Contents

Copy Dates

C Vu 15.3: May 7th

C Vu 15.4: July 7th

!!!! PRIZES for journal articles - see page 26 for details !!!!

Contact Information:

Editorial: James Dennett
76 Lawn Road,
Bristol, BS16 5BB
0117 9653875
editor@accu.org

Advertising: Pete Goodliffe
Chris Lowe
ads@accu.org

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

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

Secretary: Alan Bellingham
020 8998 6964
secretary@accu.org

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

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

Membership fees and how to join:

Basic (C Vu only): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21

There are 6 journals of each type produced every year.

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

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

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

Reports & Opinions

Editorial

This being the April issue of C Vu, some of you will be reading it at ACCU's Spring Conference. If you are at the conference, use your time wisely: have fun, talk with your peers, attend some of the most educational sessions to be found, and do not spend too much time reading the journals. The chance to meet with so many motivated developers is rare, and this magazine will wait patiently until you have a quiet moment.

I will have to offer my apologies for absence from this year's conference. My reason is a good one. This being a publication about programming, I will say only that the reason is 5' tall and well worth moving my life halfway around the world for. By the next Spring Conference I might be able to spend a week away from her, or make a case that a vacation in the UK is too good to miss.

Writing for Fame, Fortune and Prizes

Look for the piece later in this issue (page 26) for some information on new annual prizes to be awarded for authors of articles published in C Vu and Overload. For the best piece written by a new author, I will add the incentive of a year's ACCU membership.

SCO's Desperate Gambit?

Open Source cannot provide enterprise-strength solutions, or so claim SCO, a UNIX vendor whose fortunes in recent years have suffered from competition from Linux.

It may seem that SCO skirt with contradiction when they say this – after all, if Open Source software is not so good then their own allegedly superior, commercial-strength offerings should take more of the market – but SCO have decided that Linux has only become this good because IBM broke agreements with SCO and gave Linux the features it needed.

SCO may have fewer lawyers than Big Blue, but they have more than ACCU, so I shall be careful what I say. I would not want to be in court on charges of having unlawfully acquired the ability to express opinions in an editorial without legitimate commercial backing.

Those who wish to read the complaint in SCO's own terms can see it online at `http://www.sco.com/scosource/complaint3.06.03.html`

Focus on the technical issues. (SCO's incorrect use of "whom" may infuriate me, but that's not the real problem.) The real problem is the lack of plausibility in SCO's claims.

Take two examples:

- SCO suggest that Linux only scales to 4 processors, while UNIX runs on up to 32. Maybe they have not read any benchmarks showing Linux running on larger systems? (For any of you who have won a lottery recently, SGI's Linux machines such as `http://www.sgi.com/servers/altix/index.html` will let you run 64-bit Linux on 64 processors with 512GB of RAM. And they come in curvy boxes.)
- SCO claim that no other major UNIX vendor developed a UNIX for the Intel (presumably meaning x86) chipset. And yet they mention Sun as a vendor. So, can SCO really not know of Solaris x86? If so, they're not competent to talk about Unix on x86 at all and should apologize and switch to another line of business. If they really want to talk about Intel support, I refer them again to SGI's support for the Itanium II. Did I mention that SGI's machines come in curvy boxes?

It stretches credibility painfully to imagine that SCO can really be so ignorance of their x86-based competition. At least, if they know this little of what does exist in the marketplace, it is hard to accept that they are authoritative on what Linux can do either now or in the past.

For another angle, `http://mq.moo.net/Linux03/ScoSource-05_Story01.html` is Linus Torvalds holding forth on the subject.

That SCO maybe be using this feeble lawsuit to attempt to be bought out either by IBM (to shut them up) or Microsoft (to step up the battle) is a viewpoint widely seen online. In any case, I hope that IBM will treat this with the contempt it deserves and see that SCO pay for IBM's hefty legal costs. It would also be refreshing if the legal system shows that it will not tolerate this kind of abuse, by fining SCO heavily for acting in bad faith.

If you care about freedom (in computing or in general) then look into this a little, make your own mind up, and consider it if you have dealings with SCO. Remember: SCO say that Linux is taking their business. Maybe there are reasons for that.

More QA, More Quality

Much though I would love to recommend "Zen and the Art of Motorcycle Maintenance" to all C Vu readers, let me write with more focus on Software Quality Assurance. It's a magical phrase, suggesting as it does that a group separate from programmers can somehow inject Quality into a product. There is no need to take more than a sentence here to remind us that QA is only effective when it is woven right through the fabric of our development processes.

All but the worst software engineering groups recognize that Quality Assurance is an important aspect of delivering good software. There is considerable variation in how much this recognition translates into action. Many small companies do not have a separate QA role as such; they do some testing, but that is as far as it goes. Often most of the testing is done by developers – which can work, but relies not only on them being professional enough to try to do adequate testing but also on them being able to find their own errors. Finding our own errors is harder than finding those of others, for a variety of reasons, and the record of our industry is not good.

Figures I can find online in articles claiming that programmers are starving because of software piracy estimate the annual cost of software piracy worldwide at US$11-12 billion [1,2,3], and there is plenty of reason to suspect that they overstate the amount of revenue actually lost quite heavily. Losses to business caused by software defects: close to US$60 billion [4]. And don't think the problem is just with one player; security alerts for Linux based defects are on the increase now as Linux is more widely deployed and attacked. The free software community will need to address security more actively still if it is to continue to be seen to be more secure than mainstream commercial software.

ACCU members being ethical programmers, the problems caused by our industry should concern us all. Producing the best products we can within the constraints imposed by business realities is necessary, but not sufficient. We also need to do what we can to change those constraints. At present, commercial and legal realities mean that companies are rewarded for shipping software they can call "good enough". Good enough to avoid legal liability, possibly. Let us suppose that there are as many as ten million software developers worldwide [5]. Then each of us is, on average, responsible for software defects leading to a loss to business of $6000 per year. That's not good enough. It will not get good enough until software companies are forced to accept responsibility for the quality of their work. Shrink-wrap and click-wrap agreements that strip away normal consumer protection are not ethical and should not be legal. Software should be fit for the purpose for which it is sold. A license to run a piece of software should not be tied to a particular piece of hardware — hardware becomes obsolete more quickly than software. Bug fixes should not come with licenses allowing the software vendor to take complete control of the machine on which you install them. Take a look at some of the license agreements you routinely ignore, and then look at what you can do to add your voice to those who are trying to move the law in the right direction. Even if you're not in the US, look at the UCITA legislation which aims to move in the wrong direction, providing legal protection to companies seeking to avoid responsibility for their own products, and look at `http://linuxtoday.com/news_story.php3?ltsn=2000-02-06-001-05-NW-LF` for Richard Stallman's take on it.

My final word on Quality, for now: read Robert Pirsig's book (ISBN 0553277472).

Footnotes

[1] From the Business Software Alliance, `http://www.bsa.org/usa/press/newsreleases/2002-06-10.1142.phtml?type=policy`

[2] Figures from Microsoft for 1999, `http://www.microsoft.com/romania/antipiraterie/mondial/globalpiracy.htm`

[3] `http://asia.internet.com/asia-news/article/0,,161_1347951,00.html`

[4] Figures from the US National Institute of Standards and Technology (NIST), `http://www.nist.gov/public_affairs/releases/n02-10.htm`
[5] Statistic made up on the spot, I cannot deny.

From the Chair

Alan Griffiths <chair@accu.org>

This will be last “From the Chair” and it seems like a good opportunity for reflecting on where ACCU is going. It is my belief that the age-old imperative “grow or die” applies and that what we need to seek is growth. What form that growth might take is up to all of us, but there are some lessons from the past.

When I first became involved with the “C Users Group (UK)” (as ACCU was then known) I was one of a few mavericks that were interested in an obscure C based dialect called “C++” – and that interest was largely because of the cross-platform promises made by the “CommonView” class library. Well, CommonView probably predates many of you, but the lesson that I want to draw is that this was accepted as a valid area of interest – and not outside the area of interest of the C Users Group (UK). The area of interest grew to such an extent that C++ is now the lingua-franca of the organisation.

The areas of interest accepted within ACCU keep expanding: my current responsibilities at work have almost no connection with C or C++ (very occasionally I get involved in resolving problems with JNI – an interface between our Java and C++ components). They have little to do with any specific programming language: they are to do with development processes and system design. But there doesn’t seem to be any reason to seek out another organisation to discuss them: ACCU members are just as interested in change control strategies, Extreme Programming and testing as they are in the C programming language.

In the journals and mailing lists there is a very healthy diversity of subject matter covered: design, version control, C++, XML, SQL, Java, Python, interview (and interviewing) techniques, C, and probably more that don’t spring to mind immediately. This reflects the range of knowledge that the professional software developer comes into contact with during the course of his or her career.

I take the fact that the membership of ACCU has been rising slowly during my term as an indication that we have been doing something right. And the strategy has been simple: if there are members willing to do something that sounds reasonable then give them the authority to do it. (Actually, I apply the same strategy in software development teams too.)

What happens next will not be up to me, it will be up to you. If there is something that you think the ACCU should be doing, then do something about it. There is at least one committee post for which I don’t know of a potential candidate, and there are always opportunities for anyone willing to make a contribution.

A final point: although the AGM happens at the conference it isn’t part of the conference and one doesn’t need to be at the conference to attend the AGM.

Membership Report

David Hodge <membership@accu.org>

Membership stands at 1059. If you would like to set up a standing order to make the renewal of your subscription easier then just send me a request by email. We have just sorted out the renewal of our registration under the Data Protection Act. If you are interested, go to `http://www.dpr.gov.uk`, click on Search Register, put ACCU in the name box and click Search. You will be offered a choice of two names, one of which is us.

From the Conference Chair

Francis Glassborow <francis.glassborow@ntlworld.com>

By the time you read this I will have finished one more job for ACCU. The current conference is the seventh whose programme I have organised. The first was the last time that WG21 met in the UK so it seems fitting that it is meeting here again just after this conference.

I believe that the conferences have been an outstanding success for all those who have come to them, whether as speakers or as listeners. Others tell me that it is one of the very best technical conferences in the World. My only sadness is that its attendance has remained in the low hundreds. Those that come find a vibrant environment that inspires them and helps them feel part of the programming community; I just wish there were two or three times as many. I have seen commercial conferences with much weaker programmes and at three or four times the cost get twice the attendance. Anyway, that will not be my problem in future years though I will happily give my successor(s) all the support that I can. I hope that the event will continue many years into the future and that more people will get the message and join in the programming community and find that ACCU events provide an enjoyable way to enhance their professional skills or deepen their grasp of a hobby.

Sponsorship of Standards Meetings

One of the features of this year’s conference is the running of the WG14 (C) and WG21 & J16 (C++) meetings at the same venue and fortnight as the ACCU Spring Conference. If you do not already know, many participants in these meetings have to pay their own way. I am therefore very grateful to those companies that have provided money to cover the costs of hiring conference facilities and providing refreshments for those meetings. While the individual sums of money are relatively small for companies with multi-million pound revenues, that is to miss the point. The following companies made the effort to provide support and ACCU is grateful to them:

Microsoft (who picked up the entire costs of the conference facilities for WG21 & J16)
LDRA `http://www.ldra.co.uk/`
Intel `http://www.intel.com/`
Hitex `http://www.hitex.co.uk/`

If you do not already know about the development tools provided by the last three companies please take the time to visit the listed URL and find out.

Standards Report

Lois Goldthwaite <standards@accu.org>

The time is drawing near when you can have your very own copy of the C or C++ Standard, handy for reading on the train or in the bath. The project for John Wiley Co. to publish both documents is steaming ahead, and the books should be at a bookstore near you well before the end of the year. We are hoping to be able to announce a firmer publication date at the ACCU conference.

ACCU, which already supports language standardisation efforts with your generous contributions to its International Standards Development Fund, is now providing assistance of a different kind. An ACCU site will soon be operating the email reflectors for several BSI panels, including those for internationalisation, Ada(!), and Fortran(!!), as well as that of IST/5, which is the parent committee of the panels, corresponding to SC22 in the ISO world. ACCU already hosts the C panel reflector. These mailing lists were previously operated by a university computer in Edinburgh, and managed by the convenor of the Fortran panel, who is retiring from the university.

The C++ standards panel is planning to organise a Birds-of-a-Feather session one evening during the ACCU conference, to encourage C++ programmers to find out more about standards activities and how to get involved. It will probably take the form of a short version of one of our regular panel meetings, which centre around technical discussions of current issues in the C++ world, followed by more technical discussions centred around a table at the nearest pub. Watch for an announcement at the conference with more details.

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.

Late Entry to SCC 19

Before providing the entries for the current competition, here is one that arrived too late for the last one (note that competition material goes up on `www.accu.org` at the time an issue of C Vu is distributed so that those who have to wait for their C Vu because of slow post can still compete.)

Unfortunately the entry was not only late but the writer forgot to include their name in the file. Please help me give you credit by remembering to include your name in all files you send as attachments.

From Anon

Since this is my first time to reply to a code critique, I hope I won't embarrass myself too badly with any blunders. I'd also just like to say how much I enjoy getting to learn about C++ by reading C Vu and lurking in the mailing lists. The comments below were heavily influenced by my reading of several of my reference books, including *Accelerated C++* by Koenig and Moo, and Steve Myers' *Effective C++* and *More Effective C++*.

First off, the design of the class has several flaws. In class `Card`, `Display()` is defined as a pure virtual function, thus making `Card` an abstract class that cannot be instantiated, a clear error. Also, why is the destructor defined as virtual? From these two functions, it seems that `Card` is meant to be an abstract base class with other classes derived from it. But why? It seems that a class representing a physical card should not need derived classes from it, but since I am still in the beginning stages of learning C++ and haven't learned much yet about class hierarchy design yet, I could be mistaken. If `Card` is not going to be derived from, we do not need any virtual functions. I am thinking that the `Display()` function would be used at a later date to draw the actual card, and the implementation of it isn't needed yet. Also, if we don't need a virtual destructor, we can leave that out of our class, since the compiler will auto-generate a non-virtual default destructor anyway.

Next are some style concerns in the code. First off, I noticed that `RegularPack` is a global variable, which in general is not recommended. In my example, I would move this variable into `main()`, making it a local variable. Second, the main code has two loops that are both meant to index across an array. But two different end conditions are used. One style should be used, and then adhered to make the code more consistent. The second loop doesn't even use `RegularPack` to identify the number of times the loop executes, just a so-called "magic number" which again is not recommended. The first loop also uses the variable "`i`", which is defined in `main()`. Common C++ practice it to define a loop index variable in the `for` header, so that it goes out of scope and is destroyed after the loop is finished. Third, the code uses both "`\n`" and `std::endl` to insert a new line in the output stream. A fourth inconsistency is that the naming convention used is not consistent in that at least one variable (`pCard`) uses Hungarian notation, while all other variables do not. For both of these, I think the best thing to do is use one form, and be consistent. The last, and biggest style issue in my opinion is that an array of pointers is being used to represent a deck of cards. It seems that using `vector<Card>` from the standard C++ library would be perfect to hold a number of cards. This allows us to use the generic sorting functions in the standard library in the future if needed, as well as not worrying about all the problems pointers can cause (especially for us beginners).

