost certainly, a user of the Internet. Please read to the end of this column and think carefully about possible solutions. I will be offering some thoughts but only by way of getting the ball rolling.

Spam can be roughly grouped into three main categories:

- Sales material
- Virus/Trojan distribution
- Well-intentioned, if thoughtless, sharing

Well-intentioned

The last of these can be tackled by better education of casual users. We have to educate our friends, colleagues and relatives that forwarding email should be limited to those that have made it clear that they are happy to accept it. Just because it is easy to forward 'The Ten Worst Jokes' to all your friends does not make that a good idea. If you had to post those you would not do so.

I think it is our duty as, hopefully more enlightened, users of email to politely and kindly instruct those that are in the habit of forwarding that funny picture to all and sundry that this is not acceptable behaviour. We need to explain why that is the case, and the possibility that forwarding unsolicited email might contain a virus or Trojan is not the main issue though it is an important one.

Virus Propagation

The second category was, until recently, the major concern because though it made up a relatively small volume of email it had a potential for extreme damage. The activity is already illegal in most civilised countries but identification and prosecution of the perpetrators can be a problem.

We need to focus on better ways to reduce the threat, because the damage has been done long before the guilty have been arrested. To find solutions to this threat we need to look at the field of epidemiology. Reducing susceptibility via inoculation is part of the answer. Recognising that monocultures are particularly vulnerable is another. Slowing transmission rates and early identification are also part of the solution.

ISPs have a part to play though it will not be anywhere near a complete solution. One idea that is worth considering at all levels is the use of trap email addresses. By that I mean addresses that are present in address books, placed out on the Internet for 'harvesters' etc. but which are never intended for use. Any email addressed to such will act as an alert. It should not be beyond competent programmers to write software that can act reasonably on such an alert. Exactly what reasonably means depends on where the trap is triggered.

For example a trap triggered at ISP level might cause immediate suspension of the source address for a period of time (it need only be relatively short while checks can be made) and trigger a delay of all emails with a similar 'signature.'

More controversially, I think it is worth making it illegal to knowingly or through professional incompetence transmit a virus containing email. I am thinking primarily of ISPs here. Before anyone starts screaming about personal liberty, remember that transmission of hazardous material via other mechanisms such as a postal service is already subject to the law. Few people would object to the use of chemical sniffers in post offices, so why should we object to the electronic equivalent.

We should never have to worry about old viruses being delivered by email because our ISP should have detected them and quarantined them, advised us and left the decision as to subsequent action to the end recipient. We need laws assigning responsibility to persuade ISPs that they must act rather than simply leaving it to their customers.

You might wonder if we should simply encourage ISPs to add this as a paid for service. I think that is not enough, we need to stop virus propagation as early as possible and that is effectively at the first place where it can be detected.

This proposal would not stop a new virus which is why I think we need things like trap addresses and intelligent use to slow up propagation until the detection software can be updated.

Sales Spamming

A large amount of spam until recently (more in a moment) was concerned with trying to sell something. Very often the product was something that we would have preferred our children not to be exposed to even if the sale was legal.

I have been told that most such spam is the product of a tiny number of specialists who make money out of being able to propagate such material.

At first sight this activity would seem to be part of the normal commercial exploitation of technology. However I think we have to look deeper. Have you noticed how those canvassing for work (jobbing gardeners etc.) use very small fliers which they hand deliver and usually ask that they be returned? Even the marginal cost of such fliers makes their reuse worth the effort. The problem with spam promotion is that it costs whatever the spammer charges you. However the real costs are paid by the entire community.

Do you think the above is too strong a statement? Forget the cost of implementing filters: think what it has done to communication. Not so many years ago it was easy to contact someone because they would tell you how on their home page or in their signatures on posts to newsgroups. Now we are progressively isolating ourselves. The Internet is rapidly losing its early promise of a technology that would bring people together and becoming a place where suspicion of strangers is getting ever worse.

Now the latest round of sales oriented spam comes disguised as messages that report such things as delivery failure. In the past you would always check those because you would want to know which of your messages had failed to reach its intended destination. One or two false ones were bad but we could live with them but when a couple become dozens they get filtered out. So your spam filter rejects my innocent email because of something in the subject line and my spam filter rejects your system's rejection. We become isolated.

From my perspective this is a fundamental attack on our sense of community and needs to be taken as a serious issue. ISPs claim they cannot filter such email. OK so let us take that as true (though I have my doubts) and attack the cause of the problem by making it illegal to tout for trade via any form of unsolicited contact. Yes I know that some companies will consider that an attack on what they consider to be their basic rights but it is past time that we grasped the nettle. Perhaps instead of making it illegal we should require companies that try to drum up trade with unsolicited contact (by email, phone or any other automated mechanism) should pay a levy (tax or licence fee) based on the number of attempted contacts.

Of course this will not eliminate the problem, companies based on bad and/or illegal business practices will always be with us but it would restore the balance a little.

The next point of attack could be on the use of harvesters to construct databases of email addresses. Clearly an email address is personal information so I suspect that in Europe it is already illegal to create a database of email addresses without registering under your country's data protection act. If we could get the USA to implement similar legislation we would have another lever with which to control the commercial spammers.

Note that commercial spam is nearly always a matter of sending an identical message to a very large number of addresses. At the customer end that leaves another possible way to filter out some spam. Couple a widely visible trap address with suitable software and any duplicate messages that go both to your trap address and to one or more of your normal addresses could be filtered. How easy would it be to write such software? Perhaps you are someone who could do so.

A Phase Change

When I set out to write this column things were bad enough but between the draft and this copy something else happened. I do not know what but sometime last Thursday my email system moved from having an irritating amount of spam to a complete disaster. From 100-150 spams a day it moved to 2000 plus almost instantly. I.e. there was no ramp up to the new level. Most of these extra messages purport to originate from or be connected to Microsoft. Of course this is not the case. Microsoft are an innocent victim. However I have no recourse but to reject all email with 'Microsoft' or 'MS' somewhere in their headers unless they come from a specifically whitelisted source.

This is a reverse form of denial of service, I cannot contact Microsoft on any issue and get a reply unless I allow literally thousands of false emails through.

There is another issue with this level of spam, I have to access the headers in order to reject. On my home system with a broadband connection that is just a nuisance. However in a couple of weeks time I will be working on site and my only email access will be through webmail and experience tells me that I simply will not have the bandwidth to even do the filtering. That means that in all likelihood my mailbox will accumulate over ten thousand messages during my five days of absence.

Conclusion

We often hear claims about the right to freedom of speech, well it seems to me that what is currently happening is seriously prejudicing my freedom. It is time that all concerned do whatever they can to deal with this problem. It is already way out of hand.

If we care, which I think most of us do, we must attack the problem of unsolicited email from every direction available. We need to persuade our governments that it is a threat to continued commerce and an invasion of our privacy. We need to persuade our service providers to take the issue seriously and act not only to remove customers who abuse the system but also to stem the flood of incoming but unwanted mail. Filtering at the recipient's end is now too late; it needs to be done earlier.

We need to address the use of email (and cold phone calling, text messaging etc.) for 'advertising'. The decision to email a thousand people should cost, and the decision to email a million should cost much more. Holding a database of email addresses of people that have never contacted you should be illegal. I know that sounds draconian but a large part of the problem is the creation of, currently legal, databases of email addresses via harvesters.

We should not forget to educate our friends, colleagues and relatives so that they both understand the issues and avoid making things worse be forwarding unsolicited messages.

No one person can solve or even mitigate the problems we are facing but by acting constructively together we should be able to reduce it. Doing nothing is not an option.

By the way, if I reject an email from you, or you do not get a response resend with 'syzygy' somewhere in the subject header and it will be accepted, at least for now.

Another Matter for Concern

Academic libraries have long contributed to the society that funds them (at least in part) by making materials available to those who present themselves in person. For example I can go to The Oxford University Libraries (Bodlean, Radcliffe Science etc.) and as long as I have a reader's card I can research materials there even if I am not a member of the University.

For many years this has been the only reasonable way for ordinary citizens to get access to many academic and technical journals because the cost of subscribing is extremely high. However these costs have become very high for academic institutions as well with the result that many now only subscribe to blocks of journals in electronic form. The problem is that the suppliers often prohibit access to those who are not members of the Institution in question, even if the access is through on line facilities in the Library reading room. So now we fund, through our taxes, academic journals and yet can no longer read them. I do not think this is good enough, do you?

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 ???
}
```

There are two possible situations where input may fail. In the first the kind of data being read may not meet the requirements for the value being sought. In this case there is little that we can do other than throw an exception.

The second case of failure is one that is not unexpected, we have reached the end of the input stream (e.g. we have already read up to the end of the file). It seems to me that we should not handle this instance by throwing an exception. What should we do?

Commentary on Problem 10