Now for the compilation errors. I tried compiling the code with Visual C++ .NET, as well as Comeau's online compiler. I added appropriate `include` and `using` statements, which were not in the original code posting. I am guessing that they were left out for brevity, but if not then certainly this would be an error as well. The first thing that I noticed was the typo in the first loop, where `RegularDeck` should be `RegularPack`. Also, I got two warnings, one that `pCard` was never used, and the second was that "label '`Card:`' was declared but never referenced". The second warning is actually a syntax error in that the "`Card:`" shouldn't be there. The next error was the last line of the program, where no variable `Card` was defined. The last error was for the body of the second loop. To be honest, after the first loop, which just gives each `Card` in the array a value from zero to 53, I don't understand what the code is supposed to do. The second loop (I think) is meant to index through the array, and display each `Card`'s value. But the second line in the loop body is all wrong, since the pointer shouldn't be deferenced, and even when you fix that, `SetNumber` returns `void`, which is an illegal type to pass to `cin`.

I've rewritten the class `Card` so that it no longer has a `Display()` function, which wasn't used anyway, as well as leaving out a destructor, and letting the compiler generate one when needed. I've added appropriate `include` and `using` statements. In `main()`, I've used a `vector<Card>` to represent the deck, and use the standard library functions to insert and view the values of the deck. The program gives each of the 54 cards a value, from zero to 53, then prints out each card and its value. I'm sure there is a better way to re-write the code, but as I am still learning, I'm just trying to keep it simple, and (hopefully) correct.

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

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

int main() {
  const int RegularPack = 54;
  vector<Card> Pack;
  for (int i=0; i < RegularPack; ++i)
    Pack.push_back(i);
  for (vector<Card>::size_type i = 0;
       i != Pack.size(); ++i)
    cout << "Card number " << i
         << " value is "
         << Pack[i].GetNumber() << endl;
}
```

Student Code Critique 20

The Code

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

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

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

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

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

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

From Catriona O'Connell <catriona38@hotmail.com>

Major errors in the student's code:

While space is allocated for grades, the marks are actually read into `test`, which is an uninitialised pointer – leading to overwriting of storage. Pointers should be initialised before they are dereferenced.

The function `allocatemem()` is declared to `return int*`, but the definition returns `int`.

Nothing is done with the return code from `allocatemem` except to print a message. The code continues regardless of the success or failure of the allocation request. Actions need to be taken on return codes. As written, the code in `allocatemem` will not behave as the student expects if it is compiled with a conformant compiler. If `new` fails to allocate storage, it is required to throw a `std::bad_alloc` exception (3.7.3.1/3). To return a null pointer instead of an exception the user should call the `nothrow` variant of `new`. The `<new>` header must be included. At this point it is worth noting that the program does not free the allocated storage on exit. It is good practice to free-up dynamically acquired storage. Not all environments will clean-up after the program terminates. I'm thinking of some subpools in the MVS OS here.

In the function `sortnums()`, there is no good reason to declare `temp` in the `for()` and even if you did it should have been separated from `int j=0` by a comma not a semicolon. The `for()` loop as written has four statements instead of three. The `sortnums` function doesn't sort the numbers either and there is a typo where `i` has been written as `1`.

The argument is `int scores` which is never used and the loop goes over `numberOfScores` which is not defined anywhere – and even worse has two different spellings. The function needs an argument that defines the number of elements in the array. It is also defined to return an integer, but declared to return a pointer to an integer. The whole function is a mess and should be re-written.

The function `main()` should have been declared as `int main()` not `void main(void)`. The C++ standard allows only `int main()` and `int main(int argc, char *argv[])` as definitions (3.6.1/2).

Other comments on the student's code:

There is no checking of user input. As a general rule input from a user should always be checked for validity. The coding of `cin` to retrieve values from the user is too trusting. If a user enters a non-integer value then the stream's fail bit will be set and the `cin` object becomes unusable. One solution is to call the `clear` member to reset the fail bit. This should be followed by the `ignore` member to discard any additional input from the stream.

Another minor flaw with the grade input code is that the student tampers with the loop control variable to handle incorrect input. While tampering with a loop control variable is not explicitly forbidden by the standard it is a sign of a poor program design. While not true in this case, there is a danger of such practices breaking the conditional test in the `for()` header.

The variable `NoGrades` is poorly named as it might be taken to mean that the student has no grades rather than holding the number of grades. Good naming standards are helpful to code maintainers.

The student has enclosed string literals in single quotes rather than double quotes in his/her dialogue with the user. Also as a matter of style, if `cin` and `cout` are being used it would be more consistent to use `std::endl` rather than '`\n`' as a newline[1].

It is also not beneficial to use the pointer notation for the `test` array, when the array[subscript] format would have been clearer. There is an equivalence between `*(test +i)` and `test[i]`, but the latter is easier to visualise – especially if you are a Fortran programmer :-)

Comments on my alternative code:

The pointer returned from `allocatemem` is used to determine if the code proceeds with data processing or drops through to the final return. This removes the need to have multiple return points.

The `getInput()` function addresses the problems raised by the student's use of `cin` to retrieve input from the user. The `getInput` function requires that a valid range of input values be passed as arguments so that the user is required to enter a valid data type within the specified range. The `getInput()` function is a specialisation of the template code presented in response to SCC12.

I have snaffled a `bubblesort()` routine from `http://leepoint.net/notes/cpp/algorithms/arrayfuncs/bubblesort2.html`[2]. The bubble sort routine is adequate for small amounts of input and is simple to understand and implement. The bubblesort routine sorts the array in place. The student did not specify what we were to do with the output, so I printed it just to demonstrate that the sort worked.

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

int *allocateMem(int);
void bubblesort(int *, int);
void getInput(const char *, int&, int, int);

int main(){
  int numGrades;
  int *grades;
  int i;
  getInput("Number of grades? ", numGrades, 1, 10);
  grades = allocateMem(numGrades);
  if(grades){
    for(i=0; i<numGrades; ++i){
      getInput("Enter Grade", grades[i], 0, 100);
    }
    bubblesort(grades, numGrades);
    for (i=0; i<numGrades; ++i){
      cout << grades[i] << endl;
    }
    delete [] grades;
  }
  return 0;
}
```

1 Actually many experts would not agree. `std::endl` should only be used where you require the output be flushed in addition to including a newline.

2 And I suppressed it because this is not one of the rare places where a bubblesort offers any advantage, and in the context of this problem the student should simply use `std::sort`.

```
int *allocateMem(int amount){
  int *memory;
  memory = new(nothrow) int[amount];
  if(!memory){
    cout << "Error: Unable to allocate sufficient "
         << "memory." << endl;
    cout << "Program terminating." << endl;
  }
  return memory;
}

void bubblesort(int *x, int n){
// code suppressed use std::sort instead – Francis
}

void getInput(const char *prompt, int& r, int lower,
int upper){
  bool error;
  do{
    cout << prompt << ": " << flush;
    error = !((cin >> r) && (r >= lower)
                         && (r <= upper));
    if(error){
      cout << "Please try again." << endl;
      cout << "Enter an integer between " << lower
           << " and " << upper << endl;
    }
    cin.clear();
    cin.ignore(INT_MAX,'\n');
  } while(error);
}
```

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

Dear student,

First the good news – most of your difficulty in solving the problem doesn't seem to be with the concepts. The bad news is you seem to have several sorts of problems, mostly with the syntax – that is the details of writing C++ code.

Let's deal with this systematically.

Firstly, the problem you were set was to write a function to allocate memory and return it, or null on failure. Your allocatemem function almost does this (apart from a small syntax error). However you have allocated the memory with new and in standard C++ this does not return at all if it fails but throws an exception. However you were asked to check for out of memory and return null. Perhaps the teacher was asking the wrong question? I'll leave it to you to discuss that with them!

So you need to use a different 'flavour' of new to get back a null pointer if there isn't enough memory. This is done by adding an extra argument to new and writing 'new(nothrow)' instead.

That solves the main conceptual error you seem to have with the problem set you. The second error is you forgot to check the return value when you call allocatemem in main. If it failed then you'd better stop there!

Syntax, syntax, syntax.

You have a few syntax problems – perhaps you were in a hurry? "More haste less speed" when it comes to syntax errors I'm afraid.

Firstly, C++ is fussy about the difference between a single and a double quote: ' and "; some other languages don't care. A single quote is for a single character, a double quote is for a string of characters. As it happens all your single quotes can be changed to double quotes.

Secondly, C++ is fussy about referring to variables in other functions. So sortNums tries to use the variable test from main but it can't see it – you need to pass this variable in as another parameter to the function.

Incidentally, sortNums is perhaps not the best name for the function since it does two jobs – it sorts the numbers but then prints them. I'd prefer writing two functions, sortNums and printNums, and so keeping code that changes the scores separate from code that prints them. You can then make the array constant in the printNums function that among other things prevents you accidentally writing code which changes it.

Thirdly, you aren't consistent and, alas, the compiler expects you to be completely consistent. It is also easier for you and everybody else if you use consistent names for things. For example, the number of grades in the program is called many names: NoGrades, grades, numberOfScores and numberOFScores in various parts of your code. You also have grades and test both referring to the same data. Try to ensure that the names of variables are always the same – and this includes being in the same case. To make this consistency easier it is a good idea to adopt a simple policy on names – many people use a mixed case with first letter lower case like you have in numberOfScores. I've picked on one name and used it consistently in the program.

Another place where you must be consistent is that the types used for function returns and arguments must be the same in the function prototype and where the function is actually defined. The prototype for sortNums returns int* but the function itself claims to return int.

Lastly you have a small problem with the for-loop in sortNums – you can initialise multiple variables in the for-loop initialiser but if so you must separate them with commas. In your case I'd simple remove int temp=0; completely and put the int on the next line to declare and define temp in one go.

All being well you now have a body of code that compiles and does solve the problem you were set.

However we are not finished yet...compiling is not at all the same thing as working! It would be nice if your sample program worked and used standard C++.

Firstly, main is defined as void which many compiler still accept but it is an old usage. In standard C++ main always returns an int. You shouldn't need to put in a return-statement in main, the compiler ought to do this for you, but not all compilers do so put a return 0; just before the closing brace to be safe.

Secondly, sortNums is defined to return an int but doesn't return anything – and its return value is ignored where it is used! So I'd change sortNums to return void.

Thirdly, you have a very, very small loop. The for-loop in main has a spurious ';' which the compiler understands to mean 'do nothing' each time round the loop. Just remove it and then the loop will work as you expect.

And then lastly printNums doesn't print and sortNums doesn't sort! The for-loop in printNums uses 'test + 1' inside the loop where you wanted 'test + a'.

The sort is just broken but it is easily fixed. The solution here is to stop re-inventing the wheel. The problem didn't ask you to write a sort algorithm, so don't. Just use the standard library sort from the header <algorithm>.

The final code looks like this:-

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

// function prototypes
int *allocatemem(int);
void sortNums(int* grades,
              int numberOfGrades);
void printNums(const int* grades,
               int numberOfGrades);

int main(void) {
  int numberOfGrades;
  int *grades;
  cout << "Number of grades to enter: ";
  cin >> NoGrades;
  grades = allocatemem(numberOfGrades);
  if(grades == 0)return 1;
  for(int i=0; i<numberOfGrades; i++) {
    cout << "What is test grade # "
         << (i+1) << " ?";
    cin >> *(grades +i);
    if(*(grades +i) < 0 ) {
      cout << "Must be positive \n";
      cout << "Please enter Test #"
           << (i+1) << " correctly: \n";
      i–;
    }
  }
  sortNums(grades, numberOfGrades);
  printNums(grades, numberOfGrades);
  return 0;
}
```

```
int *allocatemem(int amount) {
  int *memory;
  memory = new(nothrow) int[amount];
  if(memory != 0 ) {
    cout << "We have memory \n";
    cout << "going back to main \n";
    return memory;
  }
  cout << "We do not have enough memory for "
          "this task \n";
  cout << "going back to main \n";
  return memory;
}

void sortNums(int* grades, int numberOfGrades) {
  std::sort(grades, grades+numberOfGrades);
}

void printNums(const int* grades, int
numberOfGrades) {
  for(int a = 0; a<numberOfGrades; a++)
    cout <<*(grades + a)<<' ';
}
```

From William Fishburne

<william.Fishburne@verizon.net>

The restrictions requested by the student strongly limit what can be said. I have constrained the analysis to the `allocatemem` function, as requested, but in a normal circumstance, it would be worthwhile to discuss the whole program!

A few overriding things need to be addressed, within the limitation of discussing this one function, these still apply:

1. `using namespace std;` is a bad practice[3]. It is better to specify `std::` before the items you need from the standard namespace. You probably got this from your text, but the author also probably makes some statement about how you shouldn't really use it and that he is doing so in the interest of clarity. Some clarity, huh? A large reason for a namespace is to prevent a name clash. While this little program does not really incorporate enough to generate such a clash, why would you risk it? Wouldn't it be better to get in the habit of using namespaces properly so that you never have to try and debug something where somebody got clever and used `cout` in their own namespace?
2. `main` always returns an `int`. Setting its return value to `void` is simply an error.
3. It strikes me as very odd that a C++ class would require you to write a function that dynamically allocates arrays. It is certainly possible, but it is counter-intuitive to the flexibility that the standard library offers. At the risk of pointing you in the wrong direction, consider a `multi-map` from the standard library that has built-in the ability to associate a score with a test and to sort the result.
4. Globals are dangerous. It is much better to pass what is needed to each function as the values are needed. This is true because a global variable can be changed anywhere in the program. The bigger the program, the more likely that any change will break something else.
5. It is a big mistake to change the value of a control variable in a `for`-loop. Is there another way you can make sure that a valid test grade is entered that wouldn't require you to decrement the control variable? (Francis, since you have talked about C++ standards before, wouldn't it be nice if a `for`-loop variable in a for loop was a constant *within* the loop? This would sort of be a reverse sense of scope, it could only be changed outside the loop block... Similarly, I think it makes sense `for`-loop variables to never be instantiated outside the loop statement…

Actually I strongly disagree and changing the rule would break a good deal of existing, well-designed, code.)

Now, because I respect the heck out of Francis (HEY! That is not butt-kissing, that is the honest truth…), I downloaded Quincy et al and typed this program into it. What do you know? The first error that came up was the problem with `main`.

Once I fixed my typos and a couple of obvious errors (I spare you as per your request), the next error that came up was the redefinition of `allocatemem`. As a prototype, you define it as `int *`, then when you define the function itself you use `int`. Certainly you must have meant the former as you plan to return a pointer to an integer, right? The compiler catches simple errors like this quite readily and it is worth working to make sure that your program compiles before seeking other help if you can. The listing below shows a version of your program that will compile (wait to look at it until you have managed to get it to compile yourself). If you attempt to run this program, you will find that it does not work (as you expected or you would not have submitted it). This is not unusual. Few programmers can write a program that both compiles and runs correctly on the first try (I've never met one).

OK, we have `allocatemem` returning a pointer to an array of integers. We accept in the number of elements in the array. You have correctly formulated the call to `new` and memory will be allocated to the pointer `memory` if it is available. What does `new` do, however, if the memory is not available? In the C++ standard, `new` is supposed to throw the error message `std::bad_alloc`. To catch this error message, you would need to encompass your call to `new` in a `try/catch` clause. In order to catch this error, you will have to include the `<new>` header at the top of your program.

There is a `nothrow` version of `new` which returns a `NULL` pointer. I think it is worth taking the time to get to know the `try/catch` clauses instead of using the `nothrow` version, but let us assume, for the moment, that you do decide to go ahead and use the `nothrow` version. Are `NULL` and `0` the same thing? If so, why have two different names? Consider, are `NULL` and `0.0` the same? I am not asking whether `NULL` and `0` **can** be the same, but whether they are **required** to be the same.

Finally, with regard to `allocatemem`, does your program pay attention to the return value from `allocatemem`? If not, why not? What will happen if memory isn't properly allocated?

Some questions to consider on the rest of the program:

1. What should `sortNums` return? Does it?
2. When printing an array, if you get the same value over and over, then you are either not incrementing across the array or you are not incrementing your array index.
3. What happens if you look one position past the end of an array? How can you make sure this doesn't happen?
4. How many times should a bubble sort iterate?
5. Why does C++ allow you to reference arrays as `grades[6]` instead of requiring you to write `*(grades+6)`? Is `grades[1]` always the same as `*(grades+1)`?
6. When is it appropriate to have two variables in a `for`-loop? Is your use appropriate?
7. What is the first value of `test`? Does `test` ever change? What are the implications for the first two values of `temp`?
8. How many integers do you pass to `sortNums`?
9. What happens when memory is allocated and never deleted? Where is the best place to delete the new memory you have created?

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

// function prototypes
int *allocatemem(int);
int sortNums(int);
// global variables
int *grades;
int *test;

int main() {
  int NoGrades;
  cout << "Number of grades to enter: ";
  cin >> NoGrades;
  grades = allocatemem(NoGrades);
  for(int i=0; i < NoGrades; i++) {
    cout << "What is test score # "
         << (i+1) << " ?";
    cin >> *(test+i);
    if(*(test+i) < 0) {
      cout << "Must be positive\n";
      cout << "Please enter Test #"
           << (i+1) << " correctly: \n";
      i–;
    }
  }
  sortNums(NoGrades);
  return(0);
}
```

3 Many would disagree with that assertion, particularly with regards to student code.

```
int *allocatemem (int amount) {
  int *memory;

  memory = new int[amount];
  if(memory != 0) {
    cout << "We have memory\n";
    cout << "going back to main\n";
    return memory;
  }
  cout << "We do not have enough memory for ";
  cout << "this task.\n Returning to main\n";
  return memory;
}

int sortNums(int numberOfScores) {
  for(int j=0, temp=0; j<numberOfScores; j++) {
    temp=*(test+1);
    if(*(test+1) < *(test + 1 + 1)) {
      *(test + 1) = *(test +j1 + 1);
      *(test + j + 1) = temp;
    }
  }
  for(int a=0; a<numberOfScores; a++)
    cout << *(test + 1) << ' ';
}
```

The Winner of SCC 20

The editor's choice is: **Roger Orr**.

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

Student Code Critique 21 (C source)

The original code for this program was in C++ but I have converted it to C because it was conceptually a C program using C++ for I/O. You will need to make an assumption about the input that is actually causing a problem – think of an input that has many digits. However there are several other immediate problems with the code as well as a general design error. Can you improve the design on the basis that the writer knows about for-loops but not about arrays?

The problem

I'm creating a program that inputs three integers, and returns the sum, product, average, largest and smallest. Very simple, but I'm getting a bad return on one of my variables. Except for my "largest" variable, the others are returning the correct numbers. I'm hoping that someone can tell me where I'm going wrong.

```
#include <stdio.h>

int main(){
  int num1, num2, num3, sum, average
  int product, smallest, largest;
  puts("Input three different integers: ");
  scanf("%d, %d, %d",&num1,&num2,&num3);
                          // read three integers
  sum = num1 + num2 + num3;
  average = ((num1 + num2 + num3)/ 3 );
  product = num1 * num2 * num3;
  if(num1 < num2 ) smallest = num1;
  if(num2 < num1 ) smallest = num2;
  if(num3 < smallest ) smallest = num3;
  if(num1 > num2 ) largest = num1;
  if(num2 > num1 ) largest = num2;
  if(num3 > largest ) largest = num3;
  printf("\nSum is %d",sum);
  printf("\nAverage is %d",average);
  printf("\nProduct is %d",product);
  printf("\nSmallest is %d",smallest);
  printf("\nLargest is %d\n",largest);
  return 0;
}
```

6 x 24

Paul Grenyer and Alan Bellingham

Recently, our company has been recruiting C++ programmers and, like any other forward-thinking organisation, we've taken great pains with the interview process, ensuring that we ask the candidates a carefully crafted set of questions to test their knowledge. Being as we are a company that believes in careful research, we did a quick check of a mailing list and borrowed the first set of questions we found there - after all, the combined powers of `accu.general` are likely to produce a superior set of questions to those we could produce ourselves. Since we also believe in proper testing, we first tried the questions out on our most long-serving programmer to get a truly definitive benchmark set of answers. We are rather pleased with the results and we happily anticipate finding the ideal candidate to work with the rest of our coding team. In the interests of improving the quality of all software teams everywhere, we herewith share our results:

1. Who is Bjarne Stroustrup?
"He's just this guy, you know"

2. On a scale of 0 to 10, where 0 is someone who has never heard of C++ and 10 is Bjarne Stroustrup, how would you rate your C++?
11 - did I mention I was once a roadie for Spinal Tap?

3. Write a simple Standard C++ program which displays "Hello World" at the command line and starts a new line. Describe how you would modify this program so that it could be built by Microsoft Visual C++ 6.0 without any warnings.

```
#include <Everything>
void main() {
  system("echo Hello World") ;
}
```

For VC++

```
#include <stdafx.h>
void main() {
  system("echo Hello World") ;
}
```

4. What is the difference between a class and a struct?
Schools don't teach structs.

5. In the context of C++ what is abstraction?
Staring into space with a blank expression on my face wondering what my colleague's C++ code is supposed to do.

6. In the context of C++ describe the differences between Abstraction and Encapsulation.
Encapsulation is how the Ibuprofen is packaged. This is used after the Abstraction phase, when I realise he's used templates, and before I dispatch him to hospital.

7. In the context of C++ what is Inheritance? Give an example.
It's when my colleague leaves me that source code in his will.

8. In the context of C++ what is Polymorphism? Give an example.
It's when I have a mutable member of class Parrot.

9. What is a virtual function?
One I haven't written yet.

10. What is a destructor?
It's a Transformer 'bot.

11. When would you use a virtual destructor?
When Toys'R'Us are out of the real ones. A painted Barbie will do in a pinch.

12. Does a virtual destructor work the same way as a virtual function?
Yes - neither exists, and neither does anything.

13. What member functions does the compiler generate for you automatically?
The constructor and destructor. (Or is that the VC++ class wizard?)

14. How should you write a copy constructor?
By pasting in someone else's source code.

15. What does the const keyword do, and where do you use it?
It causes compilation errors, so I don't.

16. What are namespaces, and why are they useful?
They're a way of pretending my global variables aren't global, and therefore good for fooling my colleague who thinks he's a better programmer than me just because he doesn't use globals.

Paul Grenyer originally posted these questions on `accu.general`, and Alan Bellingham is the longest serving programmer at his company, showing that twelve years C++ experience can sometimes be six month's experience twenty-four times over.

Francis' Scribbles

by Francis Glassborow

Teaching Programming

Two things turned my thoughts to the problems of teaching programming. The first was a letter in the recent issue of C Vu in which the writer thought he had detected a radical change in editorial direction. The second was sitting talking with my editor about why Java would not work for the book I was writing.

It is very easy to confuse teaching programming with teaching a programming language. To some extent you cannot do one without the other, though it is possible to invent a programming language purely for the purpose of teaching programming. The outstanding example of that route is Donald Knuth's `MIX` language that he uses in *The Art of Computer Programming*. Indeed if you are going to focus on fundamental computing issues that is, in my opinion, about the only viable choice.

In general Computer Science/Engineering Departments want to focus on teaching programming and the language used for practical work is just that, the language used by students to provide practical experience of what they have learnt in theory.

I think the objective of teaching good basic programming is very laudable, and is not, in intent, different to the way science and engineering disciplines are generally taught at that level. However, practical work is important and so students need to learn a computing language in which they can experiment.

The thing that concerns me is both the choice of language and the accuracy with which it is taught.

Suppose you were designing a course on human communication, what language would you choose for the practical side? For the sake of argument, suppose that you decided that it would benefit students to learn a new natural language so that they did not import their bad habits from the language they currently use.

Should those teaching this language be reasonably fluent in the language? Should they understand correct idiomatic usage? Should they have a basic grasp of the syntax and semantics? Remember that their objective is not teaching the language but teaching communication skills. Nonetheless, I suspect most students would be disappointed to have invested a great deal of time learning a new language badly or inappropriately.

I am reminded of a friend of my parents who read a degree in Oriental Languages at Oxford. He then joined the British Diplomatic Corps and because of his degree was posted to Khartoum. He used to relate the story of his arrival at the rail station where he leapt out of the train bubbling with youthful enthusiasm and instructed the local railway porters in fluent classical Arabic. They just stood open mouthed and deeply puzzled by this weird Englishman who seemed to be spouting some curious language that vaguely related to their normal speech. There was very little communication, until he reverted to English (though later he became fluent in spoken Arabic, and was awarded the title of Sheik by the local population out of respect for his profound knowledge of the Koran and his ability to settle disputes).

Returning to the subject of teaching programming; whatever language is used for the practical side must not, in my opinion, control what is taught on the theoretical side. It needs to be a good general-purpose language and not one tied to a specific paradigm. If the course is substantially about computing then it will almost certainly be necessary to use more than one language because I do not know of any single language that enables the student to gain practical experience of such distinct paradigms as object orientation and functional programming.

My contention is that the specialist student should finish with a sound understanding of programming coupled with a practical knowledge of at least one programming language. They need to understand the broad range of programming techniques and how those are implemented in one or more languages.

They should also, in my opinion, understand that programming requires responsibility. Such simple misconceptions as using a compiler to attempt to determine what a piece of code should do should have been long eradicated from their thinking. They should not be the computing equivalent of the Oxford graduate chemist who, needing to know the boiling point of nitro-glycerine set up a standard experiment to determine it (he had never thought of looking up such information in a book; that would have been cheating!)

The computing specialist probably should learn most, if not all, about the languages they are using but that does not apply to the student for whom programming is either an adjunct to their main subject or for whom it is provided as a form of educational enrichment.

Such students should not be required to delve into fundamental theory nor should they be expected to master an entire language. Their course should be designed to meet their needs. To some extent academics from a Computer Science Department can be very bad news to such students because such people often find it hard to restrain themselves from teaching everything. If I want to mend a fuse, I do not need to know about the physics of blowing a fuse. I need to know a number of practical things such as checking what caused the fuse to blow in the first place (yes, it was a current surge but that is too low level to be practically useful).

Basic programming is not that difficult, and definitely much easier than many programmers would have you believe. Using C++ to write simple programs is not that hard if you do not require that the student learn everything about the language.

I do not think Java is a good tool for this level of programming because the student has too many ways in which things can come unstuck, and too restricted a way in which code must be written. From the instructor's position, Java demands too much knowledge. Granted that those statements are my personal opinion but they are based on my experiences in writing my book and those of my test students. Object-orientation is a great way to write some types of program but it is far from a great way to learn to program. At the cutting code level most people think procedurally and the knowledge required to implement ideas as objects is unhelpful.

On Writing for Publication

Quite a few of the current crop of books on C++ are based on articles that have either been published in magazines or on the Internet. When I look at books like *C++ Ruminations* by Andy Koenig & Barbara Moo I see work that has clearly been rewritten for the different medium. This is not always the case and in my opinion it matters.

When I write for a periodical I can afford to express my strongly held opinions without distinguishing them from facts because the readers, other columnists and even my own future writing can make it clear that there are alternative viewpoints. It is a mistake to water down one's opinions and ideas when writing for a periodical.

This also applies to books that are clearly about personal opinion, the reader is forewarned that my excellent ideas are just mine and others may have even better ones or just plain different ones.

When writing a technical book where the reader is led to believe that the author is writing factual material, the rules change. Taking a series of articles most of which are factual but some of which are opinion and wrapping them up in a book removes the context that validates such mixtures.

Over the last decade Stephen Dewhurst has written over 120 columns under the title *C++ Gotchas*. The readers of those columns had immediate access to alternative views from other columnists in the same publications (initially C++ Report, then CUJ) so they would have been immediately aware of the places where Stephen was in strong disagreement with others such as Dan Saks, Herb Sutter and Andy Koenig.

A couple of months ago I received a signed copy of his book, *C++ Gotchas*. I read it, and decided to leave the review process to someone else. The book is waiting for a review. But if you see a copy on the shelves of your local book supplier take it off the shelf and give serious consideration to buying it. I think the author has done himself a serious disservice by including several articles that are pure opinion and have little to do with what I consider a Gotcha. That only matters in so far as that those items are largely near the start and may lead the prospective purchaser into misjudging the book as a whole.

What is more serious is that some of the other articles include material that is also a matter of opinion, but in contexts that make that far from obvious.

My conclusion is that this is an example of writing that needed much closer consideration before being transferred from periodical to book.

As I have got further into the process of writing my own book I have become increasingly sensitive to the different objectives that a book must meet. Typos and small technical errors in magazine articles are warts that we could do without, those same mistakes moved to a book can be deeply damaging because few people buy a book and immediately download errata from the Web. Indeed they would expect that a well-written book would not contain errors.

Authors will defend themselves on the basis that books are always their personal opinions. I do not think that is true. Technical books should clearly distinguish between technical answers and issues of idiom, personal style and presentation. Expressing an opinion as fact is likely to detract from the authority with which the author's factual statements are taken.

The Concept of Membership

Clubs, societies and the like come in many forms. The two extremes are clubs that are run entirely by members and for the benefit of members, and commercial clubs run for the financial benefits of the organisers.