```
int foo(bool read_all) {
  if(read_all) {
    string line;
    getline(cin, line);
    return atoi(line);
  }
  else {
    int i;
    cin >> i;
    return i;
  }
}
```

There are quite a few things wrong with the above function definition. Assume that all the appropriate headers have been included; what particular feature of C++ input makes it completely unusable?

The fundamental problem is that after running such a function the programmer cannot know the state of the input. If `read_all` is true it will have read the whole of a line from the standard input stream. If `read_all` is false it will leave at the very least a carriage return in the input stream.

Now there is no standard conforming way that you can restore the input stream to a state where you can use `getline()` in a predictable fashion. If you do not believe me, try it. There are all kinds of things that seem to offer some hope but you will find that each one falls by the wayside.

After months of trying to solve this problem for readers of my book (due out in the first week of December) I eventually gave up and provided functions that sidestep the problem. I provided a `getdata()` function that reads to the end of the first input line in which there is at least one character that is not whitespace.

I also made my `read<>` templates handle the problem by extracting terminating space up to and including a carriage return. They stop early if a non-whitespace character is encountered after the successful acquisition of the required value.

Should we be requiring inexperienced and incidental programmers to jump through such loops to handle input adequately?

Cryptic Clues for Prizes

Last time I set you the following little problem:

> If I gave you 'That foolish day' as a clue you might quite reasonably think of 1st April. But what might 'An English programmer gets pieces of eight for the day of fools.' give as a two digit number? As an added clue an American would get a three-digit answer.

Was it too difficult? Or did you all assume that it was so easy that I would be inundated with answers? I had thought that as my readers are programmers they would find it fairly easy. The first of April becomes 0104 (well my side of the Atlantic, though the other side it is 0401). The mention of programmer and pieces of eight should have re-enforced using octal. That results in 68. I had to eliminate the US possibility because that would give 257.

Note that it is an interesting characteristic of clues based on dates that they often leave an option to interpret the number as an octal one.

Time and date clues are often only valid in the context of when they are written. For example 'When next we leap.' might be part of a clue with an answer of 2004 if written this year but it will not be that after 29/02/2004.

You may know that the ancient Greek method for representing numbers was based on using their alphabet (actually they threw in a few extra symbols). The basic idea was that the first 10 letters were treated as the values one to ten, the next nine covered twenty to one hundred and the final nine covered two hundred to one thousand. For many purposes being able to represent all values from one to a thousand was quite sufficient. They added extra features to deal with bigger numbers when they came to need them. Note that this method does not require some specific order for the letters. If we use the English alphabet we have to stop at 800 because we have not got enough symbols unless we add a couple of extras. I choose not to do so in order to keep things simple.

We have A-J representing 1 to 10, J-S for 10 to 100, S-Z for 100 to 800. Your task this time, should you accept the challenge, is to produce a Greek style clue for 261. You have plenty of scope for creativity because the letters required spell three English words, and the creative might find they could use one of the other orders by finding a suitable TLA (three letter acronym). And that is before you play with headless, leaderless, gutted and tailless versions.

Try warming up with this one because my Christmas competition will be for a prize with a little more value than those I have been offering so far. The prize for this time (apart from fame) will depend on which computer language or problem domain interests you.

Francis

Features

Professionalism in Programming #22

Finding fault

Pete Goodliffe <pete@cthree.org>

Nobody's perfect. Well, except for me that is. All day I have to sit down and work through tedious problems in other people's code. The test department discovers that our software falls over when they do *such-and-such*. So I trawl through the system to find what Programmer Fred did wrong three years ago, patch it up and send it back to test for them to break again.

Of course, you wouldn't find *me* making those sorts of elementary mistakes, not a chance. My code is watertight. Faultless. Low fat and cholesterol free. I don't write a line until I've gone over everything in my head, I don't complete a code statement without considering all the special cases that might occur, and I type so carefully that I've never once misplaced = for == in an `if` statement.

Totally fault free, me. Really.

Well, perhaps not quite.

The facts of life

I don't think anyone sits trainee programmers down and explains the facts of life to them. *It's like this, son. There are the birds and the bees. Oh, and the bugs.* Bugs are the inevitable dark side of constructing software, a simple fact of life. Sad, but true. Whole departments, and even industries, exist to manage them.

Everyone reading this will be only too aware of the proliferation of faults that exist in released software. How do bugs appear with such frightening regularity and in such great magnitude? It's all down to human nature. Programs are written by humans. Humans make mistakes. They make mistakes for a number of reasons (or excuses). They make mistakes because they don't understand the system they're working on well enough, because they don't correctly understand what they are implementing, but more often than not because they just don't pay enough attention to what they're doing. Most bugs are due to mindlessness. I once saw a wonderfully simple illustration of this, play along at home:

The tree that grows from an acorn is called an _____
The noise a frog makes is a _____
The vapour that rises from fire is called _____
The white of an egg is called the _____

The yolk, right? Think about it. If you didn't fall for that one, then you were probably only paying attention because I'd just warned you. Hey, give yourself a brownie point anyway. But tell me who warns you every time that you're about to write a potentially flawed line of code? They'd deserve a lifetime supply of brownie points.

So as programmers we're all to blame for the bad state of software. We're all guilty. Do we learn to live with the guilt, or do we do something about it? There are two types of response. The first school is the *it's not a fault, it's a feature* school. A fault turns up and we respond in the words of the great philosopher Bart Simpson: *I didn't do it. Nobody saw me do it. You can't prove anything* [1]. We blame compiler quirks, OS flaws, random climate changes, or computers with a mind of their own. Or as I alluded to in the opening paragraphs, we blame other people. A Teflon raincoat can be a very handy programming tool.

However, we should really subscribe to the second school, the school that concedes that software errors are *not* entirely inevitable. Many of these kinds of mindless mistake can be picked up or even prevented, and as responsible programmers we should be taking steps to do so. In this article we'll find out about of this, and look at some good debugging techniques to employ when bugs do slip through the net.

Nature of the beast

Contrary to popular belief the term *bug* was in use before the advent of computers. In the 1870s Thomas Edison talked about bugs in electrical circuits. The story of the Harvard University Mark II Aiken Relay Calculator tells of the first recorded computer bug. In 1947, the early days of computers when they took up whole rooms, a moth flew in and managed to lodge itself in some circuits, causing a system failure. They taped it into the logbook and wrote below: *First actual case of bug being found*. For posterity's sake it has been preserved in the Smithsonian Institute.

Bugs are bad news. But what are they really? It's worth identifying the different varieties of bug we encounter, understand how they are born, survive and can be exterminated. It's also important to know what to call them; see the sidebar for more on this matter.

Nomen nudum: what *shall* we call them?

The term 'bug' is remarkably evocative, and incredibly imprecise. It's very easy to throw around words without really understanding what they mean. If we use more specific terminology then we'll get straight in our head some key facts.

The exact meaning of the three terms below depends on who's defining them; this can get a bit philosophical. These interpretations are largely inspired by IEEE literature [2].

- **Error**
  An error is something that *we* do wrong. It is a specific human action that results in software containing a fault. Whilst merrily coding away, for example, forgetting to check a condition (like the size of an array before indexing into it) is an error.
- **Fault**
  A fault is the consequence of an error, embodied in the software. I made an error, and this resulted in a fault in the code. Now at first this is a *latent* problem. If the code I've just written is never executed then this fault will never have a chance to cause problems. If execution often passes through the faulty code, but never in the particular way that triggers the fault, we'll never notice that there is a fault at all. This subtle little point is what makes debugging so notoriously difficult. A faulty line of code may appear to work flawlessly for years, and then one day it causes the most bizarre system tantrum you've ever seen; you'll not suspect the aged code since it's been so reliable for so long.
- **Failure**
  So a fault, if encountered, may cause a failure. It may not. The failure is what we really care about, the manifestation of the fault, and it's the only thing we'll probably take notice of[1]. A failure is the departure of your program's operation from its requirements, from its expected behaviour. This is where we are verging on philosophy. If a tree falls over in a forest does it make a sound; if the running program doesn't exercise a bug, is the mistake still a fault? These definitions help to answer this.

If you want a hard definition of *bug* then it is a synonym for fault. The problem with the word "bug" is that users throw it around without knowing exactly what they're describing; this dilutes any true meaning. When being precise it's best ignored. There are other related words that can be thrown into the lexicon for good measure: *defect*, for example. Again, they'll mean different things if you ask different people, and we can happily survive here without getting too anal about them.

In most situations these (perhaps arbitrary) distinctions don't really matter, you can happily talk about a fault, an error, or a bug and not worry about being pedantically misinterpreted. However, in an article about bugs it's good to be clear what we're talking about.

1 This isn't necessarily the way it should be. Code inspections, when done, should pick up on a lot of faults that have never had a chance to manifest themselves as failures.

Software bugs fall into a few broad categories, and understanding these will help us to reason about them. Some bugs are naturally harder to find than others, and this usually turns out to be related to their category. Stepping right back and squinting at them from a distance, we see these three classes emerge:

- **Failure to compile**
  It's really annoying when the code you've spent ages writing fails to compile. It means that you'll have to go and fix a tedious little typo or some parameter type mismatch, then wait for the compiler to run again before you can get to the real job of testing your handiwork. It may come as a surprise to learn that this is the best type of error you can get. Why? Simply because it's the easiest to detect and fix[2]. It's the most immediate, and the most obvious.
  Faults cost more to fix the longer it takes to detect them. We saw in the previous article that the cost of changing software rises dramatically over the life of a project, and this holds for fixing faults. The sooner we catch them and fix them, the sooner we can move on, the less fuss and cost they incur. Compilation failures are easy to notice, and usually very easy to fix. You can't run the code until you have.
  Most of the time a compilation failure will be a silly syntactic mistake, or something simple like calling a function with the wrong number or type of parameters. The failure might be due to a fault in a makefile, it might be a link stage error (say, a missing function implementation), or even a build server running out of disk space.

- **Runtime crash**
  After enough donkeying about fixing your compilation errors, out pops your executable and you merrily run it. Then it crashes. You probably swear and mutter something about random cosmic rays. After the sixtieth crash you're threatening to throw your computer out of the window. These kinds of error are far harder to deal with than compilation errors, but they're still reasonable to work with.
  This is because, like compilation errors, they are blindingly obvious. You can't argue with an ex-program. You can't pretend a crash is a feature. When it has kicked the bucket and shuffled off its mortal coil, you step back and begin to figure out where your program went wrong. You'll have some clues (what input sequence preceded the crash, what had happened previously), and can employ tools to discover more information (more on this later).

- **Unexpected behaviour**
  Now this is the really nasty one, when your program isn't pushing up the daisies, just pining for the fjords. Suddenly it does the wrong thing. You expected a blue square and out popped a yellow triangle. The code continues to meander on its happy way with total disregard for your frustration. What caused the yellow triangle to appear? Has the program been overthrown by a militant army of guerrilla COM objects? It will almost certainly be a minute logic problem in the bowels of the code that executed over half and hour ago. Good luck finding it...

A failure may manifest itself because of a defective single line of code, or may only show up when several interconnecting modules are finally glued together, their assumptions not quite matching up.

Moving in a bit, and looking more closely at runtime errors, a few more groupings of fault become clear. Here they are ranked in order of pain, from splinter to decapitation.

- **Syntactical errors**
  Whilst these *are* mostly caught by the compiler at build-time, sometimes language grammar errors slip through undetected. They can generate weird and unexpected behaviour. The syntax error will often be one of; mistaking `==` for `=`, or `&&` for `&` in a conditional expression, forgetting a semicolon or adding one in the wrong place (the classic is after a `for` statement), forgetting to enclose a set of loop statements in braces, or mismatching parentheses. The simplest way to avoid being tripped up by these sorts of error is to keep all warnings switched on; compilers tend to moan about of lot of these potential problems.

2 Provided you have a sane build environment that stops when it encounters an error and provides some reasonable diagnostic messages.

- **Build errors**
  Whilst not necessarily a runtime fault *per se*, the build error manifests itself at run time. Be on the lookout and always distrust your build system, no matter how good you think it is. In these enlightened times you're unlikely to come across a compiler bug. However, you may not always be running what you thought you built. Several times I've been hit by this: the build system failed to create a program or shared library, perhaps because makefiles didn't contain adequate dependency information, or the old executable had a bad timestamp. Every time I tested a modification I was still running the old buggy code unawares. There are a number of ways to confuse a build system, but the worst part is you don't notice it failing – like a leprous limb.
  It can take quite some time (and maybe even a brief stint in the funny farm) to notice that this is biting you. For this reason, when you feel at all wary of what's going on it can be sensible to do a total clean out of your project, and then rebuild from scratch. This should flush out any possible build system problems[3].

- **Basic semantic bugs**
  The majority of runtime faults are due to very simple errors causing incorrect behaviour. Using uninitialised variables is a classic example, and can be quite hard to track since the program's behaviour may depend on the garbage value that was previously in the memory location used by the variable. One time the program will work fine, another time it may fail. Other basic semantic faults are: comparing floats for equality, writing calculations that don't handle numerical overflow, and rounding errors from implicit type conversions (losing the sign of a `char` is common). This type of semantic fault is often caught with static analysis tools.

- **Semantic bugs**
  These are much harder to identify, the insidious errors that won't be caught by inspection tools. A semantic bug might be a low-level error like the wrong variable being used in the wrong place, not validating a function's input parameters, or getting a loop wrong. It may be a higher-level piece of wrong-headedness, calling an API incorrectly, or not keeping an object's state internally consistent. A pile of memory related errors fall in this category – they can be evil to find due to their ability to warp and corrupt your running code, so that it behaves in totally unpredictable and unreasonable ways. Programs often behave weirdly. The only consolation is that they're doing exactly what we told them to.

The best kind of runtime failures are the reliable ones. If they're reproducible, they are much easier to write tests for, and track down the cause of. The failures that don't always occur tend to be memory corruptions.

Now that we have things in neat little boxes, let's zoom right in and take a look at some of the specific types of runtime failure. These are some common semantic faults that we come across.

- **Segmentation faults (or protection faults)** come from accessing memory locations that have not been allocated for the program's use. They result in the operating system aborting the application code and producing some form of error message, usually with diagnostic information. This can be triggered by dodgy pointer arithmetic, or far too easily by typing errors involving pointers. A common C typo causing a segfault is `scanf("%d", number);` The missing `&` before `number` makes `scanf` try to write into the memory location referenced by the (garbage) contents of `number`, and *poof!* the program disappears in blue smoke. If you're really unlucky, though, `number` happens to hold a value that equates to a valid memory address. Now your code will continue as if nothing was wrong, until the memory you just wrote over is used and your fate is in the lap of the gods.

- **Memory overruns** are caused by writing past memory that has been allocated for your data structure, be it an array, a vector, or some other

3 This presumes that you trust your 'build clean' facility. To be really thorough you can delete the project and check it back out again afresh. Alternatively, manually remove all intermediate object files, libraries and executables. For large projects both of these options can be tedious in the extreme. *C'est la vie.*

custom construct. When writing values into the wide blue yonder, you'll generally end up clobbering data from some other part of your program. If you're running on an unprotected operating system (more common in embedded environments) you may even tamper with data from another process or the OS itself. Ouch. Memory overrun is a common problem and difficult to detect, usually the symptom is random unexpected behaviour manifesting at a much later point than the overrun, many thousands of instructions later. If you're lucky the memory overrun hits an invalid memory address and you get a segfault which is hard not to notice. Use 'safe' data structures wherever possible to insulate yourself from the possibility of such disaster.

- **Memory leaks** are a constant threat in non-garbage collected languages[4]. When you want some memory you have to ask the runtime for it nicely (using `malloc` in C or `new` in C++), and then you have to be polite and give it back when you're done (using `free` and `delete` respectively). If you rudely forget to release memory, your program slowly consumes more and more of the computer's scarce resources. You may not notice it at first, but gradually your computer's response will degrade, as memory pages thrash to and from the disk. Two other classes of error relate to this: freeing a memory block *too many* times causing unpredictable environmental failures, and not managing other scarce resources carefully, like file handles and network connections.

- **Running out of memory** is always a possibility, as is running out of file handles or any other managed resource. It might be rare (modern computers have so much memory, how could this possibly happen?) but that's no excuse to ignore the potential for failure. Only sloppy code fails to make appropriate checks and will consequently perform in a very brittle manner when run in constrained situations. Always validate the return status of a memory allocation or file open system call. It is worth noting that some modern operating systems[5] will never return a failure from a memory allocation call – every allocation returns a pointer to a reserved but unallocated memory page. When the program eventually tries to access this page, an OS mechanism traps the access and then really allocates memory to the page, resuming normal program operation. This all works nicely until the available memory finally is exhausted. Your program will then be sent error signals, a long time after the relevant allocation occurred.

- **Maths errors** (*or "Math" errors for those using strange variants of the English language – Ed*) come in a number of guises: floating point exceptions, incorrect mathematical constructions or incorrect use of floating point numbers (for example, divide by zero). Even trying to output a `float` but passing an `int` through `printf("%f")` can cause your program to bomb with a maths error.

- **Program hangs** are usually caused by bad program logic. Infinite loops with badly crafted terminal cases are the most common, we also see deadlock or race conditions in threaded code, and in event-driven code the waiting on events that will never occur. It is usually fairly easy to interrupt the running program, see where the code has stalled and determine the cause of the hang.

Different OSes, languages, and environments report these errors in different ways, with different wording. Some languages try to avoid types of error by not providing features you can shoot yourself in the foot with. Java, for example, has no pointers and checks every memory access you make automatically.

Pest extermination

Like a hypochondriac, our code is constantly complaining about being ill. More often than not it genuinely is in need of some attention. We're the doctors. If our code is sick then we've got to perform the diagnosis, the surgery, and nurture it through its convalescence.