ACCU is very close to the first extreme. Because certain work places a vast load on an individual we have recently moved to accepting the contracting of some work to members for their financial gain. However volunteers do the overwhelming majority of the work.

If you and a group of colleagues went down to the local place of refreshment you would soon notice the person who never contributed but always consumed. ACCU is different because many members never see the work that is quietly done by many individuals but it is still a community in which all should seek to contribute as well as consume.

Writers of articles need readers, but they also need appreciation or else they will stop writing. I know that many of you do appreciate all that is done because over the years you have told me and thanked me. But the people you should really be telling are all those who make a little contribution of time or skill.

Time is the most precious thing we have because it flows inexorably from the moment of our birth to the moment of our death. We can never go back, time once spent is gone. We should appreciate when others give us a gift of their time. We should also have the courage to speak out when someone is spending time on some well-intentioned pursuit that we think is unproductive.

Please take a few moments of your time to write down three good things about ACCU and three things that you think could be done better or differently. Now ask yourself how the ACCU Committee can know about those notes if you do not tell them.

When you pay your subscription to ACCU it is more like buying a ticket to a dance than to the theatre. The latter is just your entry to be allowed to watch but the former is your payment to be allowed to contribute. A dance in which no one danced would be pretty meaningless, and a play where the audience all wanted to be on stage would please no one.

Know the difference and act accordingly.

Problems

Problem 7 restated

Look at the following code that is designed to draw an approximate circle by drawing a polygon with a sufficiently large number of sides.

The function that draws a side is based on the same design principles as `yline()`. The supplied software uses the closest pixel to each vertex as the start/end point of a side.

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

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

Commentary

This is another little problem that arose out the book I am writing. Writing a fill function for an arbitrary closed curve is a difficult programming task. The one algorithm I know that is theoretically guaranteed to work for all closed curves has a severe practical problem, it is deeply recursive and blows up most stack based machines.

Nesting ever-smaller versions of a polygon has a number of problems. One is that some polygons simply do not nest however much you try. But regular polygons do not suffer from that problem.

The problem with the above code is that when the circumference of the circle reduces to less than 100 units our line-drawing algorithm decides that there is nothing to draw because each side of the polygon is less than 1 unit in length. The error in the above code is in the concept that a 100-sided polygon is a good representation of a circle regardless of the radius of the circle. If we define a 'circle' something like this:

```
void drawcircle(double radius,
                point2d centre) {
  drawpolygon(radius, 2*radius, centre);
}
```

We have a better chance of achieving what we want. Small circles are represented by polygons with few sides, large circles need far more sides for a good approximation.

Question

Does anyone know a good fill algorithm? Good in this context means one that a relative novice could understand even if the code lacks the efficiency you would want for your high-speed game.

Problem 8

Many of you know that C++ strings can contain embedded nulls. However how do you get them there? And what about C arrays of `char`?

Look at the following code and decide what happens in each case.

```
int main() {
   char const * mess = "One\0Two\0Three";
   cout << mess << '\n';
   cout << mess+4 << '\n';
   cout << mess+8 << '\n';
}
```

And now repeat the exercise for:

```
int main() {
   char mess[] = "One\0Two\0Three";
   cout << mess << '\n';
   cout << mess+4 << '\n';
   cout << mess+8 << '\n';
}
```

And for:

```
int main() {
   std::string mess("One\0Two\0Three");
   cout << mess << '\n';
   cout << mess[4] << '\n';
   cout << mess[8] << '\n';
}
```

And now correct that last program so that`mess` is initialised by the whole of the literal not just part of it.

For bonus credits decide whether the `c_str()` member of `string` behaves more like a pointer to literal or an array initialised with a literal.

Cryptic Clues for Prizes

In my days as a teacher I used to encourage my pupils to have fun with numbers. I found the kind of cross-number problem that got inserted into textbooks boring in the extreme. There was no great pleasure in solving them. Solving a clue should give you some kind of emotional kick.

Cryptic crossword clues can be great fun. Seeing into the mind of the setter can be a pleasure. Distorting your view of a clue till you see it clearly is both a challenge and the bringer of intellectual pleasure.

I tried the same thing with cross number problems and many of my pupils enjoyed both setting them and solving them. I think that some of my readers may enjoy them as well. I do not have time to work out an entire cross number puzzle but let me give you a clue to think about.

A tailless Roman mile.

The answer is a four-digit number. When you get the answer it will obviously be correct. Do not send me the answer but send me a clue to another number using the same general conventions. I will publish the answer to my clue in the next issue. I will also include all your clues with the names of those setting them.

I will choose one name at random for a small prize.

Features

Professionalism in Programming #19

A passing comment

Pete Goodliffe <pete@cthree.org>

Comments are free but facts are sacred.

Charles Prestwich Scott[1]

We all like comments, don't we? We all know that comments are a Good Thing, don't we? Comments are rather like opinions. You're free to make them, but just because they're expressed, it doesn't necessarily mean they're right. In this article we'll spend a little time thinking about the details of writing these things.

From the moment you were taught to program you learnt to write comments. You were told that comments aid the readability of code, and were probably encouraged to write lots of them. But in this game we need to be thinking more about quality than quantity. Comments are our lifeline, memory jog and guide through code. We should treat them with the respect they deserve.

I set my syntax highlighting code editor to display comments in green. That's my thing. I get an immediate feeling of the quality of a bit of code, and how easy it's going to be to work with, as soon as I load up a source file. A nice proportion of green spread through in the right pattern makes me feel good about the world. The opposite makes me stroll to the kitchen for a strong coffee before going any further.

Comments can make the difference between bad code and good code, between a grossly complex and unfathomable morass of logic, and clear algorithms. But let's not over state the case – there are things far more important than comments to get right. When your code is in the right state your comments are the "icing on the cake", delicately placed to add aesthetics and value, rather than liberally slapped on to cover over all the cracks and blemishes.

In this sense, good code commenting is a strategy to avoid writing intimidating code. Comments will rarely be a magic addition that will turn sour code sweet.

What is a code comment?

Don't skip this section! OK, so this is an excruciating place to start from. We all know what a comment is, surely? But this question is more philosophical than you might think.

Syntactically, a comment is block of source that the compiler will just plain ignore. Put what you like in it, the names of your grand children or the colour of your favourite mackerel, the compiler won't give two hoots as it merrily parses its way through the file[2].

Semantically, a comment is the difference between a dingy dirt track and a well-lit highway. The comment is an annotation of the code it's situated by. You can use it as a highlighter to make a particular problem area stand out or as a documentation medium in your header file. You might use comments to describe the shape of an algorithm aiding the maintenance programmer (which may be you later on) or might just place comment blocks as a marker between each function to help you navigate a source file more quickly.

Notice in all of this that comments are entirely aimed at the human reader, not the computer. In this sense, comments are the most human-focused brick in the programming wall. It's the kind of brick with ornate moulding as opposed to the surrounding functional breezeblocks. If we want to improve the quality of our comments we need to look at what the human really needs as they read code and address that.

1 C.P. Scott (1846-1932) was an eminent British journalist who edited The Guardian for 57 years. He pursued a consistently radical liberal editorial stance, even in the face of public hostility.

2 Of course, what chews up and spits out the comments differs with the kind of language you're using. In C/C++ the monstrous preprocessor beast devours comments before the compiler proper ever gets a look in. In other compiled languages the compiler itself throws away comments as it tokenises the source. In interpreted languages your intense commenting may slow down execution of the program as the interpreter has to do the work of jumping over the colour of your favourite mackerel.

Code comments are not the only documentation your code should ever get. Comments are not specifications. They are not design documents. They are not API references. However, they are an invaluable form of documentation that will always be physically attached to the code (unless someone maliciously hits delete). Their close proximity means they're more likely to be updated, and more likely to be read in context. It's an internal documentation mechanism.

As responsible programmers we have a duty to comment well.

What do comments look like?

Well, they're green aren't they? Or at least they are for me.

Traditional C comments come in blocks between `/*` and `*/`, and can span any number of lines. C++, C99 and Java add the single line comment that follows `//`. Other languages provide similar block and line type comment facilities, but with different syntaxes.

Again, this is elementary subject matter, but you'll often see these different types of comment block used in subtly different ways. We'll see some of this as we go along. However, any commenting scheme that makes too cute a use of subtle differentiations based on syntax should be viewed warily.

How many comments?

Having established that we need to work at quality, not quantity, we must ask ourselves how many comments we really need? This is totally influenced by what goes into the comments - so the next section will have a large bearing on this. Preview: favour the minimalist comment strategy.

Student programmers are taught to write comments, and lots of them. As a case in point, the university Computer Science course I took had an automated code assignment marker. To be fair, it was quite a clever piece of work. Almost. Part of the grade awarded was determined by the ratio of comments to code. This didn't encourage students to write meaningful comments, just to add large blocks of nonsense to increase their grade. The text of the Jabberwocky in one assignment managed to get me full marks, but was about as inappropriate as you can get.

There is such a thing as too much commenting. Just as bad comments can be worse than no comments at all, so can too many comments. They can easily hide the real meaning of code, especially if you have to spend more time trawling through complex paragraphs of comment than the actual code that you need to read.

I liken this skill to being a good musician. When playing in a band it's not about how much noise you can add in at every conceivable opportunity. The more you play your instrument, the more complex the overall sound, the worse the music. Too many comments muddy the code. A good musician doesn't have to think "When should I stop playing and let someone else have a chance?" They should naturally only play when it will really add something. It's about playing the minimum you can to create the best sound possible. A lot of the beauty is in the space. We should only be writing comments when it really adds something.

The kind of people who will read your comments can also read the code, so prefer to document as much as possible in the code itself rather than in comments. It's what they'll believe, anyway. Comments have a nasty tendency to lie. Consider your code statements as the 'first level' of comment, and make them self-documenting.

How do you do this? Code self-documents through intuitive function and variable naming, through good structure and control-flow, and through clear and logical indenting with layout that reflects the program structure, rather than by using elaborate comments to make up for poor algorithm design. Use named constants instead of 'magic numbers' which need explaining. Choose types well so that it's obvious what are the reasonable values - enumerations are good for this. Sacrifice small optimisations if they'll confuse the way the code reads. If you see a comment stating something that could be enforced by the language itself (e.g. "`// this variable should only be accessed by class foo`") you should be worried.

Bad code can be at worst wrong. Bad comments can lie and that can be a whole lot nastier. The fewer comments you write the less chance you have of writing bad comments!

What goes inside our comments?

Aside from the tedious language-level concerns of what characters we can put inside our comment delimiters, what should we be writing in comments?

Here are a few basic things to consider that can drastically improve the quality of your comments:

- **Only include the truth**
  When is a comment not a comment? When it's a lie. OK, you'll probably not deliberately type in lies but it's easy to accidentally introduce errors, especially when modifying code that has already been commented. Try to avoid writing any comments that may go out of date if you know some future modifications may be made. See the "Working with comments" section below.
- **Only include comments of worth**
  Little witty cryptic comments may be witty, and they might only be little, but just don't put them in. They get in the way. They confuse. In-jokes that you and one other programmer get should not enter into the code. Neither should expletives or comments that are unnecessarily critical - you can never tell where your code will end up in a month or year's time...
- **Don't describe the code**
  Worthless descriptive comments range from the elementary example of "`++i; // increment i`" to a description of an algorithm followed by the exact code for the algorithm. There is no need to restate the code laboriously in English unless you're documenting a really complex algorithm that is impenetrable without it. And then you should probably worry more about rewriting the algorithm than the comment.
- **Be clear**
  Your comment serves to annotate and explain the code. Don't be ambiguous. Be as specific as you can (without writing a thesis about each line of a function). If someone reads your comment and wonders what it means you have made the code worse, and slowed down their comprehension.
- **Explain why not how**
  This is a key point. Read this paragraph twice. Then eat the page. Your comments shouldn't be describing how the code works. You can see that by reading the code. After all, the code is the definitive description of how the code works. And the code has been written clearly and comprehensibly. Hasn't it? You should focus more on describing why something is written the way it is, or what the next block of statements ultimately achieve. Constantly check whether you're writing "`/* update WidgetList structure from GlbWLRegistry */`" or "`/* cache widget information for later */`". They might mean the same thing, but one conveys the intent of the code it refers to, the other just tells you what it's doing.
- **Write comprehensibly**
  You don't necessarily need to write complete grammatically correct English sentences inside every comment you write. However, the comment must be readable. Cute abbreviations of words usually serve to confuse the reader too much - especially if English is not their first language.
- **End of blocks?**
  You will see that some programmers have a habit of commenting the end of every control block, for example putting "`// end if (a == 1)`" after the closing brace of an if statement. This is usually a sign of a novice programmer following the advice of a well-meaning teacher. It is a redundant form of comment, and adds noise to the code that needs to be filtered out before real comprehension can occur. The bottom of a block should be viewable from the same page as the top, and the code layout should make the loop/conditional block clear. All extra verbiage should be avoided. There is a similar practice for `#ifdef` blocks, which although has a slightly more compelling argument I also think is redundant. If you have so many nested `#ifdefs` that you need this kind of documentation then you have bigger problems with your code to sort out.
- **The unexpected**
  If any bits of the code you write are unusual, unexpected or surprising then document them with a comment. You will thank yourself when you come back having forgotten all about the problem. If there are specific workarounds, say for an operating system issue, then mention this in a comment.

How to write comments

So let's begin to pull this together and look in more detail at how and when to write a good code comment.

We've seen that well written code shouldn't really need comments, that variable names should be self-explanatory. Function names like `f()` and `g()` scream out for comments to describe them, but `someGoodExample()` doesn't ask for it at all. You can see it's a good example function name.

Here's a little worked example to illustrate some of these principles of commenting. Consider the following snippet of C++ code. Aside from other idiomatic criticisms, it's not entirely clear what's being done.

```
int j = wlst.sz();
for (int i = 0; i < j; ++i)
j(wlst[i]);
```

Euch. There's some room for improvement here, so let's improve. The code can be made an awful lot clearer by applying some sensible layout rules and adding a few comments.

```
// Find the number of widgets in the widget list
int j = wlst.sz();

// Iterate over all widgets
for (int i = 0; i < j; ++i) {
  // Print out this widget
  j(wlst[i]);
}
```

Much better! Now it's entirely clear what the code snippet is supposed to be doing. I'm still not entirely happy, though. By giving the functions and variables appropriate names we no longer need any comments at all, the code describes itself:

```
int const numWidgets = widgets.size();
for (int i = 0; i < numWidgets; ++i) {
  printWidget(widgets[i]);
}
```

Note that I didn't rename `i` to something more long winded and tedious. It's a loop variable with a very small scope. Calling it `loopCounter` would have been overkill, and would arguably make the code harder to read.

Although this is a small and reasonably tedious example, now scale it up. Imagine functions much larger and several times more complex. It doesn't take much to see the difference good commenting strategies make. I've seen plenty of code like that first example above, and had to improve it by taking it to the next stage. When modifying existing code you often don't have the luxury of being able to rename all the functions and class names to more sensible choices.

Beware of the warning signs. If you end up writing reams of comments that explain how a complex algorithm is working, stop. First pat yourself on the back for thinking about documenting what's going on. But then consider whether you could change the code or the algorithm to make it clearer. Remember you don't need to prematurely optimise (and obfuscate). Perhaps you could split the code into several well named functions, rename the variables etc. As Kernighan and Plauger said: "Don't document bad code – rewrite it." [1]

If you find yourself using comments to describe use of variables, you probably need to rename the variable. If you are documenting certain conditions that should always hold, perhaps you should be writing an assertion.

A comment on aesthetics

You'll hear people religiously tout how you should format your comments. I'm not going to prescribe any One True Way to format them here. But there are a few things to consider. Interpret them as guidelines according to your personal taste rather than dictates.

Consistency

Commenting should be clear and consistent. Make a point of choosing a way of laying out your comments, and use it throughout. Every programmer has a different sense of aesthetics, so chose what works for you. Do use a house style if one exists, or examine (good) existing code and follow the styles you see there.

Many small formatting issues in comment writing may seem trivial - for example should each comment start with a capital letter or not?

However, if all your comments are randomly capitalised it will convey the sense of a lack of cohesion in the code, like the programmer didn't really think all that carefully as he crafted his code, or perhaps that the code grew by accretion rather than by design.

Clear block comments

You can tell that I like my syntax colouring editor, but there can almost be too much reliance on syntax colouring. Consider that your code may be read from a monochrome printout or viewed quickly in an editor without syntax colouring. The commenting should still be readable.

A few strategies can help here, especially regarding block comments. Placing the start and end markers (e.g. `/*` and `*/` in C and C++) on their own line makes them stand out. Placing a margin character down the left-hand side of a block comment also helps to make it appear as a single item, for example:

```
/*
 * This is much more readable
 * as a block comment in the midst
 * of a whole pile of code
 */
```

than

```
/*
a comment that might
  span a few lines but without
any margin character.
  */
```

At the very least line up the comment text so it's not a jagged mess.

Indenting comments

A comment shouldn't cut across the code and break up the logical flow. Keep it at the same level of indentation as rest of the code around it. That way the comment appears to apply to the correct 'level' of the code. Maybe it's a personal problem but I always have to stare hard at code like this:

```
void strangeCommentsAhoy() {
  for (int n = 0; n < JUST_ENOUGH_TIMES; ++n) {
// this is a meaningful comment about the next line
    doSomethingMeaningful(n);
// frankly it's confusing the pants off of me
    anotherUsefulOperation(n);
  }
}
```

In a loop without braces (which I'm not convinced is a good idea anyway) don't put a comment above the single looped statement - there be dragons. If you want a comment in there, wrap the whole lot up in braces. It's a safer strategy.

End of line comments

Most comments usually come on their own line. Sometimes a single line comment can follow a statement if it's only short. However its good practice to space the comment away to mark it as clearly apart from code, for example:

```
class HandyExample {
... some nice public stuff ...
private:
  int  numApples;         // end of line comments:
  bool oldManADustman;    // make them stand out
  int  transactionID;     // from the code
};
```

The above is a good example of using comment layout carefully to improve the appearance of the code. If each end of line comment came directly after the appropriate variable declaration they would look jagged, rather messy and require more squinting to read.

Helping you to read the code

Comments are usually written above the code that they describe rather than below it. This way the entire source code reads downwards, almost like a book. The comment serves to prepare reader for what is to come. Used with whitespace, commenting helps to break the code up into 'paragraphs'; a comment introduces a few lines explaining what they intend to achieve, these lines immediately follow, then a blank line, then the next block. This is such a convention that a comment with a blank line before it feels like a paragraph start, whereas a comment sandwiched in the middle of two lines of code feels more like a statement in brackets or a footnote.

Choose a low maintenance style

It's sensible to choose a low maintenance comment style. For example you'll see people write C style comment blocks that don't only have left asterisks as a margin, but also include a row of right margin asterisks. Whilst this arguably looks very pretty, the amount of work required to adjust a paragraph of text within such margins is ludicrous. When you could have moved on to the next task in hand, you have to instead waste effort carefully lining up all the asterisks on the right again. If the style involves using tabs things get even nastier: someone with a different sized tab stop opens the file and wonders what the original programmer was on since all the comments look incredibly ugly and badly lined up.

Note that the end of line comments we saw above are an example of reasonably hard to maintain code. How much effort you're prepared to spend is up to you. There is always a balance between good looking source code and maintenance effort. I prefer a little bit of effort to ugly code.

Breakwaters

Comments are often used as breakwaters between different sections of code. Programmers use different schemes to differentiate 'major' comments (this is a new section of code) from 'minor' comments (this describes a couple of lines of a function). This is perhaps where different people's aesthetic hackles really rise. It's not uncommon to see a C++ source file which contains the implementation of several classes with something like this between each section:

```
/***************************************************
 * class Foo implementation
 **************************************************/
```

Some people really go for large blocks of comment between functions, or even a single long comment line as a rule between them. I tend to place a couple of blank lines between functions and that's good enough for me. If you have functions large enough that you really need help to see where they start and end you may need to revise your code.

Try to avoid using these large rules to emphasise every comment in sight. Otherwise nothing gets emphasised. Good indentation and structure should group code together, not impressive comment ASCII art. However, well-chosen breakwater comments can help you to quickly navigate around a file.

Flags

Comments can also be used as inline 'flags' in the code. There are a number of conventions for these flags. It's common to see "`// XXX`" (no, not an 'explicit code' warning!) or "`// TODO`" littered though files which are still work in progress. Good syntax highlighting editors display these comments prominently by default. The former flag is used to mark troublesome code or something that needs rework. `TODO` often marks missing pieces of functionality for a later return.

File header comments

Each file should begin with a comment block that describes the contents of the source file. It's just a quick overview, a preface, providing some essential information that you always want displayed as soon as a file is opened. If such a header exists then another programmer who opens this file will feel safer about the contents; it shows the file was thoughtfully created rather than just hacked up as a dumping ground for some new code.

Some people advocate that this header should provide a list of every function, global variable etc defined, but I think that this is a maintenance disaster, and such a comment would rapidly get out of date.

The kind of information this file header should contain is:

- the purpose of the file (e.g., implementation of foo interface),
- the date the file was created (not last modified - this would fail to get updated and so become misleading),
- the author(s) and any modifiers, and
- a copyright statement describing ownership and copying rights.

Specifically this header should not contain a source file history describing every modification ever made. This kind of information exists in your source control system and doesn't need duplicating here. Indeed this is not just an issue of duplication; if you have to scroll through ten pages of modification history to get to the first line of code the file becomes tedious to work with. This has caused some to advocate moving the history to the end of the file, but it would still make the file slow to load and bothersome to work with.

If a source file is automatically generated by some tool during the build process, then you must arrange for this file to receive a comment header that states very clearly (in BIG SCARY CAPITAL LETTERS) where it originated from. This should prevent someone mistakenly editing it, only to have the contents regenerated at the next build.

Bug-fix notices

Another practice you'll see is placing comments where faults have been fixed. For example, you may come across code like this in the middle of a function:

```
// <bug reference> - changed to use blah.foo2()
// method because the old code didn't handle
// <some condition> properly
blah.foo2();
```

Although these are entered with the best intentions, to help you (or any newcomer) see what's happened in the course of development, they often do more harm than good. To understand the real problem being fixed you'd now have to look up the fault in your fault tracking system. Most of these comments are useless unless you pull out the previous revision of the file from source control to investigate what changed, by which time you may as well have retrieved the check-in comment anyway.

Comments like this quickly proliferate during the later stages of development and maintenance, and end up littering the source code with sidelines, stale information, and distractions from the main thread of execution. This is another of those political issues. There is often an argument for inserting a comment when you make a non-obvious fix, to prevent someone later revising the code and reintroducing the bug. However, in these well-chosen cases you are documenting the unexpected rather than placing a bug-fix notice.

Working with comments

You can use comments as a working tool in a number of useful ways as you go about writing code. But we also need to be careful we don't abuse them.

Helping you to write routines

A common approach when starting to write a new routine is to write out the structure of what needs to be achieved in English comments first, then start to fill the code in underneath each comment line.

If you do this, then when you've finished ask whether the remaining comments are still useful. Evaluate them against the criteria above, and revise or remove them if necessary. Don't just leave them and move on.

The alternative is to write the new routine without all the comment rigmarole, and then come back and add the necessary comments afterwards. However, the experienced programmer will comment as they go along. Experience shows you the right amount of commenting to use. Coming back to comment something either doesn't happen, or doesn't lead to the most appropriate comments because knowing the code so well, the non-obvious bits are all too obvious to you now.

Don't be afraid of using the flags we saw above, like `TODO`, whilst coding as markers to yourself. It's a good technique to avoid the embarrassment of forgetting to complete dusty code corners. You can easily search your entire codebase for these flags to find what still needs completing.

Comment rot

Comments rot. Well, all carelessly maintained code tends to rot, acquiring unsightly blemishes and losing the original neat design. However, comments seem to rot much more quickly than any other piece of code. They have a tendency to get out of date with the code they describe. This can quite quickly cause profoundly annoying results. For example, I was recently working on a section of code where a file contained the comment "`features A and B not yet implemented`". I needed both these facilities, so I went about implementing them. Only after having done this did I discover that feature B had in fact already been implemented and I have just reinvented a wheel. Feature A was redundant since the implementation of B had handled it as well. If the person who did this had removed the incorrect comment I would have been saved a lot of work.

The simple solution is this: when you fix, add, or modify any code, fix, add, or modify any comments around it. Don't just quickly hack a couple of random lines to get them to work and move on. Make sure that any code changes don't turn comments into lies. The corollary of this is that we must make comments easy to keep up to date, or they won't be updated. Comments must be clearly related to the correct section of code, not placed in random locations.

Another bad habit to avoid is leaving code commented out. This will bite you when you come back in a year's time, or when any other programmer stumbles across it. If you encounter some code that has been left in the source file but in a comment block you'll wonder why it's there. Was it a fix that was never completed? Is it work in progress? Did the code never work? Is the rest of the code functionally complete? Either leave a note explaining why you have commented the code out or take it out completely. You can always get it back from the source control system, after all. Even if you only think you're knocking something out temporarily, leave yourself a note. It's so easy to forget to come back and finish off a job.

Maintenance and the inane comment

As a maintenance programmer, it's best not to remove any inane comments you find, unless they are downright dangerous. Leave them as a warning for future maintenance programmers. Learn the interesting area flags like '`XXX`' and treat them with respect and caution. Also notice `printf` or other output statements in the code that have been commented out. These are a sure sign that there has been a problem area here in the past, treat the code with care! Be aware of comment rot, for example just because the comment says "`this defined in foo.c`" it doesn't mean it is any more. Always have faith in code and doubt comments.

Literate programming tools

I can't write about code comments without touching on this subject. Earlier on I said that comments are not API documents. However, there is one case when they can be. Literate programming tools, like Doxygen [2] and Javadoc [3] can read specially formatted comments in your source code and produce neatly formatted documentation from them[3]. I regard this as better than separate written API specs.

Now, if you already have a separate API document then keep it - and consider it the definitive guide. If a code comment and this specification differ, the specification is the one that should be right, since that is a document that will have been reviewed, discussed, agreed with a customer, etc.

However, for any new project I strongly recommend using a literate programming tool to document your code. Public interfaces need very careful specification documents to describe preconditions and explain the exact semantics of each function. By putting this in comments right there in the code, the documentation stands a much better chance of staying up to date, since an API change merely requires the change of a header file comment, rather than an off-line word processor document edit. You also have the documentation easily to hand whilst you're coding - it's conveniently there in the header file.

Documentation has a habit of not getting done. This is a simple and effective way to tackle the problem. Programmers also get a buzz out of running their neat code through a magic tool and getting high level documentation seemingly for free.

Conclusion

We write a lot of comments. That's because we write a lot of code. Learning to write the right sort of comment is important or our code may keel over under the weight of inappropriate and outdated commenting.

Comments are no more important than the code they annotate - you can't make bad code good using comments. Your aim should be self-documenting code that requires no comments at all.

I'd welcome any comments on this article...

Pete Goodliffe

References

[1] B.W. Kernighan, P.J. Plauger. *The Elements of Programming Style.* McGraw Hill, 1978. ISBN: 0-07-034199-0.
[2] Doxygen. Available from: `http://www.doxygen.org/`
[3] Javadoc. Available from: `http://java.sun.com/j2se/javadoc/`
[4] Pete Goodliffe. "Professionalism in Programming #5: Documenting code." *C Vu Volume 12, No 6.* ISSN: 1354-3164.

3 See a previous Professionalism in Programming article for more discussion of literate programming tools [4].

Installing Red Hat 8.0 (Psyche Linux)

Paul Grenyer <pjgrenyer@iee.org>

This article is an updated version of my Installing Red Hat 7.3 (Thread Linux) article, which can be found on my website's articles page [1], for Red Hat 8.0 (Psyche Linux).

Red Hat supply a reasonably comprehensive installation guide [2] which is more than most people will need to install Red Hat 8.0 Linux. However, although it explains what all the different options are, it is not always clear which should be used in what circumstance. My intention here is to describe each of the installation screens and explain what I think should be entered for the purposes of the Linux Server. For each screen I will give the page of the PDF version of the guide so that further information can be gained easily. From here on in The Official Red Hat 8.0 Installation Guide will be referred to as "the Installation Guide".

In my previous article [3] I stated that "There is probably no point in buying a monitor for the server… …as it is unnecessary to use one after installation." In the final part of this article I will explain how to install OpenSSH [4]. OpenSSH is a free version of the SSH protocol suite of network connectivity tools. It will allow you to control the Linux Server remotely and negate the need for a monitor or keyboard.

Installing the Linux Server

From this point forward I assume that you have a fully built and working PC, of a similar specification to that discussed in my previous article, and that it is ready to have Red Hat 8.0 Linux installed.

1. Booting from the CD and Selecting an Installation Option (p.33)

If your system is unable to boot from CDs please see section 1.4.1 on page 13 of the Installation Guide for alternative boot methods.

Set the computer's BIOS to boot from the CD-Rom, place the first Red Hat 8.0 installation disk in the drive and reboot. Let the machine go through the memory test and boot from the CD. A screen full of installation options should appear along with the Red Hat logo and version number.

The two listed installation options are:

Install or Upgrade Red Hat Linux in Graphical Mode

This is the default installation option and takes the user through the installation process with a high resolution GUI (Graphical User Interface). A reasonably good monitor and graphics card (capable of 800x600 resolution), keyboard and mouse are needed. To start this installation option simply press enter.