Weeding out bugs is hard. Not only do humans make mistakes when writing, they also make mistakes when reading. When I proof read these articles I have a tendency to read what I *meant* to write and not what I *really* wrote; it works the same for software. When we look at our faulty code we'll tend to see what we intended, not how the compiler actually interprets our instructions. In this respect the compiler is really quite pedantic, it can only produce exactly what we asked, not what we were hoping for.

Some programmers introduce far fewer faults into their code than their peers (as much as 60% less), can find and fix faults quicker (in as little as 35% of the time), and introduce fewer faults as they do so (figures from [3]). How do they do it? They are naturally able to pay more attention to the task, and can focus on the microscopic level of the code they're writing whilst keeping the broader picture in mind.

The professional programmer is always mindful of introducing faults, and will try to fix a detected problem sooner rather than later. Certainly, it's wrong to presume that we only check for problems when the software has been written. I've known many programmers who believe that the test department exists to detect their bugs for them. This is just plain wrong.

There is a clear difference between testing and debugging. Testing identifies the presence of a fault, e.g. the program output is incorrect, whereas debugging is the process of reproducing, locating, understanding, and fixing a fault.

Testing is QA, that is quality assurance; debugging is repairing a problem. You don't get quality by fixing bugs, you can't add it in at the end of software development, you must plan the quality into the architecture and implementation. Testing won't prove the absence of faults, it won't catch all errors. It's impossible to draft exhaustive test cases; software is just too complex. We will inevitably release software into the field containing faults that may still crop up. Yes, the quality of our software is in part down to the quality of our testing department, but also to our personal testing, and the quality of the fixes that we implement.

Debugging techniques and tools

There is an art to debugging, and it's very much something to be learnt. It's a skill. Experience shows you how to become an effective debugger. And this is something that we *will* all get plenty of experience at. Now, different people's brains work in very different ways, and they have different ways of problem solving. What works for one programmer may not for another. However, there are some general principles that always apply.

The sidebar (next page) offers a whistle-stop tour of the tools available to aid our bug hunting. How we use these tools and where and when they are applicable will differ from situation to situation. However, one of the most potent weapons in our debugging arsenal is a distrust of anyone's code mixed with a healthy dose of cynicism. The cause of your errant behaviour could be absolutely anything, and in the act of diagnosis we should start by eliminating even the most unlikely of candidates.

How difficult it is to find a fault depends on how well you know the code it's lurking in. It's hard to jump into some random source and make any kind of judgement about it without knowing the structure and how it's intended to work. For this reason, if you have to debug some new code take time to learn it first, it really will pay off in the long run.

The ease of debugging is also dependent on the control you have over the execution environment, how much you can play around with the running program and inspect its state. In an embedded environment debugging can be much harder because the tool support is sparser. You're also probably running in an environment that is providing a lot less insulation from your own stupidity; little mistakes can have much bigger consequences.

There are two distinct facets to debugging: *finding* the fault and *fixing* the fault. The following sections describe a sensible approach to both.

The golden rule when debugging is this: *Use Your Brain*. Think. Consider what you're doing. Don't flail around thoughtlessly hacking at bits of code until something begins to look like it might be working. Now, sometimes a quick fiddle about *will* get you results, sometimes some hacky little exploratory tests will pinpoint the problem quickly. So is it a justifiable thing to do? Perhaps, but if you make the conscious decision to do some quick-and-dirty stabbing around, set yourself a hard time limit to do it in. It's all too easy to spend an entire morning with the 'just one more little

4 OK, it is possible to leak memory in a garbage collected language. Hand two objects references to one another and then let go of both of them. Unless you have a very advanced garbage collector they will never be swept up.

5 This is certainly the case for Linux, at least until you exhaust the virtual memory address space. At this point `malloc` may return 0, but the system would probably have keeled over before you got a chance to notice. I'm not sure how Windows works in this respect.

go' approach. After the time limit is up, follow the more methodical approach laid out below.

If your quick stab turns up trumps and you do find the fault, re-engage your thinking gear. Look at the **How to fix faults** section below, make the change carefully and thoughtfully. Just because the fault was easy to find, it doesn't necessarily mean that the fix is quite as obvious as it looks.

Bug hunting

So how do we find bugs? If there was a simple three-step process we'd all have learnt it and our programs would be perfect by now. As it is, there isn't and they aren't. Let's try to distil the available bug hunting wisdom.

Compile time errors. We'll look at these first, since they are comparatively easy to deal with. When your compiler comes across

Wasp spray, slug repellant, fly paper...

Debugging would be a lot nicer if there was someone else to do the job for us. Whilst that'll never happen, we can make the job a lot more palatable with a little help. Many useful tools exist; you'd be stupid not to take advantage of them. A little time learning how they work may reduce your debugging time immeasurably.

Some tools are *interactive*, allowing you to inspect the code in various ways whilst a program is actually running. In advanced development environments these tools may be seamlessly integrated, or they may need to be run as separate programs. Other tools are *non-interactive*, often running as a code filter or parser spitting out information about the code following analysis. In this list we'll also consider tools you may not have thought of as debugging aids, and even some helpful procedures.

- **Debugger.** This is perhaps the most well known debugging tool, its name kind of gives its purpose away. A debugger is an interactive tool that allows you to view the internals of your running program and poke around with it. You can follow the flow of control, inspect the contents of variables, set *breakpoints* in the code for later interruption, even run arbitrary sections of code at will. Debuggers come in many shapes and sizes, some command line tools, some graphical applications. Usually there will be at least one available for your particular development platform (although the ubiquitous `gdb` seems to get ported to every conceivable platform these days). A debugger relies on *symbols* being left in your executable (these are the compiler's debugging information which are normally stripped out at the final link stage) – it uses these to provide you with information about function and variable names, and the location of the source files. A debugger is a rich and powerful tool, however I believe that they can often be misused or overused, and can actually inhibit good debugging. Programmers easily get wrapped up chasing what the program is doing, getting side tracked by observing the wrong variable values, stepping into the wrong functions, and don't sit back and *think* about the problem they are trying to solve. A little more thought about a failure may pinpoint the specific fault far quicker than trying to hunt it down in a debugger.

- **Memory access validator.** This interactive tool inspects your running program for memory leaks and overruns. It can be remarkably useful, showing up reams of memory release failures you never knew existed.

- **System call trace utilities**, like Linux's `strace` show all the system calls issued by an application. This can be a good way to see how a program is interacting with its environment, particularly useful when it appears to be stalled on some external activity that is not happening.

- **Core dump.** This is a Unix term for the OS-generated snapshot of a program that can be produced when it exits abnormally. The term derives from archaic machines with ferrite core memory, however the dump file is still called *core*. It contains a copy of the program's memory when it died, the state of the CPU registers, and the function call stack. The core dump can be loaded into an analyser (which is most often the debugger) to query a number of useful bits of information.

- **Logging facilities** allow you to programmatically generate information about your application as it runs. Rich logging systems allow you to assign priorities to the output (e.g. debug, warning, fatal), and then filter out a particular message level at run time. The program's log gives a history of activity that can help pinpoint what circumstances triggered a failure. The logging facility may be an integral part of the operating environment, or provided by a third party library. Without such support you'll see the use of `printf`/`cerr` diagnostic information, introduced on a very ad hoc basis. This is about as basic as you can get, and must be carefully removed in the production code release. `printf`s may also clobber the normal program output. I have worked in environments where even lowly `printf`s weren't available; when bringing up a system board the only diagnostic output I had was a single eight segment LED display, and a scope attached to a spare system bus! There are downsides to logging: it can slow down program execution and bloat the executable size if the logging statements can't be compiled out completely. Some logging systems are useless for trapping a program crash, since at the crash time messages may still be stuck in an output buffer that will never get flushed. Be sure you know how well your logging mechanism works, and always send diagnostic `printf`s to the unbuffered `stderr`, not `stdout`.

- **Static analyser.** This is a type of non-interactive tool that inspects source code for potential problem areas. Many compilers include support for this kind of functionality when set to their maximum warning level, but good static analysis tools go far beyond this. Products exist to discover problem code, any usage of undefined behaviour or non-portable constructs, to identify dangerous programming practices, to provide code metrics, to enforce coding standards, and to create test harnesses. Use of a static analysis tool can eradicate many errors before they have a chance to bite. A handy safety net. It's a sound pragmatic idea to use a static analyser from a different company than your compiler manufacturer – they're less likely to have made the same set of mistakes.

- **Code reviews** often identify problem areas that would otherwise go undetected. They were described in an earlier article [4]. If you've never done one, you'll be surprised how many faults can get unearthed this way.

- **Defensive programming techniques** [5] greatly reduce the likelihood of all sorts of errors. In particular, the use of assertions to check logical invariant conditions can be crucial. Whilst tracking a bug you can insert more assertions to validate the assumptions you've made about the code.

- **Fault logging/reporting database systems** such as `Bugzilla` provide persistent records of all failures so no problem, no matter how small, is ever forgotten. It helps you gather statistics on the quality of the project, so you know when it has reached a releasable state. It is a key tool, integral to the development process. It won't find faults for you, but helps co-ordinate the process of doing so. It allows you to assign problems to engineers, to mark issues as resolved or duplicated, and acts as a bridge between the test department and development. No software development organisation should function without such a system in place, although it's frightening that many do.

- **Source code editor.** A good editor will prevent you from making a whole pile of silly mistakes. Syntax highlighting often provides visual cues when you've made an error. You'll see when you mismatch comment delimiters, or get brace or parenthesis mismatches. A good editing environment also provides navigation around your code so you can find offending areas easily.

- A **version management system** stores the source code and a history of its development. It allows you to review changes that have been made, find out who made them and when. When a fault rears its head you can revert to a previously working revision and inspect the differences that have been made.

something unpleasant it will not normally just complain the once, but take the opportunity to sound off about life in general, spitting out a ream of other subsequent error messages. It's been told to do this; upon encountering any error the compiler tries to pick itself back up and carry on parsing away. It's not always too good at it, but with code like yours who could blame it?

The upshot is that the later compiler messages can all be quite random and irrelevant. You should only need to look at the very first error reported, and sort out that problem. Have a glance further down the list by all means, there may be some other good things down there, but more often than not there isn't.

Even this first compiler error may be cryptic or misleading, depending on the quality of the compiler (if you're really stumped by what an error means try another compiler, perhaps). Hardcore C++ template code can produce inspired errors from some compilers. The reported fault usually is on the line that the compiler reports, but sometimes it may actually be on the preceding line – a syntax error there causes the following line to be nonsensical, and this is what the compiler notices and moans about.

Linker errors, on the whole, are far less cryptic. The linker will tell you that it's missing a function or library and so you'd better go off and find it (or write it). Sometimes the linker may complain about arcane v-table related C++ problems, this is usually a symptom of missing a destructor's implementation or something like that.

Run time errors require a little more of a game plan. If your program contains a bug then it's likely that somewhere in the code a condition you believed to be true isn't. Finding the bug is a process of confirming what you think is correct until you find the place where the condition doesn't hold. You have to develop a model of how the code really works and compare this with how you'd intended it to. The only sensible way to do this is methodically.

Scientific method is the process scientists use to develop an accurate representation of the world. That sounds akin to what we are trying to do. There are four steps to scientific method: (i) observe a phenomenon, (ii) form a hypothesis to explain it, (iii) use this hypothesis to predict the results of further observations, and finally (iv) perform experiments to test these predictions. Now I'm not proposing that we use scientific method wholesale, for a start we're trying to *get rid of* the errant phenomenon rather than build a model of it. However, scientific method is a good backbone and you'll see it reflected in the steps below.

- **Identify** a failure. It all starts here, when you notice that the program doesn't do what it's supposed to. It may crash, it may just produce a yellow triangle, but you know something's up and you've got to fix it. The first thing you do is put a fault report into the fault database. This is particularly valuable if you're in the middle of tracking some other bug or have no time to handle the fault right now. Making a record ensures the fault doesn't get lost. Don't just make a mental note to come back to a problem later. You'll forget.

  Even if you're going to start fixing the fault immediately, having the record in the database serves a useful purpose – it shows other developers that a problem has been identified and is under investigation. It also allows reports to be generated about the number of issues remaining/resolved in the codebase.

  Identify the nature of the errant behaviour. Characterise the problem as completely as possible by answering questions like: is it timing sensitive, does it depend on input, system load, or program state. If you don't understand the bug before you try to fix it you'll just be changing code until the symptom disappears. You may only have masked a cause so the fault will crop up elsewhere.

- **Reproduce** it. This goes alongside characterising the failure. Work out the set of steps you can take to reliably trigger the problem. If there is more than one way then document them all.

  You have a problem if the bug isn't reproducible; the best you can do is set mousetraps for the fault and see what you can find out when it does occur. For these unreliable failures, keep careful notes of the information you collect, it may be a while until you next see the problem crop up.

- **Locate** the fault. This is the big one. You've got the scent, now you need to track the beast and pinpoint its location from what you've learnt. That's far more easily said than done. This is a process of eliminating all the things that don't contribute to the failure, or are working correctly, Sherlock Holmes-style. You may need to draft new tests. You may need to poke around in the seedy underbelly of the system. You will probably find that there is more information you need to gather as you progress.

  Analyse what you have found about the failure. Without jumping to conclusions, draw up a list of code suspects. See if you can spot patterns of events that hint at causes. If possible, keep a record of the inputs and outputs that demonstrate the problem.A good starting point for the investigation is where the error manifests itself – although this is rarely the actual habitat of the fault. Remember, just because a failure exhibits itself in one module that doesn't necessarily mean that that module is to blame. Determining this position is easy if your program crashed, you can use a debugger to get information like the line of code in question, the value of all variables at that point, and what called this function. In the absence of a crash, start from a point you know exhibits incorrect behaviour. Work backwards from there following the flow of control, checking that the code is doing what you expect at each point.

  There are a few common bug hunting strategies. The worst is randomly changing things to see the failure goes away. This is an immature approach. (A professional will at least try to make it look scientific!) A far better strategy is *divide and conquer.* Say you have the fault pinned down to a single function that consists of ten steps. After the fifth print out the intermediate result, or set a breakpoint and investigate it in your debugger. If the value is good then the fault lies in the instructions after this, otherwise it's in the instructions before. Concentrate on those instructions and repeat until you've cornered the fault.

  Another technique is the *dry run* method. Rather than relying on intuition to locate the error, you play the role of the computer, tracing program execution through a trial run, calculating all intermediate values, to get the final result. If your result and reality don't match then you know a fault lies in the code. Although time consuming this can be very effective, highlighting your bad assumptions.

- **Understand** the real problem once you've found where it's lurking. If it's a simple syntactical error then getting your head round it isn't too bad. For more complex semantic problems make sure you really know what the problem is, and all the ways that it may manifest itself before you move on.

- Create a **test**. Write a test case for the failure that exercises it. You may have done this in the 'reproduce it' step if you were clever. If you didn't, then you really want to write one now. With your new understanding make sure the test is rigorous.

- **Fix** the fault. See the following section for a discussion of this part.

- **Prove** you've fixed it. Now you know why you wrote a test case. Run it, and prove the world is a better place. The test case can be added to your regression test suite to ensure that the fault is never reintroduced at a later point.

Sometimes you try all this but it just doesn't work, you're left wailing and gnashing your teeth, with a sore head from banging it against a brick wall for too long. When things get this bad I always find it helps to explain the whole problem to someone else. Somewhere in the description everything seems to slip into place and I see the one key piece of information I had been missing all along. Try it and see. Perhaps this is why pair programming is such a successful strategy.

How to fix faults

You'll notice that this section is much smaller than the preceding one. Funny that. Usually the whole problem is finding the darned fault. Once you've worked out where it is, then the fix is obvious.

But don't let that lure you into a false sense of security. Don't stop thinking once you've diagnosed the source of your errant behaviour. It's very important not to break anything else as you make the fix – it's surprisingly easy to trample over something in the flower bed as you stroll over to pluck out a weed.

As you modify code always ask yourself 'what are the consequences of this change?' Be aware of whether the fix is isolated to a single statement,

[concluded at foot of next page]

Combining the STL with SAX and XPath for Effective XML Parsing

by David Nash

This article appeared (slightly edited) in the January 2003 issue of The C/C++ Users Journal. It is reproduced here by kind permission.

There are two main methods in common usage when parsing an XML document: The Document Object Model, and the Simple API for XML (SAX). Parsers that support the first method read the whole document into a data structure in memory, then provide access to it using the W3C's DOM API. This requires that the whole document fits into memory, and takes a little time while the parsing is done. Furthermore the user then has to navigate the DOM tree to gain access to the data in the document.

The second method is event driven, in that the parser calls user-supplied event handlers as it encounters occurrences of various parts of the XML document, such as Elements, Text and so on.

This article describes an efficient way to parse an XML document, using standard C++ library containers in conjunction with a SAX parser, resulting in fast de-serialisation of data from an XML file directly to data structures held in memory.

XML data may represent a variety of different kinds of data, plain character strings, integers, floating-point numbers and so on. A look at the W3C's XML-Schema recommendation shows the number of data types that have been anticipated and provided by this standard. We need a way to read from an XML text element into any one of a number of C++ data types. Ideally this should also be extensible for user-defined types.

We can achieve this by the use of a polymorphic `Element` class, which has the ability to convert and store textual XML data in any data structure the user wishes:

```
class Element {
public:
  virtual void put(const std::string& text)=0
                                          const;
  virtual ~Element() {}
};
```

Clearly this is a base class, as evidenced by the pure virtual method "`put`", and the virtual destructor which ensures proper behaviour if we delete objects of classes derived from this one via a base type pointer.

For each type of data we wish to be able to parse from the XML we need a new, derived, class with an appropriately-typed data member pointing at a variable suitable to hold the data item. There are two ways we could do this.

Explicitly deriving concrete classes

We can derive a specific class for each data type we need to parse. It must include a suitably overridden `put` method that can convert the character data from the XML document into the specific data type we need.

For example, a class for type `long` would look like this:

```
class LongElement : public Element {
  long* ptr_;
public:

  LongElement(long* ptr) : ptr_(ptr) {}
  virtual void put(const std::string& text)
                                          const {
    if (ptr_) {
      *ptr_=atol(text.c_str());
    }
  }
};
```

We would need to define a different derived class for each data type on which we want to operate.

Using a class template

I said there were two ways to do this, and we do have a choice of how to implement this: Inheritance polymorphism, or parametric polymorphism. The latter is more commonly known as "templates" in C++. Given that we are only changing the type of data operated on, why don't we choose to implement this as a class template?

The crucial factor is the design of the `put` method. Since each derived class handles a different type of data we need a way to code this function in such a way that we don't need to specialise the template for each type – which would negate the advantage of using a template. The ideal way would be to use a conversion function, which itself is a template. Luckily we have such functions in the standard `iostream` library, which makes its business the conversion of varied data types to and from character streams, such as we might find in an XML document.

The template version of the derived class looks like this:

[continued from previous page]

or affects other surrounding bits of code. Might the effect of your change ripple out to any code that calls this function, does it subtly alter the behaviour of the function?

Convince yourself that you have *really* found the cause of the problem, and not just another symptom. Then you can feel confident you've put a fix in the right place. Consider whether similar mistakes may have been made elsewhere in related modules, and go and fix them if necessary[6].

Finally, try to learn from your mistake. We must learn or otherwise be doomed to repeat the same errors for all eternity. Is it a simple programming error you keep making, or something more fundamental, the incorrect application of an algorithm?

Prevention

Anyone will tell you that prevention is better than a cure. The best way to manage bugs is to not introduce them. Sadly I don't think we'll ever completely reach this ideal, but careful programming can avoid so many problems. Good programming is about discipline and attention to detail.

This section could be enormous, but all prevention advice boils down to one simple statement: *Use Your Brain.* Enough said.

6 This is a good reason why "cut and paste" programming is bad - it is far too dangerous. You may end up mindlessly duplicating bugs, which then can't be fixed in one single place.

Conclusion

Like death and taxes, no matter how hard we try to avoid them, bugs happen. Sure, we should use every sort of anti-wrinkle cream available and manipulate our money in cunning ways to mitigate the effects. But if we don't know how to deal with faults when they stare us in the face then our code is doomed.

Debugging is a skill you develop. It doesn't rely on guesswork, but on methodical detection and thoughtful repair.

Pete Goodliffe

References

1. The Simpsons. *Do the Bart Man.* 1991, Geffen. GEF87CD.
2. ANSI/IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* 1984, ANSI/IEEE Standard 729.
3. John Gould. "Some Psychological Evidence on How People Debug Computer Programs." 1975, *International Journal of Man-Machine Studies.* No 7.
4. Pete Goodliffe. "Professionalism in programming #4: Code reviews." *C Vu*, Volume 12, No 5. ISSN: 1354-3164.
5. Pete Goodliffe. "Professionalism in programming #9: Defensive programming." *C Vu*, Volume 13, No 3. ISSN: 1354-3164.

```
template<typename T>
class ElementData : public Element {
  T& ref_;

public:
  ElementData(T& item) : ref_(item) {}
  virtual void put(const std::string& s)
                                    const {
    std::istringstream stream(s);
    stream >> ref_;
  }
};
```

You can see that, like the non-templated version, this class overrides the `put` method to put the XML data it is passed into the data storage it was given in its constructor. However this class can cope with any type for which a stream extraction operator has been defined.

Now, we have a number of classes (i.e., different instantiations of the template) that can hold a reference to a data item (a program variable, in other words). How are we going to manage objects of these types and how will they fit into the SAX parsing methodology?

Remembering that the SAX parser will call our handler object for the start and end of each element, and each piece of character data in the XML document, we need to arrange that the `put` method of the appropriate object be called at the appropriate time with each piece of data.

The best way to do this is to use a look-up table that will direct us to the relevant `ElementData` object for each piece of XML text. For this we will use the look-up table data structure supplied with the C++ standard library, the STL `map` class template.

Recall that the `std::map` takes two main template arguments, and these are the types of the key into the map, and the type of the item to be stored. In this case these are `std::string` and pointer to `Element` respectively. Let's declare a typedef to make our lives easier:

```
typedef std::map<std::string, const Element*>
          ElementMap_t;
```

So what do we use for the key of the map? Since we are mapping from each XML element to its data item, the key must identify the XML element in question. This means we should use its name as the key.

Let's put all this together and see how it can be used to parse the following simple XML document:

```
<?xml version='1.0'>
<Person>
  <FirstName>Elvis</FirstName>
  <LastName>Presley</LastName>
  <DateOfBirth>
    <Year>1935</Year>
    <Month>1</Month>
    <Day>8</Day>
  </DateOfBirth>
</Person>
```

We want to store each individual data item in a separate variable, each with its own `ElementData<>` object with the template instantiated for the appropriate type:

```
std::string FirstName;
std::string LastName;
struct Date {
  int year,month,day;
};
Date dob;
ElementMap_t element_map;

element_map.insert(std::make_pair
  ("FirstName",
   ElementData<std::string>(FirstName));
element_map.insert(std::make_pair
  ("LastName",
   ElementData<std::string>(LastName));
element_map.insert(std::make_pair(
  "Year",
  ElementData<int>(dob.year));
element_map.insert(std::make_pair(
  "Month",
  ElementData<int>(dob.month));
element_map.insert(std::make_pair(
  "Day",
  ElementData<int>(dob.day));
```

These can be wrapped in a set of overloaded functions to make it easier, or may even be automatically called by some program that gets the information from a metadata repository of some kind.

Then we need a SAX parser with a handler that looks up each XML element in the map, and calls the `put` method on the object it finds there, passing the character text from that element.

XPath

The element names we have used in the element map above are hardly descriptive. What if our XML document has more than one date, a start and an end of a period for example? We need a more specific way to identify the element in the document. This is what XPath was designed to do.

XPath allows us to specify an element in an XML document using a hierarchical directory path-like notation. The root of the document is represented by a slash, and each element name is appended, separated by more slashes.

Some examples from the document above:

```
/Person
/Person/FirstName
/Person/DateOfBirth/Month
```

XPath is much more expressive than this, but we are going to use this simple form of the notation to identify individual elements of our XML document.

The SAX Handler

There are several XML parsers around that include SAX capabilities. Here we will use the Xerces C++ parser from the Apache project as an example, although the technique could just as easily be applied to any other SAX parser. The handler class here derives from the Xerces `HandlerBase` class.

The SAX handler is the piece that does all the work. It has a number of methods that are called by the parser as the XML document is processed. In this case we are concerned with the beginning and end of XML elements, and with character text. We use the beginning and end element notifications to keep a record of where we are in the XML document, constructing an XPath string as we go along. This path to the current element is stored on a stack. Any character data for the active element is accumulated until we come to the end of the element. When we do reach the end of an element we pop the top item off the stack, so that the previous element's path becomes the active one.

Here is the declaration of the class:

```
class MySaxHandler : public HandlerBase {
  const ElementMap_t& element_map_
  std::stack<std::string> current_path_;
  std::ostringstream current_text_;

public:
  MySaxHandler(const ElementMap_t& map);

  void startElement(const XMLCh *name,
                    const AttributeList atts);

  void endElement(const XMLCh *name);

  void characters(const XMLCh *text);
};
```

The constructor simply initialises the object's member variable with a reference to the element map.

```
MySaxHandler::MySaxHandler
  (const ElementMap_t& map)
  : element_map_(map) {}
```

Handler Methods

First, `startElement` registers the start of a new XML element and adds it to the XPath name on our stack:

```
void MySaxHandler::startElement
            (const XMLCh* const name,
             AttributeList& atts) {
  std::ostringstream this_path;
  if (!current_path_.empty()) {
    this_path<<current_path_.top();
  }
  this_path << '/';
  write_xml(this_path, name);
  current_path_.push(this_path.str());
}
```

The reason for using a `stringstream` rather than a simple `string` is so that we can take advantage of the insertion function we will see shortly; this will handle conversion of XMLCh unicode characters to our local encoding. However, for reasons we shall soon see, this needs to be an explicit `write_xml` function rather than an overloaded `operator<<`.

Next, `characters` is called by the SAX parser for all textual element content. We simply maintain a `stringstream` and insert the new characters into it whenever we get some.

```
void MySaxHandler::characters(
     const XMLCh*const text,
     const unsigned int length) {
  write_xml(current_text, text);
}
```

Finally, at the end of each element, `endElement` finds the element in question by looking up its XPath name in the map and then calls `put` to write the characters saved so far to the stored variable reference:

```
void MySaxHandler::endElement(
                   const XMLCh* const name) {
  if (!current_path_.empty()) {
    ElementMap_t::const_iterator i
          = element_map_.find(
                      current_path_.top());

    if (i != element_map_.end())
      i->second->put(current_text_.str());

    current_path_.pop();
    current_text_.str("");
  }
}
```

XML Character encoding

The XML standard allows you to represent the characters that make up an XML document in any encoding you like. There are a number of rules used by XML parsers to determine the correct encoding, including the "`encoding=`" attribute on the "`<?xml ?>`" declaration at the beginning of the document.

The Xerces SAX parser represents characters using an `XMLCh` data type, and passes us strings by pointers to this character type.

These XML characters are represented in a Unicode encoding whereas to store them in standard `string`s we need them in our local encoding. Xerces provides a static `XMLString::transcode()` function to do this conversion. The conversion could be automated by building it into the insertion operator for the type `XMLCh`.

However, `XMLCh` is a typedef from `short` which makes it difficult – you can't overload based on a typedef because `typedef` does not create new type, but simply an alias. Therefore the standard inserter for `short` will be used by the compiler instead. To get around this problem there are a couple of alternatives: Use a different function to insert in the stream (rather than `operator<<`) or explicitly translate the encoding before inserting in the stream.

Here is the function `write_xml` used earlier to transcode and insert XML characters into a stream:

```
void write_xml(std::ostream& target,
               const XMLCh* s) {
  char *p = XMLString::transcode(s);
  target << p;
  delete [] p;
}
```

To avoid the call to `delete[]` you could replace the `char*` with a smart pointer capable of holding and deleting an array (unlike `std::auto_ptr`). If `target<<p` could throw, for example, this would be necessary to make the function exception-safe.

Using the SAX processor

With the handler class in place and now using XPath style element names, we can rewrite the parsing code. First, add a helper function to make it easier to add an element to the map:

```
template<typename T>
void AddElement(ElementMap_t& map,
           T* ptr,
           const std::string& path) {
  map.insert(std::make_pair
      (path, new ElementData<T>(ptr)));
}
```

The final code looks like this:

```
char filename[]="file.xml"
ElementMap_t element_map;

AddElement(element_map, &FirstName,
    "/Person/FirstName");
AddElement(element_map, &LastName,
    "/Person/LastName");
AddElement(element_map, &year,
    "/Person/DateOfBirth/Year");
AddElement(element_map, &month,
    "/Person/DateOfBirth/Month");
AddElement(element_map, &day,
    "/Person/DateOfBirth/Day");