Install or Upgrade Linux in Text Mode

This takes the user through a text based installation process. It is ideal for people with very low resolution monitors and graphics cards or those who don't want to plug in a mouse. To start this installation process simply type `text` at the prompt and press enter.

Although it is not listed, the following installation option is also available:

Enable Low Resolution mode

This takes the user through a low resolution GUI installation process which is ideal for people who are using a low resolution monitor and graphics card that are capable of displaying the GUI, but not up to 800x600 (I'll be using this mode as my black and white monitor won't display 800x600). To start this installation process type `lowres` at the prompt and press enter.

All three processes have the same screens and request the same information. They are just displayed differently.

Select the mode you want and press enter. After the Kernel and your PC's hardware are initialised you may be asked whether you want to "...begin testing the CD media before installation..." If you want to test your CDs before the installation choose OK, otherwise choose skip.

Next, Anaconda, the installation program, is started and the user is first presented with a Red Hat splash screen and then the "Welcome to Red Hat Linux screen". Be patient as this can take a while and the screen may go blank for periods of 30 seconds or more.

2. Welcome to Red Hat Linux (p.40)

This screen just gives information about where to find installation documentation and information. Simply click next to move to the next screen.

3. Language Selection (p.40)

This screen allows the user to select the language that the installation process will be presented in. English is the default. If you want to use a different language simply select it.

Click next to move the next screen.

4. Keyboard Configuration (p.40)

This screen allows the user to select the keyboard layout they want to use. The default is `U.S. English` so is likely to need to be changed to `United Kingdom`.

Once you have configured your keyboard click next to move to the next screen.

5. Mouse Configuration (p.41)

This screen allows the user to configure their mouse. Simply select your mouse from the list, or the closest generic mouse, and put a tick in the `Emulate Three Buttons` check box if you want to emulate a three button mouse (with a non-three-button mouse).

Click next to move to the next screen.

6. Installation Options (p.42)

This screen allows the user to select the type of Red Hat Linux installation they want. For more information on what the individual options are see the Installation Guide. We want maximum control so that we can create our own partitions (if we want to) and install the minimum number of applications, so choose `Custom`.

Click next to move to the next screen.

7. Choosing Your Partition Strategy (p.43)

This screen allows the user to set up the various partitions that Linux uses. The easiest thing to do at this point is select "Have the installer automatically partition for you." This will even work if you already have a Windows partition you wish to keep on the disk.

However, this option will create smaller partitions than the one I listed in the previous Linux Server article (Red Hat 8.0 recommends that some of the partitions should be bigger than were stated in my previous article, which was based on Red Hat 7.2). So in order to make good use of the large amount of space that is likely to be available to you, select "Manual partition with Disk Druid" and click next. Disk Druid (p.45) is very simple to use:

Your hard disk is likely to be labeled `/dev/hdc` in the list box at the bottom of the Disk Druid screen. If you already have a Windows partition this will be listed as type `vfat` or `NTFS/HPFS`. If you have a second disk this is likely be labeled `/dev/hdd`. To create the partitions follow the steps below:

1. Swap partition (256MB)

Click New to create a new partition. In the Allowable Drives list box make sure the hard disk you want to install the partition on is the only disk selected. For the swap partition ignore the Mount Point drop down box and select `swap` from the File System drop down box. Enter `256` into the Size (MB) edit box to create a partition of 256 MB. Make sure that Fixed Size is selected in the Additional Size Options group box and click OK.

2. Boot partition (75MB)

Click New to create a new partition. In the Allowable Drives list box make sure the hard disk you want to install the partition on is the only disk selected. For the boot partition select `/boot` from the Mount Point drop down box and select `ext3` from the File System drop down box. Enter `75` into the Size (MB) edit box to create a partition of 75 MB. Make sure that Fixed Size is selected in the Additional Size Options group box and click OK.

3. Root (3700MB)

Click New to create a new partition. In the Allowable Drives list box make sure the hard disk you want to install the partition on is the only disk selected. For the root partition select `/` from the Mount Point drop down box and select ext3 from the File System drop down box. Enter 3700 into the Size (MB) edit box to create a partition of 3.7 GB. Make sure that Fixed Size is selected in the Additional Size Options group box and click OK.

4. Usr (4000MB)

Click New to create a new partition. In the Allowable Drives list box make sure the hard disk you want to install the partition on is the only disk selected. For the usr partition select /usr from the Mount Point drop down box and select ext3 from the File System drop down box. Enter 4000 into the Size (MB) edit box to create a partition of 4 GB. Make sure that Fixed Size is selected in the Additional Size Options group box and click OK.

5. Var (385MB)

Click New to create a new partition. In the Allowable Drives list box make sure the hard disk you want to install the partition on is the only disk selected. For the var partition select /var from the Mount Point drop down box and select ext3 from the File System drop down box. Enter 385 into the Size (MB) edit box to create a partition of 385 MB. Make sure that Fixed Size is selected in the Additional Size Options group box and click OK.

6. Home (2500MB)

Click New to create a new partition. In the Allowable Drives list box make sure the hard disk you want to install the partition on is the only disk selected. For the home partition select /home from the Mount Point drop down box and select ext3 from the File System drop down box. Enter 2500 into the Size (MB) edit box to create a partition of 2.5 GB. Make sure that Fixed Size is selected in the Additional Size Options group box and click OK.

Click Next to move on from Disk Druid.

8. Boot Loader configuration (p.51)

This screen allows the user to change and configure the boot loader. GRUB is set as the default boot loader. it can be changed by clicking the Change Boot Loader button. Leave the default set to GRUB unless you particularly want to use LILO or a boot disk. See the installation guide for more details.

The user can also select which operating system is booted by default. This is only relevant if you already have another operating system installed, such as Windows. Put a tick in the box beside the operating system you would like to boot by default.

It is also possible to set a GRUB password from this screen. GRUB passwords provide a security mechanism in an environment where physical access to your server is available. If you are installing GRUB as your boot loader, you should create a password to protect your system. Without a GRUB password, users with access to your system can pass options to the kernel which can compromise your system security. With a GRUB password in place, the password must first be entered in order to select any non-standard boot options. To set a GRUB password, put a tick in the Use A Boot Loader Password box and enter a password.

Leave the tick out of the Configure Advanced Boot Loader Password box unless you particularly want to install the boot loader somewhere other than the Master Boot Record. See the Installation Guide for more information.

Click next to move to the next screen.

9. Network configuration (p.55)

Take the tick out of Active On Boot tick box for DHCP as eventually the Linux Server will also be a DHCP server.

Click Edit to edit the eth0 interface, which is the servers network (card) interface. This brings up the Edit Interface eth0 dialog box. Take the tick out of the Configure Using DHCP tick box and put a tick into the Activate On Boot tick box. If you are planning to use the Linux server as part of an existing network you should know what IP Address and Network Mask settings to enter. If you are creating a new network or just connecting the Linux Server to another PC running Windows or Linux use the following settings:

```
IP Address: 192.168.1.1
   (use 192.168.1.2 for the other (client) PC)
Netmask: 255.255.255.0
```

We are not concerned with the other settings at this point so leave the other boxes blank.

Click Next to move to the next screen and ignore the warnings.

10. Firewall Configuration (p.57)

Leave the default firewall settings as we will be revisiting them when we set up OpenSSH later.

Click Next to move to the next screen.

11. Language Support (p.59)

Put a tick in the boxes for the language you want the Linux Server to support. Most people will probably only want to support one language (e.g. English (Great Britain)). Take the tick out of the English (USA) box and put it in the language of your choice.

Click Next to move to the next screen.

12. Time Zone (p.60)

Select your time zone (e.g. Europe/London).

Click Next to move to the next screen.

13. Root Password (p.61)

Setting up a root account and password is one of the most important steps during your installation. The root account is used to install packages, upgrade RPM packages, and perform most system maintenance. Logging in as root gives you complete control over your system. Use the root account only for system administration. Create a non-root account for your general use. These basic rules will minimize the chances of a typo or an incorrect command doing damage to your system.

The installation program will prompt you to set a root password for your system. You must enter a root password. The installation program will not let you proceed to the next section without entering a root password.

The root password must be at least six characters long; the password you type is not echoed to the screen. You must enter the password twice; if the two passwords do not match, the installation program will ask you to enter them again.

You should make the root password something you can remember, but not something that is easy for someone else to guess. Your name, your phone number, qwerty, password, root, 123456, and anteater are all examples of **bad** passwords. Good passwords mix numerals with upper and lower case letters and do not contain dictionary words: Aard387vark or 420BMttNT, for example. Remember that the password is case-sensitive. If you write down your password, keep it in a secure place. However, it is recommended that you do not write down this or any password you create.

Once you've entered the root password, set up a user account by clicking Add. Complete the User Name, Full Name, Password and Confirm (password) edit boxes and click OK to create the new user.

Click Next to move to the next screen.

14. Authentication Configuration (p.64)

Leave defaults. Consult the Installation Guide for further details.

Click Next to move to the next screen. The message “Reading Package Information” will be displayed for a short time before the next screen is displayed.

15. Select Package Groups (p.66)

Deselect **all** the packages as we want the most basic system possible onto which we can install the latest versions of only the software we want. Consult the Installation Guide for further details.

Click Next to move to the next screen.

16. About to Install (p.69)

You are now ready to install the Linux Server.

Click next to install. Only installation CD1 is required for our minimal installation. Installation on my system takes a little over 11 minutes according to the displayed counter.

17. Create Boot Disk (p.70)

Once the installation is complete you will be asked to create a boot disk.

To create a boot disk, insert a blank, formatted diskette into your 3.5 inch disk drive and click Next. It is highly recommended that you create a boot disk. If for some reason your system were not able to boot properly using GRUB, LILO, or a third-party boot loader, a boot disk would enable you to properly boot your Red Hat Linux system. After a short delay your boot disk will be created. Remove it from your drive and label it clearly.

Click Next to move to the next screen.

18. Congratulations, your Red Hat system is complete (p.73)

Click Exit. The installer exits and your machine should reboot.

Make sure you take the CD out of the CD-Rom drive. Allow the system to boot back into the Linux Server. Remember, if you have set the boot loader up to boot a different OS by default you must select “Red Hat Linux” from the boot menu. If you decided to use a boot disk to boot the Linux Server, remember to use that. The boot process may try to auto-configure an onboard sound card, if you have one, during the boot process. Allow it to do this. Once Linux has finished booting you should see something similar to the following:

```
Red Hat Linux release 8.0 (Psyche)
Kernel 2.4.18-14 on i586

localhost login:
```

Congratulations! You have installed the Linux Server.

19. Shutting Down

It is very important to shut down the Linux Server correctly. Just like Microsoft Windows, Red Hat Linux cannot just be ‘switched off’. To shut down the Linux Server you must be logged in as `root`. It is not enough just to use the `su` command and enter the root password. If you are logged in as a user other than `root` type `logout` to logout.

To shutdown the Linux Server log in as `root`. At the login prompt enter `root` and press enter. Then enter the root password and press enter. At the prompt enter:

```
shutdown –h now
```

to shut down the Linux Server or:

```
shutdown –r now
```

to reboot the server. If you have an ATX motherboard the PC should power down automatically, if not you can turn it off manually when you receive the `Power Down` message.

Installing OpenSSH

This is the first package you should install on the Linux Server so that the monitor and keyboard can be removed permanently. I’m going to start as I mean to go on by explaining where to download the latest version of OpenSSH from and how to install it on the Linux Server. An older version of the two OpenSSH packages can be found on the first and second installation CDs in the `/RedHat/RPMS directory`, but I would highly recommend downloading the latest versions and using those.

The details of the SSH protocol are beyond the scope of this article, but it should be enough to say that SSH is a secure protocol that allows users to open consoles on remote computers and control the Linux Server via the command line. There are a number of SSH clients available. I will explain how to download and use one of the more popular Microsoft Windows SSH clients, PuTTY [6].

This section of the article is based on an article [7] that is part of the RedHat Yahoo [8] groups FAQs [9].

From this point on I assume that the client PC, running Microsoft Windows has the correct IP address and is connected to Linux Server via some kind of network or cross over cable. This is quite a big assumption. If it is too big please let me know.

1. Downloading OpenSSH and the PuTTY SSH client.

Red Hat 8.0, unlike Red Hat 7.3, comes with version 3.4 of OpenSSH and the OpenSSH Server preinstalled. The firewall is also pre-configured to allow access to SSH. If you would prefer not to update to the latest version of OpenSSH jump straight to step 7 (below), however this is not recommended.

The two SSH packages that need to be downloaded from the OpenSSH ftp server [10] are the OpenSSH package itself and the OpenSSH server, you may also like to download the OpenSSH client. The latest versions of the packages at the time of writing are as follows:

```
openssh-3.5p1-1.i386.rpm
openssh-server-3.5p1-1.i386.rpm
openssh-clients-3.5p1-1.i386.rpm
```

You should download the most up-to-date package on the server. All the packages are less than 1MB in size and do not take long to download even over a 56k dial-up connection.

The PuTTY executable can be downloaded from the PuTTY website’s download page [11]. PuTTY is a standalone executable which does not require any installation.

2. Transferring the OpenSSH Packages to the Linux Server

As we did the most basic possible installation of the Linux Server there is no way to transfer the packages from the machine that they have been downloaded on to the Linux Server, other than using a floppy disk.

Assuming the packages have been downloaded on a Microsoft Windows PC, get a fresh floppy disk and reformat it as usual. Although most disks come pre-formatted these days, I suggest you reformat it because for some reason my Linux Server couldn’t read a BASF floppy disk straight out of the box without reformatting it on Windows. Copy the OpenSSH packages onto the floppy disk, remove it from the Windows PC and place it in the Linux Servers floppy disk drive.

Before it can be used the floppy disk drive must be mounted. To do this log in as the non-root user that was set-up during the installation and type the following, followed by enter:

```
mount /dev/fd0
```

To view the contents of the floppy disk type the following and press enter:

```
ls /mnt/floppy
```

You should see a list of the OpenSSH packages that were copied onto the disk. It is very important that the floppy disk is not removed form the drive until after the drive has been un-mounted. To un-mount the floppy disk drive type the following and press enter:

```
umount /dev/fd0
```

However, there is no need to un-mount the floppy drive until after the packages have been installed. The packages can be installed directly from the floppy disk, so there is also no need to copy them to the Linux Server.