MySaxHandler handler(element_map);
parser.setDocumentHandler(&handler);
parser.parse(filename);
```

Further refinement

An obvious enhancement is to enable multiple occurrences of the same element name in the XML document to store data in corresponding multiple variables, something which is not catered for by the code I have presented here.

Another way in which this design could be extended would be to support multiple XML document types. Since the element map objects contain full XPath names for each element, the same map could be used, and it would continue to uniquely identify each element as it is discovered in any known XML document type.

David Nash

Further Reading

XML: `http://www.w3c.org/xml`
Xpath: `http://www.w3c.org/TR/xpath`
Xerces XML parser: `http://xml.apache.org/xerces-c`
Tim Pushman, The SAX Parser, *C Vu*, December 2002

Patterns in the Web

John Morrison and Jonathan Heeley

During the development of a an internet based GIS system we came across the problem of maintaining session state on two separate server entities – the website (the application itself) and the mapping server through which the mapping component (a Java Applet embedded in the application) communicates. With our initial architecture, there was the possibility for the website session to timeout while the mapping session remained active. The consequence of this was the website session timeout had to be set to as high as possible a value to minimise the occurrence of this situation, which is a problem in our system due to the fact we are restricting clients to a certain number of concurrent logins and using the ASP `SessionEnd` event to log a user out.

A graceful solution to force both sessions to end was needed.

The solution came in the shape of the GOF PROXY pattern, which although it is a very simple and well known idea, its application is not immediately obvious in an IIS/ASP environment.

In summary the Java Applet makes requests to the proxy (an ASP page) which forwards the request on to the appropriate CGI programme on the mapping server and conversely feeds the response back to the applet. If the web session times out we can prevent requests going through the proxy. We can also return some text representing an error to the applet, which can close down gracefully.

This solution also has the benefit of allowing us to simplify general error handling in the applet itself. Any errors in making requests to the mapping server can be trapped inside the proxy page and fed to the applet in a more friendly or standardised way.

The final benefit is that we can implement simple load balancing (there are potentially several mapping servers) inside the proxy by using the ASP `Application` variable to store the address of the next server to use.

The technique is of course not limited to Java applets – any TCP/IP aware component can use it.

The design and code for the proxy is shown below in the style of a GOF pattern.

PROXY

Intent

Provide a surrogate or placeholder for another object to access.

Also Known As

- Surrogate
- Ambassador

Motivation

It may be a requirement to hive off some heavy processing from the main web server without losing the advantages of having a single web server as a point of contact.

Applicability

1 A forwarding proxy provides a web page to a client while forwarding the processing for that request to a different web server.
2 A protecting proxy provides a web page that includes logic to dictate when to allow access. (This could include changing port numbers).

Structure

Collaborations

Proxy forwards requests onto the real subject when appropriate, depending on the kind of proxy.

Consequences

The PROXY pattern introduces a level of indirection when accessing an object. In a web environment this can have large (time) overheads.

Sample Code

Ensure that the response is not cached and that all output is buffered. *Note that it is **very** important that the <% be the first lines in the file, especially when binary data may be proxied, as any leading data may corrupt what you are trying to proxy.*

```
<%
  Response.Buffer = True
  Response.CacheControl = "Private"
  Response.Expires = 0
```

Dimension all variables.

```
  on error resume next

  dim xmlHttp
  set xmlHttp
        = Server.CreateObject(
                        "Msxml2.ServerXMLHTTP")
```

Construct the url to proxy. This sample only proxies `GET` parameters, but could be easily modified to include `POST`.

```
  dim url
  url = "http://a.n.other.webserver.com/
                              scripts/gcis.dll"

  if(len(Request.ServerVariables(
                "QUERY_STRING")) <> 0) then
    url = url + "?" + Request.ServerVariables(
                "QUERY_STRING")
  end if
```

Get the request.

```
  xmlHttp.open "GET", url, False
  xmlHttp.send
```

If no error and the status of the request was OK.

```
  if Err.number = 0
              and xmlHttp.Status = 200 then
```

Depending upon the data type; for textual data use;

```
    ' plain text response
    Response.ContentType = "text/plain"
    Response.write xmlHttp.responseText
```

For binary data (commented out for this example as the data is text);

```
    ' binary data (e.g. gif image)
    ' Response.ContentType = "image/gif"
    ' Response.binarywrite xmlHttp.responseBody
```

Otherwise handle any errors (you can respond with html, or even do a `Server.Transfer` operation).

```
  else
    Response.write "Error"
  end if
```

Tidy up

```
  set xmlHttp = Nothing
```

Send the response and ignore any other instructions on this page. This is especially important when sending binary data as any further data may corrupt what you are attempting to proxy.

```
  Response.End
%>
```

Related Patterns

- ADAPTER
- DECORATOR

John and Jon

How to Talk Oneself Out of a Job

Colin Hersom <colin@hedgehog.cix.co.uk>

In this (true) tale, all the names have been changed to ensure innocence.

One morning a few weeks ago, I received a telephone call from John Dawes. John is an old colleague, but I hadn't heard from him for ages. He said that he had been commissioned by a client of his to find someone, or some company, who could do a piece of programming; would I be interested? John would not tell me who the client was, or indeed what they did, except that the work would involve parsing a language called Extracode, a language with which I was fairly familiar, and passing the results to an their internal database. He also said that they wanted to a first release of the code in about five months time, although the contract would not be given for over a month, leaving less than four months of development and test. I agreed to look at a specification if he could provide one, although I pointed out that although I was familiar with Extracode, I had not had occasion to write a parser for it.

I contacted Nigel who is the one person whom I knew would be capable of doing this work but is tied by a serious non-compete clause with his current employer. We discussed what this project might involve and were in agreement that 6-9 months was a reasonable estimate for the work without more detailed information. Writing the BNF is straightforward, although tedious, but getting the semantics of the language right takes time. There are a number of features that are really quite obscure, and some of these are not defined accurately in the standard, only by reference to existing implementations.

So a few days later the spec arrived. It was not a long spec, but gave a good indication of which parts of the language were needed, and that this project was to provide a front-end from Extracode to their internal database. It gave the features (by chapter number in the standard) in the order in which they were to be implemented, with a review half way through when a prototype should be available. At the end of the spec they stated: "This contract covers both Extracode and ESL. It is estimated that an ESL front-end would require a similar amount of effort again to implement as the original Extracode". ESL is another language designed to do similar things to Extracode; I have some experience of it, but not as much as with Extracode.

Alarm bells have started ringing. My original estimate of the project was 6-9 months, which would not meet the client's delivery expectations, and now they want to double it (on their estimate, not mine!) although I am fairly certain that they do not expect ESL within the original 4 month delivery date. I think that ESL is rather harder than Extracode, I do not know anyone who is more experienced in it, and the spec for this part of the project is even lighter than that for Extracode. Looks like this might tie me up fairly well full time for two years. If the client wants this to be a fixed-price contract (as John originally implied) then they can forget it. I am not even sure now that I want the Extracode part, let alone the ESL half. I might, however, be able to recover the situation if I can find some existing code that does most of what is required. Hello Google!

Searching the Internet revealed that there are two commercial offerings which will provide most of the functionality required. These from are Integrated Software Corp (ISC) and Vertical Solutions Inc (VSI). Both these purport to support Extracode and ESL, producing a common data structure. There is also an open source project called Pegasus Extracode which, although implementing Extracode only, might be sufficient to get things started and we can sort out the ESL later. I can't find any suitable open-source ESL compilers. I contact ISC and VSI and download the Pegasus source code to give it a go.

I speak first to a representative from ISC. He is very helpful and says that they have a product that seems to be a perfect match for the requirements. He gives me their pricing policy – somewhere in the region of $100K p.a. plus $5K p.a. per customer. He asks whether this is within budget, and I have to admit that I have no idea. I do not even know who the client is, let alone how much they are willing to spend. I will pass on the information and let him know when I have anything more to say.

Next comes VSI. They have a different way of operating – they supply the full source code to all customers, and want payment either as a lump sum or in installments. They trust their customers not to use the software if they don't pay. However the payments are made, they need to earn $350,000 over three years, and they will also take 15% for maintenance (which is required in order to keep up with changes in the standards). Therefore the pricing of the two commercial products looks about the same if the product is to be used in-house, but if it is to be sold, then as little as 20 customers doubles the price of the ISC offering.

I compile Pegasus and run it. It works after a fashion. However, there are fundamental features that are not implemented, like include files. This makes this option much less attractive. It is covered by the GPL, so the client would have to ensure that the front-end would write a text file, or have some other means of disconnecting it from the rest of their (proprietary) code in order to be able to give away the source when they sell the software. There appears to be a fair amount of work here, possibly not much less than starting from scratch, but what we do get is free.

I send off an email to John. It says much as I have done in the last few paragraphs, i.e. I do not want this contract as it stands; I do not think that the Extracode part will be done in time; I am not confident about the ESL part at all, and certainly not in the terms written; the client should look at ISC, VSI and possibly (if money is really tight) Pegasus. I expect John to say thanks, but no thanks. He doesn't. A few days later he rings to say that the client (whom he now identifies as a start-up called Excellarate) is really happy with what I have said and that they would like to meet. Oh, and by the way, you know one of them. I do? Oh yes, you were working with them years ago, and Mike Fish is looking forward to meeting you again. I was in a collaboration project with Mike over ten years ago, when John was in the same company as me, but John never met Mike a that time. Ah well, small world; laughs all round. Excellarate already use ISC and are unhappy with them (mainly, I think, because of that customer licence price), so are trying to find an alternative. They had not encountered VSI before.

So off I go on a train to meet John, Mike and someone I haven't met before, Alan, who is the director of engineering at Excellarate. They introduce their technology, I tell them about my experience – John and Mike know some of this, but it is all new to Alan. I then say (again) the thoughts that I had on this project. ISC is a good match – well spotted – so we quickly pass on to VSI. After a brief discussion of the other options it is clear that they are quite taken with VSI (and indeed this is what I am suggesting as the best option). Since VSI does most of what ISC already does, and so only the calls into their database need coding, is there anything left for me to do? Well, to be honest, no. Either they could get VSI to learn their interface and write custom code, or Excellarate could learn the VSI classes and write the interface themselves. Getting me to learn both the VSI and Excellarate systems seems like a complete waste of time.

They have half a dozen other bidders for this project and there is a meeting in a couple of weeks to determine the preferred option, so I am off to enjoy my holidays and will find out the results when I get back.

Back from holiday and I contact John to find out the situation. Excellarate were very impressed with me, and were delighted that I found VSI (they can't work out why they did not find VSI themselves) and the decision, still to be finally ratified by the board, is to use VSI and write the customised code in-house. That gives them their project easily on time, and gets ESL incorporated too. No work for me, and just as importantly, no work for the other bidders. John says that he put forward other individuals like me, small companies and some large multinationals so that Excellarate had a really good choice. It seems that the others stuck to the spec, and produced plans accordingly, and some of these bids were extremely expensive (maybe they didn't want the work either). I was the only one who took out the "how" of the spec and looked for the "what", to find an alternative solution.

So, I talked myself out of a job. Did I achieve anything? I think so. I showed Excellarate that doing it all themselves is not necessarily the best solution. I did not overload myself with work that I did not particularly want, and probably could not have dealt with anyway. And I earned some Brownie points: Excellarate has put me on their list of approved contractors. Doing nothing is sometimes far more satisfying than embarking on a large project. It is completed more quickly and is, of course, the ultimate in bug-free software :-)

Colin Hersom

I Wish They'd Use the Standard

Silas S Brown

I have many Chinese friends, so I have firsthand experience of the poor quality of many Chinese email clients. Email is supposed to be international; it is supposed to enable you to communicate with other countries, not just your own. But many Chinese email programs (and I'm talking about modern, up-to-date versions of such programs, not ancient ones) completely ignore the relevant international standards, making it unnecessarily difficult for their users to communicate with anyone of any other nationality. Their users may like the user interfaces, but they are oblivious to the poor quality of the underlying protocol-handling code, and when problems occur, they don't understand why.

Take FoxMail, for example. It formats dates incorrectly. I downloaded an evaluation copy and looked at the binary with a hex editor. The format string used to format minutes and seconds is `%d:%d`. What is wrong with that? It means that if a minute or second is less than 10, only one digit will be printed – 3:05 would be printed as 3:5. This could easily have been avoided by using `%02d` in the format string.

Big deal? Yes, because many spam filters (such as SpamAssassin) use incorrectly formatted dates as a clue that the message is spam. The number of times genuine Chinese messages have ended up in my spam folder is too high for comfort (and I don't like having to read through the hundreds of spams I get just to find them). Of course, I've adapted – I altered the filtering rules to be kinder to FoxMail, and I use BogoFilter in conjunction with SpamAssassin (BogoFilter is based on Bayesian probabilities and you can train it to your own samples of emails, so I get SpamAssassin just to tag emails with its test results and BogoFilter works out the real probability of the mail being spam based on my own sample). But I am a computer scientist. How many people out there are losing important emails because of problems like this?

(If anyone knows how to report a bug to FoxMail's developers, please do so. I can't find their contact info.)

But it gets worse. Recently I was trying to help a Chinese professor with the details of becoming a visiting scholar in the University of Cambridge, and she couldn't read the email that told her what the rent was. There was nothing wrong with the email in question: It was formatted using the ISO-8859-1 (aka Latin-1) character set, and it used the UK Pound sign (code 163), which was MIME-encoded (using "quoted printable") and the headers clearly stated which character set the MIME encoding was using. But the Chinese professor's email client (which didn't even identify itself, but apparently it's being used by Tsinghua University which is widely considered to be China's top university) did the following: It decoded the MIME quoted printable (so we know it's modern enough to understand MIME), but it completely ignored the header's statement of the MIME character set; it assumed that all incoming messages are in the Chinese GB-2312 coding. It then tried to interpret the pound sign and the following byte as a Chinese character; this failed, so it replaced the two bytes with a question-mark. As a result, the first digit of the price was lost. She tried to forward the email for me to read, but the damage had already been done: the client provided no means of forwarding an email without re-interpreting its characters first. I had to contact the original sender for the information.

Everyone concerned thought it was their fault, but it's not. It's due to a poorly-implemented email client. Perhaps the programmers had such limited time that they couldn't properly learn and implement the standard, so they only implemented just enough of it to make it work in their test cases. After all, Cambridge University's "WebMail" system is hard-coded to use the ISO-8859-1 character set (because the authors didn't have the resources to support other character sets and still make sure the program is secure against cross-site scripting attacks and so forth); at least this limitation is well-documented and the character set is identified in the headers of outgoing mail. But there is definitely room for improvement, especially if you are an establishment that has a policy of encouraging exchange with a certain foreign country and your staff members can't read emails that are sent from that country.

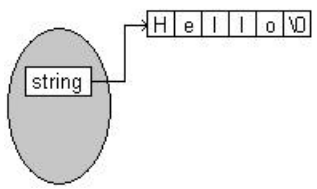
More generally, there is obviously scope for a greater promotion of good programming practice. The free software movement tends to get it right, because problems like these are fixed by the public before the program becomes popular. But many establishments prefer to write or buy proprietary software with insufficient support for the standards. That may be their loss, but it's everyone's loss if it makes it more difficult to communicate with people just because they happen to be in an establishment that uses broken software. More people need to be aware of this, especially if they're interested in promoting international exchange. The general standard of programming practice still has much room for improvement.

Silas S Brown

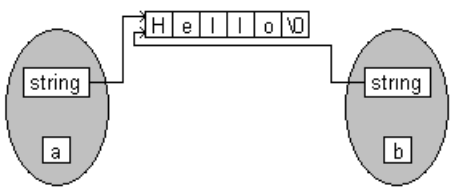
Copy on Write

Paul Grenyer

What exactly is Copy On Write (COW)? This has been the source of some discussion on `accu-general` just recently. Copy on Write is a lazy optimisation [1]. Consider a string-class implemented in such a way that it only stores a pointer to a string of data [2].



The normal way to make a copy of such an object would be to create a new instance of the object, allocate enough memory to hold the string and then copy the string from the old object into the newly allocated memory pointed to by the new object. This is called a Deep Copy. However, if the string data is particularly long it may be quicker to just copy the value of the pointer from one instance of the string-object to the other and have both instances pointing at the same string data.



Then you have to make sure that if one of the string-objects modifies the string data the other does not get its string data modified as well [2]. So we need a way of determining whether more than one string points to the same data [3]. This is achieved by introducing a system of reference counting. Every time a new copy of the string-object is created the reference count in incremented. The reference count is decremented again each time one of the string-objects is destroyed. When the reference count falls to 0 the string itself is destroyed.

But how does this stop an instance of the string-object modifying the string data pointed to by all the other string-objects? Each time a string-object wants to modify the string data it checks the reference count. If the reference count is 1 it knows it is the only string-object pointing at the string data and it can modify it however it wants. If the reference count is greater than 1 it must perform a deep copy (as described above) and decrement the reference count by 1. This process of not performing a deep copy until the 'data' needs to be modified is termed Copy On Write.

However, for many programs though the average string is approximately six characters long. The extra overhead of COW, including the difficulty of identifying when a write operation is about to take place [3], takes more time than just doing the deep copy. Once multi-threading is taken into account it becomes very difficult to make COW work correctly for an object such as `std::string` and the extra overhead of needing to synchronize (even for distinct strings used in different threads, if they might be copies of the same string) is an extra nail in the coffin [4].

Paul Grenyer

[1] Jim Hyslop
[2] Terje Slettebø
[3] Phil Nash
[4] James Dennett

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***.

Mac OS X Tech Talk Tour: UNIX on the Desktop

Thaddaeus Frogley

On a bright clear day on the outskirts of London, I sat in a lush hotel seemingly staffed exclusively by beautiful people with exotic accents, waiting for my sandwich. Businessmen in expensive looking suits, and hippies with expensive looking Apple notebooks surround me.

As I read about the war going on halfway around the world in the 'free' newspaper laid on by the hotel I can hear two men with beards wage their own private battle with their network configuration and the connection to the hotel's wireless network gateway.

There is an O'Reilly stand selling Unix and Apple books outside the conference room. Later I will learn that they are discounted by 25%, but by then it will be too late. There is a "free" CD on each chair, containing the latest version of the Apple Developer Tools.

There are maybe 100 people; just over half the seats are occupied. I overhear someone observe that there is only one woman – an unusually low male to female ratio for an Apple event, apparently.

The presentation will be delivered using Apple hardware, but initially there is a problem with feedback from the radio mic.

System Architecture and UNIX

The first presentation is an overview of the OS X System Architecture. They talk about how Quartz uses PostScript, and how well Quartz Extreme integrates with hardware acceleration.

They go on to talk about the wide selection of languages available to the developer, and how the POSIX compatibility is "mostly done", and that Apple are now committed to Open Standards. For graphics / UI development there is OpenGL, GLUT, X11, and of course Carbon and Cocoa. They noted that AliasWavefront used X11 as a way to rapidly port Maya to OS X, and went on to get 20% new sales on the platform.

There was mention of Fink, which pretty much got an official recognition, despite Apple admitting that they were looking into doing their own BSD style "Ports" system.

Developers moving from UNIX need to be aware that HFS (native file system) is case insensitive (but case preserving), and that resource forks are still supported, but should be thought of as deprecated.

Developers can of course use ".dot" files for configuration, but that OS X comes with its own XML based system for this.

There was talk about Frameworks, and all the places that they can live. For example Safari has two unfinished frameworks, "webcore" and "webscript", which are currently bundled into the application. When the APIs are stable Apple will move these frameworks into `/System/Library/Frameworks/` where they can be used by everybody.

Java

Java 1.4.1 is now available for Mac OS X. Apple's Java engine is now based on Cocoa, which is closer to the Java model than Carbon, which was the framework 1.3.1 was written with, and so the Java VM is tighter and less bloated (300 implementation classes as opposed to 900 previously). 1.4.1 also has better Safari and Keychain integration. The new JVM also has the new shared system library technology, which reduces the overhead in running multiple Java apps on one system. Apple are clearly very happy with the new JVM, with claims that it is "the best JVM implementation in the world".

Cocoa

Cocoa, the Objective-C API for OS X is a "proved" RAD environment. The Apple developers use it, and Project Builder to do their internal development, and the quality and quantity of the output of the software division of Apple has measurably increased.

On a personal note I was very sceptical about Objective-C, but have to say I was impressed by what Apple had to say about this. A lot of people claim this or that about whatever development language, environment, or methodology, but very few have actually done any genuine productivity studies to back up their claims.

Anyway, they talked a lot about how Objective-C worked, but it's not something I want to try to go into details here – I wrote a lot of notes, but I don't think it would make very good reading. If you're interested there are lots of good books. The demo was very very impressive, in that they wrote a simple application from scratch in front of us, then delved into the application bundle, and changed one of the GUI widgets from a text entry box to a slider-bar and the app worked with the new UI without recompilation.

AppleScript

The final presentation was on AppleScript. Apple considers AppleScript to be a first class citizen of the developer tool world, and something that you can use to write real applications. It's not just Yet Another Scripting Language. It is a Mature Language that they have done a lot of research into and done a lot of work on and in.

You can embed shell scripts in AppleScript, and you can script Java apps with it. Basically any GUI app on the Apple platform "supports" AppleScript, because AppleScript support is built into the OS.

Thaddaeus Frogley

Reviews

Bookcase

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

Francis Glassborow writes:

There are over 250 books (literally thousands of pounds worth) sitting in stacks on the stairs to my office waiting for someone to review them. Of course some of them are rubbish but many of them are worth reading if the subject is anywhere near your specialisms or one that you want to learn about. One thing about reviewing books is that even reviewing a bad book is beneficial to its reviewer. Taking on the task of looking at a book in a critical fashion changes the reviewer. This isn't just my opinion but that of many of those who have responded to the challenge and reviewed a couple of books for this column.