3. Installing the OpenSSH Packages

To install the packages, root permission is required. At this point the non-root user could be logged out and re-logged in as `root`, however there is an easier way. While logged in as a non-root user, type the following and press enter:

```
su
```

You will be prompted for a password. Enter the root password and press enter. You should now have root permissions. When you have finished type the following and press enter to relinquish root permissions:

```
exit
```

I found that the easiest way to install the latest OpenSSH was to remove the old one first. Do this by entering the following three commands one at a time at the command line and pressing enter:

```
rpm –e openssh-client
rpm –e openssh-server
rpm –e openssh
```

Assuming that the OpenSSH packages are on the floppy disk and the floppy disk drive has been successfully mounted, all that needs to be done to install the packages is type the following and press enter:

```
rpm –Uvh /mnt/floppy/openssh*
```

Evidence of the packages being installed will be displayed on the screen. The OpenSSH packages should now be installed and the floppy drive can be un-mounted (see above). Now would also be a good time to relinquish root permissions.

4. Configuring the Firewall

The firewall must be configured in order to allow remote computers to access the SSH Server that has just been installed on the Linux Server. To configure the firewall you must log in as `root`. Simply using the `su` command is not enough. If logged in as a non-root user, logout and re-login as `root` (see above). To run the firewall configuration program type the following and press enter:

```
lokkit
```

This will bring up the firewall configuration screen. Leave the Security Level as `High` and use the Tab key to select the Customize button and press enter. When the customize screen opens use the Tab key to selectSSH and then use the space bar to put an asterisk (*) in the box next to it. Use the Tab key again to select the OK button and then press enter. This will take you back the first screen. Use the Tab key to select to select the OK button and leave the firewall configuration program.

The firewall must be restarted for the new settings to take effect. To do this type the following and press enter:

```
/etc/rc.d/init.d/ipchains restart
```

5. Starting OpenSSH

To start OpenSSH you must be logged in as root or have used su to get root permissions (see above). To start OpenSSH type the following at the command line and press enter:

```
/etc/rc.d/init.d/sshd start
```

You should receive a number of messages on the screen which indicate that OpenSSH has been setup and started successfully. You are now ready to access the Linux Server remotely and will soon be able to remove the monitor and keyboard.

6. Using PuTTY to Access the Linux Server

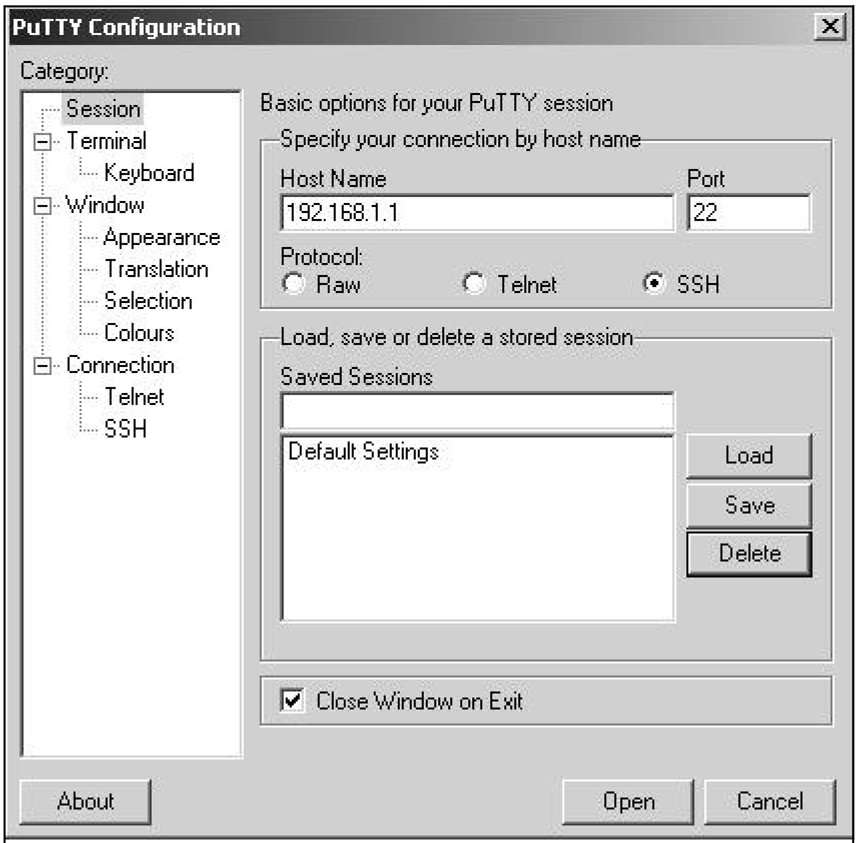
Assuming that OpenSSH has be started and is running successfully on the Linux Server and that the Windows client PC and Linux Server are correctly connected and configured, connecting to the Linux Server with PuTTY couldn't be easier!

Run the PuTTY executable by double clicking on it. Set the protocol to SSH, enter the IP address of the Linux Server and click open.

You'll receive a PuTTY security alert. As this is a closed system (i.e. not connected to the internet), ignore the alert and click OK. When prompted to do so, login as you would on the Linux Server. You will now get a command prompt you can use in exactly the same was as you would directly on the Linux Server.

I would recommend practicing logging in and shutting the Linux Server down, as discussed above, a few times through PuTTY. Then when you are comfortable with it, remove the keyboard and mouse from the Linux Server.

Paul Grenyer



Basic options for your PuTTY session
Specify your connection by host name
Host Name: 192.168.1.1
Port: 22
Protocol: Raw [✓ SSH] Telnet
Load, save or delete a stored session
Saved Sessions: Default Settings
[✓] Close Window on Exit

Figure 1: Putty Settings

References

[1] My website articles page: `http://www.paulgrenyer.co.uk/articles`

[2] The Official Red Hat Linux 8.0 Installation Guide: `http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/pdf/rhl-ig-x86-en-80.pdf`

[3] Linux Server Article: `http://www.paulgrenyer.co.uk/articles/choosing_linux_hardware.html`

[4] OpenSSH: `http://www.openssh.com`

[5] Red Hat Linux 8.0 Getting Started Guide: `http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/pdf/rhl-gsg-en-80.pdf`

[6] PuTTY SSH Client: `http://www.chiark.greenend.org.uk/~sgtatham/putty/`

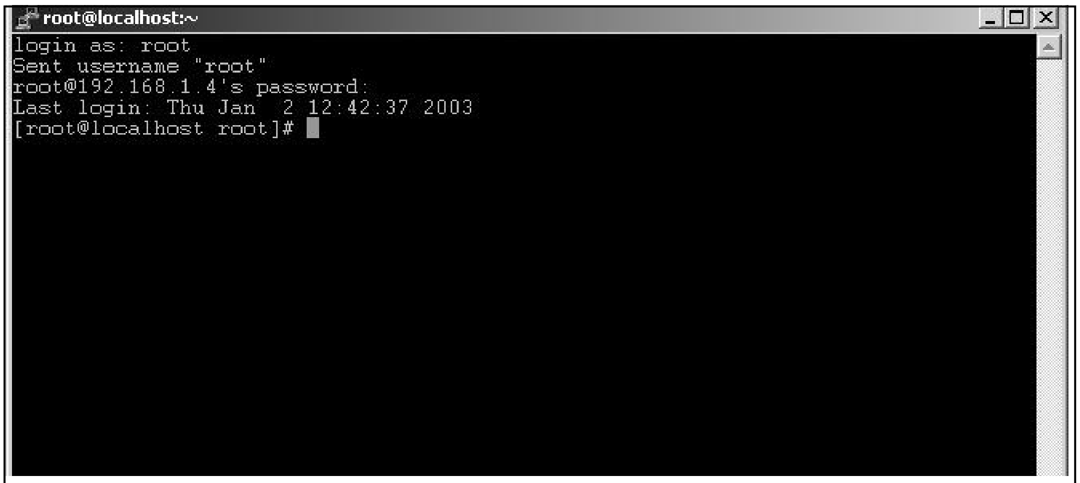
[7] RedHat Yahoo group FAQs SSH article: `http://home.nyc.rr.com/computertaijutsu/ssh.html`

[8] RedHat Yahoo group: `http://groups.yahoo.com/group/redhat/`

[9] RedHat Yahoo group FAQs: `http://home.nyc.rr.com/computertaijutsu/linfaq.html`

[10] OpenSSH FTP Server: `ftp://ftp.esat.net/pub/OpenBSD/OpenSSH/portable/rpm/RH73/`

[11] PuTTY Download Page: `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`



```
login as: root
Sent username "root"
root@192.168.1.4's password:
Last login: Thu Jan  2 12:42:37 2003
[root@localhost root]#
```

Figure 2: Putty Command Line Console

10 Things You Always Wanted to Know About Assert (But Were Afraid to Ask)

Garry Lancaster <glancaster@codemill.net>

This article is aimed primarily at those who are unfamiliar with assertions. However, even advanced programmers may make one or two new discoveries.

1. How do I call `assert`?

Write

```
assert(condition);
```

Some examples:

```
assert(i < 0);
assert(ptr);
assert(!str.empty());
```

2. What does `assert` do?

If assertions are enabled (see point 8) and the condition is false, the assertion fails, displaying a diagnostic message, then calling `abort`, a C and C++ standard library function that terminates your program without calling destructors or performing much else in the way of clean-up. Otherwise, if the condition is true, program execution continues as normal.

The exact format of the diagnostic message is implementation defined, but it must include the condition, converted to a string, plus the source file name and line number of the `assert` call. C99, the latest patchily-adopted standard version of C, additionally requires that the name of the function containing the `assert` be given.

Ensure your asserted condition have no side-effects: the mere evaluation of the condition should not affect program behaviour. There are a number of reasons for this, separation of concerns being one of them.

3. Why would I want to use `assert`?

`assert` allows you to document your assumptions about your program's state in code, in such a way that they can be verified automatically at runtime. The dramatic effects of an assertion failure will alert you to any cases where your assumptions do not hold. This makes `assert` a powerful bug detection tool.

4. Which headers contain `assert`?

`assert` is available in both C and C++. In both standard versions of C, C90 and C99, it lives in the C-style header file `<assert.h>`. In standard C++ it lives in the C++-style header file `<cassert>` also.

5. Is `assert` a member of the `std` namespace?

No: `assert` is a macro so it is not a member of a namespace even when you use the C++-style header file. Alternatively, given the way macros are globally visible, we could say it is a member of all namespaces.

`assert` is an odd choice of name for a macro, since it is all lower case. This is one instance where you should not follow the language standards' example in your own code: any macros you write should be named using `ALL_CAPS`.

6. Should I use `assert` to check for I/O errors?

Almost always not. Most programs should be designed to cope gracefully with I/O errors and the behaviour of a failed assertion is anything but graceful.

Use `assert` only to verify assumptions that must always be true for the program to function correctly: in other words to detect bugs.

The same arguments apply to other runtime checks that should be able to fail gracefully: don't use `assert`.

7. Should I use `assert` to check for memory allocation failures?

No. Memory allocation failure is another example of a failure that should be tolerated gracefully, which makes `assert` an unsuitable choice. See the previous point.

In C++ when you are using `new`, this is doubly misguided because `new` does not return a `NULL` pointer on allocation failure, it throws a `std::bad_alloc` exception.

```
// Please don't do this!
foo* ptr = new foo;
assert(ptr); // Wrong. Doubly.
```

8. How do I remove assertions from a program?

Assertions are most useful during program development and testing. Although some people advocate leaving assertions enabled in released code, it is common also to disable them, particularly when the overhead of the checking would affect your program's ability to hit its performance targets. It is not my intention to get into a full discussion as to the pros and cons of disabling assertions in release builds. I believe both approaches to be valid in some circumstances.

Disabling `assert` calls is supported as standard: `assert` is defined in such a way that if the macro `NDEBUG` is defined before inclusion of `<assert.h>` or `<cassert>`, `assert` calls compile to a no-op. In other words, somewhere in these header files lies code equivalent to

```
#ifdef NDEBUG
#define assert(cond)
#endif
```

This provides another good reason why the `assert` condition should not have any side-effects: if `NDEBUG` is defined, then the condition will never be evaluated.

Many compilers support debug and release modes of compilation. Often `NDEBUG` is defined automatically when compiling in release mode. See your compiler's documentation for more information.

Beware: `<assert.h>` and `<cassert>` are unlike most header files with respect to multiple inclusion, which may have a different effect from single inclusion. Each time you include one of these files, any previous definition of `assert` is undefined (using `#undef`) and `assert` is redefined. If `NDEBUG` is defined then the no-op definition of `assert` is given, otherwise the normal, checking, definition of `assert` is given. This allows you to write code like this:

```
// This code is allowed, but not recommended.
#include <cassert>

void with_assertions_enabled(int i) {
  assert(i > 0);  // Will be checked.
  ...
}

#define NDEBUG
#include <cassert>

void with_assertions_disabled(int i) {
   assert(i > 0);  // Not checked. No-op.
   ...
}
```

Code like this is confusing so the practice is not recommended. Nonetheless, you may see it in other people's work, so it is good to be aware of it.

9. Does `assert` support design by contract?

Design by contract is a term to describe a strict approach to verifying program assumptions and is most fully documented by Bertrand Meyer, the creator of the language, *Eiffel*. Briefly, he divides conditions into three categories:

- Pre-conditions that must be true on function entry.
- Post-conditions that must be true on function exit.
- Invariants that must be true both on function entry and on function exit.

The fundamental principle behind design by contract is that a correctly written function promises that its post-conditions and invariants will be true on exit *provided that* its pre-conditions and invariants are true on entry.

`assert` provides a mechanism for checking pre-conditions, post-conditions and invariants in C and C++ programs. However, it is fair to say

[concluded at foot of next page]

Non-Standard Code

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

I recently faced a problem which I felt I could only solve using non-standard code, and thought that some of the issues this brought up might be of interest to the readers of CVu.

Although the example is using the Microsoft compiler, the principles will hopefully be of general interest.

Firstly, what was my issue?

I'm writing C++ code which uses exceptions, and using some libraries which can also throw exceptions. Unfortunately it seems to be hard to provide useful information when exceptions occur, unless you know the types of exceptions which might be thrown; and as in this case there were several, unrelated, exception hierarchies involved as well as character literals this was not easy to discover.

A common idiom in standard C++ for such cases is to have a 'generic' exception handler function which is called from a catch statement. This simply re-throws the exception within another try/catch block where a catch handler is written for each known type.

An extremely simple example of this idiom follows:

```
// Called from within a catch handler to log
// a string from the current exception
void logException() {
  std::string error;
  try {
    // re-throw the exception to have another
    // look at it
    throw;
  }

  catch(std::exception & ex) {
    error = ex.what();
  }
  catch(CException * pEx) {
    static char szMsg[255];
    pEx->GetErrorMessage(szMsg,
                         sizeof(szMsg));
    pEx->Delete();
    error = szMsg;
  }
  // etc etc
  catch(const char *pStr) {
    error = pStr;
  }
  catch(...) {
    // Search me...
    throw;
  }
  std::cerr << "Uncaught exception: "
            << error << std::endl;
}

// Example of use
int main(int argc, char **argv) {
  try {
    return realmain(argc, argv);
  }
  catch(...) {
    logException();
    return EXIT_FAILURE;
  }
}
```

There were two problems with this approach. The first was that the idiom does not work in my current environment (Microsoft VC 6.0) because the destructor is called twice for re-thrown exceptions. This bug appears to have finally gone in VC.NET, but I need to continue supporting VC 6. The best work around for this bug is to ensure all destructors for exception objects can be called twice, but since I didn't write all the exception objects this was not an option.