If you do the job properly you will find that it sharpens your eye so that you begin to spot errors of all kinds more easily. This helps when you come to buy your next book or look in your local library. I know books are expensive but their cost in money fades into insignificance when you look at the investment of time spent in studying a technical book. None of us have enough time to do everything we want, which is why we legitimately resent time wasted on a book that proves useless to our needs. At least when you are reading in order to write a review you can feel the benefit of sharpening your critical faculties as well as get a warm feeling when you are either able to warn others away or point them at a book that they might otherwise have missed.

If you have never done a book review, please give careful consideration about trying it at least once. At least you are in the tiny minority who understand the value of reading, unlike a C++ programmer that I lent a copy of Effective C++ to, when I recovered it six months later he still had not read a single item from it. Sadly, he is in the majority of people who claim to be professional programmers (in the sense of making a living from writing code).

Francis

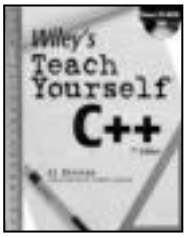
The following bookshops actively support ACCU (the first three offer a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing 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 1.4 or better as appropriate (in a context where the true rate hovers around 1.5). I consider any rate below 1.25 as being sufficiently poor to merit complaint to the publisher.

C & C++



Teach Yourself C++ 7th ed by Al Stevens (0 7645 2644 8), Wiley, 704pp + CD @ £22-95 (1.52)
reviewed by Alec Ross

The intended readership of this book is programmers who want to learn C++. Cutting to the chase, I would certainly recommend this book for most people in that position.

It is a big book, at just over 700 pages; but for its intended purpose, of course, a difficulty is in what to leave out. To my mind Al has done an excellent job in his choice of coverage and in the detail given.

In my humble opinion there is now a range of good C++ textbooks using completely different tutorial designs. (Just think of *Accelerated C++*.) For introductory training in C++, whether it is delivered in a book or otherwise, there are important choices as to the sequence in which certain key concepts are introduced. In particular, the position of the description of classes, ABCs, templates and the standard library affects what can be used in the examples for other topics. It can, no doubt, also affect how the learner starts to think 'in' C++.

Teach Yourself C++ introduces OOP and data abstraction fairly early on (p112) and then again in the final chapter. Access specifiers and member functions are illustrated with `struct`s (at p105), with forward references to the later detailed coverage of classes. Function template basics are covered around p193; class template basics are at p375, just before four chapters on the standard library.

Other design dimensions for such a book are in the emphases put on abstract and concrete aspects, e.g. on OO design principles and on language details. This book is very much focused on the concrete, as reflected in the choice of the domains(s) used in code illustrations. Al has chosen to use what he considers realistic examples that his audience can relate to, rather than anything concocted or abstract. These examples all have some application content. As a result they do not use generic names such as 'foo' or 'X'. I expect that the majority of beginning C++ programmers will be thankful for this choice – though they may move on later to seeing the abstracted style.

The code examples used are complete programs, not simply fragments, and are supplied on the book's CD. Building and running and editing and experimentation, are very simple with the supplied software. The main component of this is Quincy 2002, an IDE that front-ends MinGW (also on the CD).

I would have liked some more graphics and particularly canonical UML for the class diagrams, but the bottom line is as at the top; this book is highly recommended.

Visual C++ .Net: A .Managed Code Approach by Deitel & Deitel (0 13 045821 X), Prentice Hall, 1622pp @ £47-99 (1.25)
reviewed by Asad Altimeemy

This book is not for the experienced programmer. It is ideal as a beginner's guide to Visual C++.Net. It is not an in depth book, which is surprising, given the title and the 1500 page volume. However, the text is an extensive 'How to book'. It covers many areas of .NET and does provide a good starting point by giving many useful tips.

It begins by using a series of tutorials to explain and guide the reader. Then there is an introduction to Visual C++ .NET fundamentals. We are moved rapidly to more advanced topics including; multithreading, ADO .NET database integration, ASP .NET Web services, network programming and XML processing. The manual closes with a detailed treatment of unmanaged code, including attributed programming, Web-based application development with ATL Server and managed and unmanaged interoperability.

Developers will find that Visual C++ .NET offers them good flexibility in writing managed and unmanaged code and ATL-Server applications. However, the reader must remember Visual C++.NET is not 100% in compliance with C++ standards. Nevertheless, it is a good development language/tool to develop software.

This is a better than average book, ideal for somebody new to programming. It would not suit an experienced programmer.

[I think that the author means anyone who has any programming experience when he writes 'experienced programmers. Francis]

Inside Microsoft Visual Studio .NET (2003) by Brian Johnson et al (0-7356-1874-7), Microsoft*, 542pp @ £36-99 (1.35)
reviewed by Ralph McArdell

I was looking forward to reading Inside Microsoft Visual Studio .NET (2003) as I have a need to extend Microsoft Visual Studio .NET – the focus of the book.

In the main I was not disappointed and even picked up some everyday usage hints. Most subjects are covered in detail – occasionally to the point of being slightly tedious. These mostly concern macros and the various forms of add-ins and topics related to them, and include customising

Visual Studio .NET help and designing setup projects – useful if you plan to distribute your add-ins.

In one or two places the book is less detailed, describing a solution based on the downloadable code that accompanies the book. This was understandable as these cases tended to require knowledge beyond the scope of the book, such as writing ActiveX controls.

I found the material generally easy to read and understand but the text was marred by several obvious typos. Although covering the latest 2003 version of Visual Studio .NET most of the material also applies to the 2002 version. I particularly liked the notes and asides on tips, bugs and pitfalls, which will no doubt save hours of frustration.

On balance I think this book will prove to be as useful as I had hoped and I would recommend that anyone who wants or needs to know more about extending and customising Microsoft Visual Studio .NET take a look at this book.

C#

C# Unleashed by Joseph Mayo (0 672 32122 X), Sams, 900pp @ £36-50 (1.37)

Paul F. Johnson

Sams in the past have come into a lot of criticism for their C++ in <insert unrealistic number> Days and most of the time, it is well founded. As such, they have had a rough ride in the book reviews.

This time, they have a fantastic book. It makes one assumption (which is fair enough given that it is aimed at intermediate/advanced level) and that is that you're coming from a C++ or Java background. That said, if you're fresh in with a bit of brains, you should be able to follow the book.

The book covers the language in great depth with plenty of good examples with clear, concise explanations of the code. Every aspect of the pure language is covered with tips and hints for C++ and Java programmers.

It is not a light book, weighing in at a mighty 820 pages (or so) and many would be thinking that it's quantity over quality. This is not the case here. Take the sections on file handling. An entire chapter is given over to this subject (around 40 pages) with even more later on in the book. Even in some of the best C, C++ and Java books around, I have not seen this much given over to the file handling routines.

As it is the pure language that is covered (for the most part), the book is an invaluable resource for those using GNU .NET and Mono. Towards the very end of the book, it begins to cover the Windows specific parts, however, these parts are described and explained as well as the pure language part, so when the Windows parts are finally available, the use of these parts will be have already been covered sufficiently.

The source code from the website is clear and well documented.

There is only one drawback with the book – it doesn't tell you how to compile the source code from the command line; it is assumed that you will be using the Visual Studio IDE (and debugger). While I don't have that much of a problem with this as such, having the command line compilation command would given the book that little bit extra.

C# Class Design Handbook by Richard Conway et al (1-86100-708-6), Wrox*, 365pp @ £28-99 (1.38)

reviewed by Huw Lloyd

No less than 5 authors have contributed to this 'handbook'. The principal author (his name is listed first) has a namesake whom I wish they had all attended to. I am referring to Conway and his famous law of architecture. This book's contents sorely reflect a paucity of shared focus and communication.

The apparent focus is C# class design, yet there is very little information presented on class design. If I picked up on the authors' tones correctly only one of them mentions class design issues. These are sprinkled about in bullet point form. Given that 'design' is such a loaded word, what I am referring to as absent from this text are references to such concepts as coupling and coherence, commonality and variability, library leverage, representation, orthogonal separation and domain analysis, scale, testability, variants and invariants and type leverage – all with respect to C#.

A more suitable title for the book may be 'C# Class Preliminaries'. More than half the book is devoted to spelling out the basics of the C# language. This is not achieved very well either. For instance the keyword `internal` restricts method access to assembly local calls, not as one author would have us believe to calls from the same namespace.

Much of the content is quite pragmatic, it just really hasn't been thought out properly. When you consider that class design is such a rich and interesting domain, I would expect anyone who knows a thing or two about classes to be disappointed by this book.

Not Recommended.

GDI+ Programming: Creating Custom Controls Using C# by Eric White (1 861006 31 4), WROX*, 518pp @ £36-99 (1.35)

reviewed by Duncan Kimpton

This is quite a weighty book that tries to cover a lot of ground: GDI+, Custom Controls, ASP.Net Controls and Web Services.

The book is well laid out with appropriate use of screen shots and code samples.

Early on the writing style seems obfuscated by rolling the introductions to all sections into one chapter. However once the author gets into his stride and begins to concentrate on the topic at hand then the style becomes fluid and easy to read.

The section on GDI+ is exceptionally well written and leaves almost nothing to be desired in terms of an introductory text.

.Net Custom controls are covered succinctly and build upon the GDI+ material, however if I were being highly critical I would have been happier with a little greater depth of study.

ASP.Net and Web services are covered sparingly, but the author manages to get across enough of the basics that it will serve as a good springboard for further study.

Attention to detail and consistency of style are a big bonus for this book, which makes it much easier to absorb the information presented.

Whilst I feel that the content of this book might have been better handled as two separate books I cannot deny that the author has done a superb job with the space available to him.

This book will be taking a well-earned place on my reference shelf and is worth every penny of its cover price.

Highly recommended

Java

Micro Java Game Development by David Fox & Roman Verhovsek (0 672 32342 7), Addison-Wesley, 548pp @ £36-50 (1.37)

reviewed by Matthew Strawbridge

The target audience for this book, according to its 'mission', consists of professional game designers, games programmers, Micro Java enthusiasts and micro gamers. However, it soon becomes clear from the informal language used throughout ('*there are already scads of cool Java applications and applets out there*', to, most tellingly, '*this isn't your dad's old reliable J2SE Image class*') that it is aimed at the youth market. Since knowledge of Java is assumed, but the concept of gaming is explored in painful detail from the ground up, this seems rather strange.

If the style grates slightly, the content really annoys. It is the errors, which are mainly typographical, that cause the problem and it's fair to say that there are 'scads' of them! One paragraph even reads as follows

> *Although keeping an By default, HTTP connections are kept alive This means that the same connection will be used for multiple requests. HTTP connection alive is well and good, it is usually cleaner and easier for a server to close out a connection after every request.*

This slap-dash approach is also evident in the code snippets – random indentation is the rule rather than the exception and several of the classes seem to have been renamed since they were first written, so that the code and the commentary do not tie up. When some code is rewritten to reduce the size of its class files by a third, the text suggests that the size is reduced from 654KB to 431 bytes.

It is worth noting that one of the authors is from Slovenia and so it is fair to assume that the book has already been subject to a fair amount of copy editing – but clearly not enough.

It's not all negative – there are some useful tips and the source code for a whole game – but I cannot recommend this book in its present ragged state.

Other Programming Languages

Mastering Kylix 2 by Marco Cantu & Uberto Barbini (0 7821 2873 4), Sybex, 688pp + CD @ £37-99 (1.32)
reviewed by David Nash

Going by the title, it would appear that this book was already out of date when I received it, since version 3 of Kylix, the Linux port of Borland's Delphi, had been available for some time. However although Kylix 3 is probably of more interest to ACCU members generally, since it includes C++ as well as the Pascal-based Delphi language, it seems that this addition is the major part of the upgrade and you can use this book very well to learn the Delphi part of the package. Furthermore the authors have not released a Kylix 3 edition of the book, presumably because they are Delphi experts not C++ experts.

The book is not for the beginner. It is well-written, in an easy-to-read style but assumes a degree of familiarity with programming. Naturally this would not be a problem for ACCU members. It does start with a brief technical introduction to Linux, which is somewhat superfluous, since I believe anyone contemplating writing software for Linux would probably already know at least the basics, which are all that this introduction covers anyway. If you need to learn about Linux before beginning Kylix programming I recommend getting a book on Linux or a knowledgeable friend.

Next in the book comes a description of the IDE and the structure of a Kylix project. This is done well, with plenty of information including a surprisingly long table containing a description of every file type produced by the system.

The next couple of chapters cover the Object Pascal language and object-oriented features. Frankly, to a C++ programmer there are not too many surprises. Everything you expect is there, with plenty of examples. Most programmers would not have any problems with this.

Now, the more important parts of the standard libraries and, since this is a graphical programming product, the visual (QT-based) programming facilities. Again, there are plenty of examples covering components, events, forms, database programming and so on, until the (almost obligatory these days) web and XML section.

The book has been ported, if that's the right word, from the authors' Delphi book and given that Kylix is heavily based on Delphi that's no bad thing, although it does show in places. However there are plenty of Linux-specifics and sections on, for example, the X window system and Linux-style shared libraries.

A CD is provided containing the Open edition of Kylix (containing less features than the paid-for edition, which the book points out where necessary), source code, free books on Object Pascal and SQL (in PDF format) and other software including Interbase.

In conclusion, I would recommend this book to anyone contemplating beginning development using Kylix.

Development Support & Methodologies

Component Software 2ed by Clemens Szyperski (0 201 74572 0), Addison-Wesley, 589pp @ $54-99
Reviewed by Andrew Marlow

This book covers the emerging area of component software very well. It is comprised of five parts: motivation, foundation, models and platforms, architecture and process, and markets. The motivation section would have benefited from taking some of the market summary information from part 5, in order to give the reader a better sense of where we are now. This, coupled with a plodding analysis in the foundation section, makes the book rather hard to get started with. The architecture and process section covers the current technical approaches very well, much expanded from the first edition, and contains a lot of technical material. Towards the end of this part there is a slight and welcome slacking of pace as component assembly and development is discussed. This lighter pace continues in the last section, which provides a good summary of the current state of play and possible future directions.

The book covers a lot of ground, having at least something to say about practically every aspect of the subject. I recommend it for senior developers, designers and architects. It provides useful material on what technologies are available, what standards are emerging, who the relevant standards bodies are and what commercial companies there are, and what similarities and differences there are between approaches. This material will probably not be of interest to the average developer and the whistle-stop tour may be too fast for some readers, but this is why it is recommended for more experienced people. Even then, it would help the reader to already have been exposed to some of the areas.

The book is very dense, there are few pictures or diagrams and the tone is not light very often. This, coupled with the enormous amount of factual material, makes the book quite hard going. It is worth the effort though, especially in the last third of the book by which time the purely technical material has ended and other factors, such as frameworks, tools, assembly and marketing are covered.

The author points out in the preface that he now works for Microsoft. He is aware that this could cause accusations of bias in advocating certain component technologies over others. He says that he has enlisted the help of two co-authors to ensure fair representation of competing approaches. In my view he has succeeded. The section on J2EE was one I found particularly interesting.

The book is a vast improvement over the first edition, even though the first edition was also good. This is due to better coverage of certain basic topics such as 'what is a component', and how does 'component software differ from object-oriented software'. This was only touched on in the first edition but much has been added in the 2nd edition to address this, making the book somewhat larger. Another area of improvement is the (new) section entitled 'what others say' which provides some useful comparisons between the views of various authors on components.

In conclusion, I recommend this book but with the proviso that it is quite hard going and is better suited to more experienced senior developers. It is essential reading for any system architect.

Effective Software Testing by Elfriede Dustin (0-201-79429-2), Addison-Wesley*, 271pp @ £26-99 (1.30)
reviewed by Daire Stockdale

This is an 'exactly what it says on the tin' type of book. Inside its covers you will find fifty clearly written and very specific suggestions as to how you can improve your software testing. They are grouped into ten headings, as follows; requirements, planning, the team, system architecture, design and documentation, unit testing, automated testing tools, selected best practises in automated testing, non-functional testing and managing test execution. I liked the writing style, which is very clear, concise and to the point and coupled with the fact that I was nodding in agreement in every section, makes this a pleasant book to read. I would even go so far as to say the author should simply publish the 50 topic headings, which are all imperatives such as 'Involve Testers from the Beginning', as they would make a useful check-list to beginning any project.

If I had to have a complaint about this book, it's that perhaps some of the items are a little too obvious, e.g. Item 19 'Verify That the System Supports Debug and Release Execution Modes'. It's difficult to imagine a project where this would be omitted. This probably isn't a fair criticism, as the inclusion of such items makes for a thorough list of items on a testing wish-list, but if you consider yourself experienced in planning testing, then you might find that although you agree with almost all of the book, you have actually learned little from it. I would argue that the book's relatively low price would still make it a worthwhile investment for a company, if only as a useful set of guidelines for planning project testing. For a small or new company considering taking testing and quality assurance more seriously, then this book is also a good starting point for developing a company policy.

Software Engineering, An Object-Oriented Perspective by Eric Braude (0 471 32208 3), Wiley, 534pp + CD @ £26-95 (1.85)
reviewed by Daire Stockdale

This book is a quite thorough text on software engineering, clearly written by someone with experience of large-scale projects. I read the book cover

to cover and one of the first things that struck me about it was that it was clearly written by an engineer (and I don't intend that as a criticism). I enjoyed the book and learned a lot from it, so I'll get my only real complaint about it in early; it's a very dry, technical read and is reminiscent of university textbooks. I suspect that the author is aiming towards the 'prescribed reading' academic text market and I certainly think it will fill this role very well. However, it might require an unusual interest in the technical aspects of large-scale project software engineering (or the need to review it) to propel the reader completely through this book. The language is quite jargon and acronym heavy and although the book is attempting to commit these words to the reader's vocabulary, it does occasionally make the book a chore to read.

That said, the book is otherwise a good, thorough text and I would recommend it to be considered as a course textbook by anyone in the position to do so for a relevant course. The book covers in good detail the process of planning and documenting a large-scale project and illustrates this with a sample case study, which runs through the book. Chapters include project management, requirements analysis, software architecture, detailed design, unit implementing and testing and maintenance. The book focuses heavily on the IEEE standards for software and if your company is required to use any of these for a project, this book might be useful.

Each chapter breaks down its topic into sub-headings and the subjects are explained well and amply accompanied by illustrations, graphs, flow charts and UML diagrams where necessary. Each ends by summarising and then the contents are put into practice in the sample case study. Exercises are provided and many also have sample solutions or hints. The author does not dictate methods and often suggests several ways to approach aspects of projects.

In the chapter on maintenance the author quotes a study whose results surprised and amused me and which I must mention in this review: the author quotes a study which says that improved methods of system development actually results in more rather then less maintenance! The reason is that better-designed applications, being easier to change, are more likely to be adapted to new circumstances.

In summary, I think this is a well-written and detailed book about software engineering. It is not an easy read, but would be useful as a university textbook or to a company undertaking a large IEEE standardised project.

Performance Solutions by Connie Smith & Lloyd Williams (0 201 72229 1), Addison-Wesley, 510pp @ £37-99 (1.32)
reviewed by James Roberts

This book attempts to define a method for addressing performance problems with software, with some particular emphasis on spotting problems before they manifest themselves.

The authors have a distinct method for handling performance issues within the development cycle. They emphasise the importance of tackling performance items as early as possible in the development cycle (taking a few sideswipes at XP on the way).

The method generally seemed sensible, although there were no eye-opening insights and the book seems short on guidance of how to sensibly make the requisite estimates to drive the process.

The book had a couple of annoyances. For example, the chapter on 'Web Applications and Distributed Solutions' had entire paragraphs quoted from the chapter on 'Web Applications'.

On the plus side, the presentation was good and the book was quite readable.

There were some useful ideas in the book. However, I felt they were somewhat few and far between and although they were an interesting summary, they were not a revelation.

The main problem that this book has is that it falls between two stools. Although some of the ideas in the book might be interesting and useful for someone with little experience in performance management, the book is somewhat specialised (and long) for that audience.

It might form a useful chapter in a more general-purpose textbook. For a performance specialist, I didn't feel that the book offered enough of interest to be worth buying.

DSDM by Jennifer Stapleton (0-321-11224-5), Addison-Wesley, 235pp @ £29-99 (1.33)
reviewed by R N Lever

DSDM is the Dynamic Systems Development Method, which was created by a not-for-profit consortium that formed initially in the UK in 1994 but now includes USA, Benelux, Denmark, France and Sweden. It describes project management, estimating, prototyping, timeboxing, risk and configuration management, testing, quality assurance, roles and responsibilities, empowered team structures and so on. In short, everything you need to develop software that has active user involvement, is fit for purpose and meets business need with a frequent delivery cycle.

The book is split into four parts with the first part describing the framework; process overview, principles, time versus functionality, people working together, agile professional. Part two is a set of ten case studies that have been taken from various different types of projects and sources but share the common thread of being a DSDM based project. Finally there is additional information and appendices. This book is aimed at those involved in delivering IT focused projects and is really an introduction to what DSDM is about and some examples of where it has been used. It is not aimed at being a reference or a how to manual and those readers who already know about DSDM but wanted to go deeper will need to contact/join the DSDM consortium.

Whilst the book is readable and provides a useful introduction to DSDM it is really a taster for understanding what the DSDM consortium manuals contain and which are the real source of value. Anyone who reads this book in the expectation of being able to practice DSDM on their next project will find that whilst the book contains the 'what' there is very little 'how'. However, if it is understood and treated as an introduction then it serves its purpose well enough.

Non-Programming

sendmail 3ed by Bryan Costales (1-56592-839-3), O'Reilly, 1204pp @ £42-50 (1.41)
reviewed by Asad Altimeemy

This versatile well-illustrated book is detailed enough to take the place of a reference text. The book begins by guiding you through the building and installation of Sendmail and its companion programs, such as Vacation and Makemap. These additional programs are pivotal to Sendmail's daily operation. Then you cover the day-to-day administration of Sendmail. This section includes two entirely new chapters, 'Performance Tuning' to help you make mail delivery as efficient as possible and 'Handling Spam' to deal with Sendmail's rich anti-spam features. The next section of the book tackles the Sendmail configuration file and debugging. Finally, the book concludes with five appendices that provide more detail about Sendmail than you may ever need.

The text has kept pace with all of the evolving versions of the software (up to the 8.12 edition). A whole chapter on Spam and Anti-Spam has been added to this latest edition. Other new additions include an elaborated chapter on Performance Tuning; as well as expanded coverage of Vacation and Makemap. Every new option regarding the software (version 8.10 through 8.12) is included. Yet the simplistic and accessible outlook of the book remains. This is a good text to own, although some of the very detailed sections are unnecessary. It is organised and well written. This guide is an excellent update from the 2nd edition and useful to anyone who administers a Sendmail server, including STMP authors.

Content Syndication with RSS by Ben Hammersley (0-596-00383-8), O'Reilly, 208pp @ £20-95 (1.43)
reviewed by Christopher Hill

'The wonderful thing about standards is that there are so many to choose from'. Sadly this old chestnut does apply to content syndication on the Internet. Hammersley reviews the four current 'standards' – RSS 0.91, RSS 1.0, RSS 0.92 and RSS 2.0 (this would appear to be the correct chronological order and a sign of the factions with the content syndication arena).

The heritage of each 'standard' is explained and the key differences from its predecessor are highlighted. There is a useful

section on how to publish your feed, covering links in the HTML pages and submission to aggregators.

There is a chapter on consuming feeds and displaying them on your own web site and another chapter assessing the qualities of some RSS readers.

Hammersley writes in a clear style that readers of UK broadsheet newspapers would recognise. The programming examples are given in Perl, but I wonder if this was the best language for the task. Most examples are short routines to help parse RSS or form RSS from other sources, but one example runs to 13 pages of Perl program – a poor use of page space.

There are a number of minor editorial errors – a heading not agreeing with the body of chapter and a couple of references to a non-existent appendix of country codes. There is a thorough index and four pages of web site references. This book is a useful introduction to the subject, but probably not one you will keep to hand for long.

Small Business Websites that Work by Sean McManus (0 27365 486 1), Prentice Hall, 240pp @ £19-99 (2.38)
reviewed by Christopher Hill

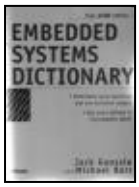
So you've got a domain and in your first flush of enthusiasm have a page or two on your business web site. So now what do you do – design it yourself? Hire a designer? Either way what do you want to achieve with your site? What are the bear traps that stop your site from effectively promoting your business?

McManus' journalistic style makes this an easy and enjoyable book to read. It is written for small business people who are new to this area. Yet as an experienced web designer I have found it a very useful resource.

McManus covers the whole gambit – from why do you want a web site; choosing domain name; offering great customer service online; keeping your site legal and ethical and finally measuring your website's success.

You will read many books that will tell you how to do each aspect of HTML, advertising, selling online, etc. but this is one of the few books that will tell you why you should prefer one route over another and what is wrong with those other routes. For example getting your friend to design your site may well be cheap, but will they understand your business needs, or use it as a vehicle to display how clever they can be with an obscure 6Mb plug-in?

A jargon-free down-to-earth review of the process of making the most of your business web site, packed with sound practical advice, written in a way that is accessible to a very wide readership. Highly recommended.



Embedded Systems Dictionary by Jack Ganssle and Michael Barr (1-57820-120-9), CMP, 286pp @ £25-99 (1.34)
reviewed by Francis Glassborow

Note that this is a dictionary, and that is exactly what it is. It covers words, phrases and acronyms that might be met in the context of embedded systems, defining them in that context or related ones. For example looking up 'yield' we find:

> yield 1. v. To offer use of the CPU to another task that is ready to run.
> 2. n. The percentage of good die on a wafer. IC vendors test each die, with the sure knowledge that some percentage of them will not pass. Mature manufacturing processes with smaller chips give high yields; very dense parts or those made with new processes could result in yields less than 50%.

As far as I can see the book gives a very good coverage of the area with good definitions. The pronunciations are, of course, American but I can live with that. I can envisage this book being exactly what the newcomer to embedded systems needs to avoid irritating his/her colleagues with requests for explanations. When a team leader tells you that you need some JK flip-flops it is probably not a good career move to ask which clothing shop stocks that brand.

The real expert will have little use for this book, but I would recommend it to others that have to talk with such experts. Like all dictionaries it will spend much time sitting on your shelf but it is comforting to have it there to grab when you need an explanation of some piece of 'jargon'.

Even Grues Get Full by J.D."Illiad" Frazer (0-596-00566-0), O'Reilly, 122pp @ £8-95 (1.45)
reviewed by Francis Glassborow

Sometimes O'Reilly lets its hair down and publishes a book just for enjoyment. This is the fourth collection of Illiad's cartoons that they have published. I think you need to know some computer jargon to appreciate all the humour but if you are reading this then you should qualify in that department. So the two remaining questions are whether you find his cartoons amusing or even funny and whether you want them in book form. If the answer is yes to both those, then this is the book for you. (Might be a suitable Christmas present)

If you have never come across Illiad's work (shame on you) try looking at

`http://ars.userfriendly.org/cartoons/`

Read Me First! by various (0-13-142899-3), Prentice Hall, 355pp @ £27-99 (1.25)
reviewed by Francis Glassborow

As titles go this must be one of the least descriptive. The subtitle is a bit more helpful 'A Style Guide for the Computer Industry'. But that could easily lead you to think that it might be something about software/source code style. You would be completely mistaken.

This book is about writing, ordinary technical writing. In this context the injunction of the title is right on the mark. If you have never written before (or even if you have, but are yet to get glowing tributes about the clarity of your writing) this is a book that would be worth investing some time with before you write your next article, manual or book. Mind you, I am not happy with all the advice and examples in the book. For example the authors sometimes get so focused on one aspect of a piece of advice that they miss some other aspect. For example on page 10 concerning pronouns:

- Do not use first person pronouns

  Incorrect:

  We recommend that you install the custom components only on large systems

  Correct:

  Install the custom components only on large systems.

Now I do not know about you, but the correct version would seem to be:

> *Only install the custom components on large systems.*

The book covers all forms of writing including writing for the Web.

While I am happy to recommend this book as being better than nothing I think that the book (now in its second edition) would benefit from having a genuinely fluent writer of English involved in its next edition. Much of the language is unnecessarily stilted and thereby gets between the reader and the content. In other words the content is fine but the book fails to teach by example.

The PhotoShop Elements Book for Digital Photographers by Scott Kelby (0-7357-1392-8), New Riders*, 255pp @ £23-50 (1.28)
reviewed by Francis Glassborow

A couple of months ago I was in Blackwells nattering with a couple of the staff about books on applications. I commented that I was surprised by the number of books on Adobe's Photoshop because that product is both very expensive and definitely aimed at the professional. I was told that actually the books sold quite well. That still surprises me because I would have thought that books on Jasc's Paint Shop Pro or on Photoshop Elements (the cut back version of Photoshop for non-professional users) would have been more appropriate to the mass market.

A couple of weeks ago this book arrived on my desk and I am very happy to tell you about it as it is exactly the kind of book to give to relatives or friends who are dabbling with digital photography. Of course they will have to have a copy of the software but it is one of two packages that I recommend for managing digital photographs and it comes at a reasonable price for the serious amateur. Using a digital camera without good software to manipulate the results is to miss the main advantage of using this mechanism over traditional film.

The book covers most areas that should interest the newcomer to digital image editing in a way that makes progress relatively easy even for the person who has only basic computing skills.

Unclassified O'Reilly Books

Linux Server Hacks by Rob Flickenger (0-596-00461-3), O'Reilly, 221pp @ £17-50 (1.43)
reviewed by David Ross

The blurb on the back cover makes it clear that the 'Hacks' in the title refers to the traditional meaning and has no relevance to script kiddies and their ilk. More subtle is the fact that this book does not cover the desktop at all. Instead, it covers an eclectic mix of topics ranging from networking, services such as BIND and Apache, and general administration issues such as backup, rsync, SSH, etc. The focus is on advanced administration of a Linux system.

The book itself is divided in to 100 short 'Hacks' (reminding me of O'Reilly's UNIX Power Tools book, albeit a much more portable version at 210 pages). This gives the book its main advantage – it's great to dip in to and there's always something of interest. Indeed, I'm already rereading some of the tips to implement the ideas on my systems. The downside is that topics cannot be covered in great depth but pointers to 'further reading' are provided and of course, some things simply won't be of interest to all readers.

That said, I really liked the book and would be happy to recommend it to Linux users. Anyone new to administration should look somewhere else first though and then come back to this book once they understand the basics.

Windows XP Hacks by Preston Gralla (0-596-00511-3), O'Reilly, 392pp @ £17-50 (1.43)

Amazon Hacks by Paul Bausch (0-596-00542 3), O'Reilly, 281pp @ £17-50 (1.43)

eBay Hacks by David A. Karp (0-596-00564-4), O'Reilly, 331pp @ £17-50 (1.43)

'reviewed' by Francis Glassborow

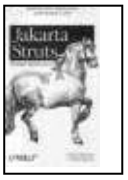
I am not going to review these books in any detail because they arrived on my desk as a result of O'Reilly's policy of sending me everything they publish and leaving it to me to decide what to review. (At least that way I do not miss anything important).

These three books share the qualifying subtitle '100 Industrial Strength Tips and Tools'. Does anyone else feel that there is something odd about a book on Windows XP sharing such a qualification with Amazon and eBay? My feelings is that the authors of the latter two books were really stretching to meet the series qualification, whilst the author of the first must have been badly pushed to decide what to leave out.

I think that the use of the 'Hack' in the title is significant. Each book assumes a considerable level of computer expertise from the reader and I would not want to hand these books to an enthusiastic novice who would promptly get their system into a tangled mess.

I have to confess that I am far from happy at the steady conversion of Amazon into the default merchant on the Internet. I remain of the opinion that in the long run dominance of this aspect of the Internet by a single company is as bad for users as is the dominance of the desktop by Microsoft. We need competition. Look how much better things became when Intel got a serious competitor in AMD and how much harder Microsoft works now that Linux is beginning to look like a product that can be used by the ordinary, non-technical, user. Of course this has little to do with 'Amazon Hacks'.

Perhaps another shared property of these three books is that each is about a product that wants to rule its corner of the IT world. Think about it. In the meantime if you really use Amazon and eBay a lot then the books on these are possibly worth the price. In the case of The book on XP, I have nothing against it as such but I would first buy 'Windows XP Annoyances' from the same publisher.



Jakarta Struts Pocket Reference by Chuck Cavaness and Brian Keeton (0-596-00519-9), O'Reilly, 133pp @ £8.95 (1.45)

Extreme Programming Pocket Guide by chromatic (0-596-00485-0), O'Reilly, 90pp @ £8-95 (1.45)

Oracle PL/SQL Language Pocket Reference 2ed by S Feuerstein et al. (0-596-00472-9), O'Reilly, 121pp @ £8-95 (1.45)

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

CVS Pocket Reference by Gregor N. Purdy (0-596-00567-9), O'Reilly, 84pp @ £8-95 (1.45)

Regular Expression Pocket Reference by Tony Stubblebine (D-596-00415-X), O'Reilly, 93pp @ £8-95(1.45)

'reviewed' by Francis Glassborow

Another bundle of Pocket Guides and References from O'Reilly. Once again I have similar reservations about the first three. If you need a reference to any of these subjects a 'pocket' one hardly seems adequate. In the case of 'Extreme Programming' I have to say that I think its only value is to participants in Trivia quizzes. Extreme Programming is an Agile Methodology that you should understand in far more detail and if you do you would have no use for a pocket reference. I would welcome alternative perspectives on the first three books.

The next three seem different to me because in each case their subject matter hardly justifies more than a book of this size. If you need to know anything about RTF (Rich Text Format) I would think that this book will provide it. While Regular Expressions are important and turn up in many places a book this size is just about right.

If you are involved in open source or other forms of distributed development the CVS (Concurrent Version System) tool or something like it is just about essential. This small book is exactly what the user needs.

Note that in the last three cases it is entirely incidental that the book's size and format fits into an anorak pocket. The important thing is that the amount of material does not justify a larger book.

My final comment is that I have a general moan about almost all O'Reilly books; they do not like being open and make every effort to close unless manually restrained. Reference books should not do that, we almost always want our hands for other things when we are using them.

Needs a Reviewer

Packaged Composite Applications by Dan Woods (0-596-00552-0), O'Reilly, 186pp @ £20-95 (1.43)
non-review by Francis Glassborow

I am not intending to review this book, but O'Reilly have sent me a review copy. If I just added it to the books for review list I doubt that many would know what it is about. Here is an extract from the back cover:

> Packaged Composite Applications (PCAs) represent a new architectural paradigm for enterprise applications. Using web services, they combine new functionality with services from existing applications to enable flexible cross-functional automation. "PCAs are applications that result from the marriage of IT, business process, and business strategy," says Yvonne Genovese, Vice President and Research Director of Gartner, Inc.

Fortunately the text is not as densely jargon/buzz phrase laden. If you feel that this is a subject area where you have some knowledge and interest I would be happy to have you review this book.