[continued from page 21]

that other languages, particularly Eiffel, more fully support design by contract. For more details of Meyer's thinking see [1].

For checking pre-conditions it is permissible to use an `if` statement and a `throw` instead of an `assert`. For example,

```
void foo(int i) {
  if(i < 1)
    throw std::range_error("foo: i < 1");
}
```

versus

```
void foo(int i){
   assert(i >= 1);
}
```

There is no hard and fast rule as to which technique is best. I prefer assertions for testing pre-conditions when the calling code will be written by myself or another member of my development team. When the calling code will be written by external developers I prefer to use exceptions here.

10. What about extending `assert`?

Many compilers' `assert` macros go beyond what is required by the language standards. It is common to be offered the choice to continue execution past an assertion failure (just as if the assertion had not failed) or to drop into a debugger.

Furthermore, developers often create their own `assert`-like macros which provide more flexibility. Typical extensions are:

- Extensions similar to those offered by vendors and discussed previously, for working with compilers whose vendors provide only the standard-mandated assertion support.
- Providing more user-friendly messages. Particularly useful for those whose release builds contain enabled assertions.
- Multiple levels of protection, instead of the all-or-nothing `NDEBUG` approach. For example, if you define `ASSERT_QUICK` (for conditions that are checked quickly) and `ASSERT_SLOW` (for conditions that take longer to check), you have three speed versus bug detection tradeoffs available: you can build with both enabled, only enabling `ASSERT_QUICK` or with neither enabled.
- Assertions that throw on failure (the Eiffel approach). Pre-condition testing aside (see previous point), I don't recommend this approach: assertion failures should be loud and proud, not open to silent `catch`-and-continue. If you go down this road, think very carefully about the interactions with exception safety.
- Assertions that automatically log their message to an error file, or to the Windows Event Log. They could even send an e-mail or an SMS message to a system administrator.

Extending `assert` can be very useful but, to avoid confusion with the standard macro, please use a different name!

Summary

We have discussed what assertions are, how they behave, when to use them and, just as importantly, when *not* to use them. Finally we explored some more advanced issues such as extensions to the basic `assert` functionality.

You should now know enough to use assertions productively. Even making the basic transition from no assertions to employing the humble standard `assert`, can, I believe, bring a significant quality boost to the development process.

Garry Lancaster

Acknowledgements

Thanks to Heidi for her help with reviewing this article.

References

[1] Bertrand Meyer discusses design by contract: `http://archive.eiffel.com/doc/manuals/technology/contract/`

The second problem with this approach is that it only works where you know the complete list of exceptions which can be thrown. Given the lack of documentation which seems endemic in our industry it is all too easy to miss one.

What happens then depends on your runtime. The C++ standard simply says: "If no matching handler is found in a program, the function `terminate()` is called; whether or not the stack is unwound before this call to `terminate()` is implementation defined".

My environment produces the message "`Abnormal program termination`" which provides little help in finding the root cause of the problem.

I decided that the lack of useful information meant I was spending too much time debugging problems and so I was prepared to consider using some non-standard code to help me.

It is obvious from how the exception mechanism works that the runtime must know the type of the thrown object so it can match it to the appropriate catch handler. I decided that if I could get the runtime type of the exception many of my problems would be over.

Unfortunately this is not (yet) possible in standard C++. there have been a few discussions I have read over the years but, as far as I know, there still isn't a concrete proposal to provide this feature.

Of course at this point any Java or C# programmers reading this article are proably feeling pretty smug, since these languages (for a variety of sound reasons), provide significantly more information about exceptions than C++ does.

After some "poking around" in the compiler output and referring to some articles in MSJ by Matt Peitrek I discovered that I could write some Microsoft specific code to catch the underlying Win32 exception used to implement Microsoft's exception handling, and decode it to get hold of the type_info for the thrown exception.

So my new, highly non-portable code, looks like this:

```
#include <windows.h>
#include <typeinfo>
#include <iostream>

// Microsoft specific code
DWORD logException(_EXCEPTION_POINTERS*
                                pException) {
  // exception code for MSVC C++ exception
  static const int
              MsvcExceptionCode(0xe06d7363);
  // see EXSUP.INC for only public definition
  static const int MagicNumber1(0x19930520);

  PEXCEPTION_RECORD ExceptionRecord =
                pException->ExceptionRecord;
  if( (ExceptionRecord->ExceptionCode
                    == MsvcExceptionCode) &&
      (ExceptionRecord->NumberParameters
                                  == 3) &&
      (ExceptionRecord->ExceptionInformation[0]
                      == MagicNumber1) ) {
    struct link {
      link *chain;
    };

    // Hackery to get the type_info buried
    // in the data
    link *p = (link*)ExceptionRecord->
                    ExceptionInformation[2];
    p = p[3].chain;
    p = p[1].chain;
    const std::type_info *t =
        (const std::type_info *)(p[1].chain);

    std::cerr << "Uncaught exception: "
              << t->name() << std::endl;

    return EXCEPTION_EXECUTE_HANDLER;
  }
  return EXCEPTION_CONTINUE_SEARCH;
}
```

and the corresponding sample usage is:

```
#include <excpt.h>

int main(int argc, char **argv) {
  __try {
    return realmain(argc, argv);
  }
  __except(logException(exception_info())) {}
  return 0;
}
```

However, I have gone from the original code which used standard C++ features to some code which uses several Microsoft specific calls and, even worse, a number of undocumented 'magic numbers' and pointer hacks.

This raises a few questions, and motivated me to write this article.

1. Is it worth it?
2. How do I encapsulate the trickery?
3. What can I do about maintainability?

Many people may feel the first question is all that needs asking, and answering with a resounding NO. However, in this case I felt that the gain for me of getting out the name of the object being thrown was greater than the ugliness of the solution. This is a judgement call, and each problem like this presents different trade-offs.

There is also a separation between code used for development and code shipped in production systems. I would be much less happy to ship a product which relied on this trick for normal operation, but if something goes wrong handling an uncaught exception I'm not really much worse off.

Encapsulating the 'trickery' is important for two reasons.

Firstly, I want to be able to write some test harnesses for the trick code which can be run whenever an upgrade is installed, or other compiler settings are used, etc.

Secondly, if I find the code fails in some as yet undiscovered way I want to have a single place to fix.

Thirdly, I want to hide the code so I can 'forget' this dangerous knowledge.

Hence I want to write this function inside a separate C++ source file. I would also add checks for the preprocessor symbol for the compiler to check it matches one of the versions I have tested the code for. This ensures I explicitly check the code when I move to a different compiler level, and also prevents me accidently trying to use this code with a non-Microsoft (in this case) compiler.

The big problem with non-portable code is maintainability. This is a problem for many reasons.

First of all I have used undocumented features of the compiler and so my chances of support if anything goes wrong is almost zero. Since I am using such features I am relying on my guesses about the structures being right. It is quite likely my guesses are incomplete, causing the code to break under some conditions.

Then I have a vulnerability to the future - I am at the mercy of the compiler writers who may well change this sort of internal detail with a subsequent release of the compiler (or even with a patch release). Will it be possible for me to change my trick to cater for the future structures? Will I still be around to do this work by then, and if not is there anyone else who could?

The best I can do to reduce this vulnerability is to ensure the function is documented to use non-portable code and provide some documentation about the 'detective work' which went in to finding this solution. A good set of test cases exercising the function in as wide a variety of cases as possible and compiled with as many versions of the compiler I can find helps greatly with ensuring the robustness of the solution and also provides a ready made test bed for future compiler releases.

Having made the function a separate compilation unit, as described above, also makes it easier to write stand-alone test cases.

I hope this quick run through the issues I thought about when writing this non-standard piece of code may help you when you find yourself facing a problem which you can't seem to fix using the standard set of features.

Roger Orr

Python Section

A Python Project

Silas S. Brown <ssb22@cam.ac.uk>

This article describes the Python code in a small project that I've written lately. I wrote this article when I heard that C Vu was short of articles. It is a little rushed. But it might be a useful "dive in" introduction to Python for programmers experienced in other languages, as well as being a starting point for discussion. Some of this code could be written better.

My program was designed to help me increase my vocabulary in a foreign language, by playing audio samples to me. It is entirely non-interactive. It plays an English word, then pauses while I'm supposed to remember how it's said in the other language, then plays the foreign word so I can check what I've remembered. The tricky part is I wanted new words to be repeated more often than old words, in graduated intervals (i.e. intervals that start small and increase). That meant that each lesson needs some planning.

I decided to represent the lesson as a list of events. Let's have a class called `Schedule` that holds a sorted list of (start,finish) times:

```
class Schedule:
    def __init__(self): self.bookedList = []
    def book(self,start,finish):
        self.bookedList.append((start,finish))
        self.bookedList.sort()
```

We'll leave it at that for now (no functionality to check overlaps etc yet).

Now, there are two main kinds of events - actual events that play sound, and events that link other events (I called this "glue", from TeX typesetting terminology). For example, two repetitions separated by an interval of time will be represented by two (clusters of) events separated by a glue event. The glue event is invisible to the booking of other events (i.e. other things can happen in the lesson while the glue is taking place) but it is still there, ensuring that the delay between the two repetitions is of an acceptable length. (The glue will tolerate a certain amount of stretching or shrinking in order to make everything fit.)

Here is a class containing some functionality for both events and glue. The parameter '`invisible`' is non-zero if it's glue and other events can take place at the same time; the parameter '`plusMinus`' gives the tolerance for stretching/shrinking. Note that the length is given in the constructor, but the start time is not given until the methods are called. This is because we won't know the start time when we're creating the event; we need to find a time where it will fit in.

```
class GlueOrEvent:
    def __init__(self,length=0,plusMinus=0,invisible=0):
        self.length = length
        self.plusMinus = plusMinus
        self.invisible = invisible
    def bookIn(self,schedule,start):
        if not self.invisible:
            schedule.book(start,start+self.length)
```

The `bookIn` method reserves time on the schedule. This is used later when checking for overlaps, so invisible events are not booked in.

```
    def addToEvents(self,events,startTime):
        assert not self.invisible
        events.append((startTime,self))
```

The `addToEvents` method adds this event to a list with its start time. This will be used later when calling the Python schedule library to play the lesson. It will be overridden by composite events which add each of their constituent events to the list.

Now for an overlap check. The following method is designed to return a value indicating how much this event has to move in order not to overlap with anything else on the schedule. It is passed a parameter indicating what direction it should move in (`+1` or `-1`).

```
    def overlaps(self,start,schedule,direction):
        if self.invisible: return 0
        if not schedule.bookedList: return 0
        count = 0
        # Skip over all events that finish before we start
        while schedule.bookedList[count][1] <= start:
            count = count + 1
            if count >= len(schedule.bookedList): return 0
        # Does this event start before we finish?
        if schedule.bookedList[count][0]<start+self.length:
            # We have an overlap
            if direction < 0:
                # Make sure we finish before it starts
                backwards = start+self.length-schedule.bookedList[count][0]
                return self.overlaps(start-backwards,schedule,direction)+backwards
            else:
                # Make sure we start after it finishes
                forwards = schedule.bookedList[count][1] - start
                return self.overlaps(start+forwards,schedule,direction)+forwards
        return 0 # No overlap
```

Finally, a `play()` method which will be overridden later.

```
    def play(self): pass
```

Now we can create some subclasses, such as `Event` and `Glue`:

```
class Event (GlueOrEvent):
    def __init__(self,length):
        GlueOrEvent.__init__(self,length)
```

```
class Glue (GlueOrEvent):
    def __init__(self,length,plusMinus):
        GlueOrEvent.__init__(self,length,plusMinus,1)
```

A CompositeEvent is an event made up of several others in sequence with nothing intervening (no glue).

```
class CompositeEvent (Event):
    def __init__(self,eventList):
        len = 0
        for i in eventList: len = len + i.length
        Event.__init__(self,len)
        self.eventList = eventList
    def addToEvents(self,events,startTime):
        for i in self.eventList:
            i.addToEvents(events,startTime)
            startTime = startTime + i.length
```

Now we'll have a class called GluedEvent (i.e. an event with some glue before it), which has functionality to adjust the glue (stretch/shrink it within its tolerance) in order to get the event to fit properly. This needs an exception, which can be declared as an empty class:

```
class StretchedTooFar: pass
```

Here is the GluedEvent class. Note that there is functionality both for random adjustment (using a Gaussian distribution) and for adjustment to avoid an overlap. It is written in such a way that it is possible to call the adjustment code more than once with different directions.

```
class GluedEvent:
    def __init__(self,glue,event):
        self.glue = glue
        self.event = event
        self.glue.adjustment = self.glue.preAdjustment = 0
    def randomPreAdjustment(self):
        if self.glue.length < randomAdjustmentThreshold: return
        self.glue.preAdjustment = random.gauss(0,self.glue.plusMinus)
        if abs(self.glue.preAdjustment) > self.glue.plusMinus:
            self.glue.preAdjustment = self.glue.plusMinus # err on the +ve side
    def adjustGlue(self,glueStart,schedule,direction):
        needMove = self.event.overlaps(glueStart+self.glue.length+self.glue.preAdjustment,schedule,direction)
        needMove=needMove*direction+self.glue.preAdjustment
        direction=sgn(needMove) ; needMove=abs(needMove)
        if needMove > self.glue.plusMinus \
           or glueStart+needMove*direction < 0 \
           or glueStart+self.glue.length+needMove*direction+self.event.length > maxLenOfLesson:
            raise StretchedTooFar()
        self.glue.adjustment = needMove * direction
    def getAdjustedEnd(self,glueStart):
        return glueStart+self.glue.length+self.glue.adjustment+self.event.length
    def bookIn(self,schedule,glueStart):
        self.event.bookIn(schedule,glueStart+self.glue.length+self.glue.adjustment)
    def getEventStart(self,glueStart):
        return glueStart+self.glue.length+self.glue.adjustment
```

Now for a function called setGlue which "sets" (adjusts) the glue for all the events in a list. This function needs to backtrack when a StretchedTooFar exception is raised, and try a different way of setting the glue. (Don't forget that we can always try another direction.) If there is no solution at all then the StretchedTooFar exception will be raised by the function itself.

```
def setGlue(gluedEventList, schedule, glueStart = 0):
    if not gluedEventList: return
    try:
        gluedEventList[0].randomPreAdjustment()
        gluedEventList[0].adjustGlue(glueStart,schedule,1)
        setGlue(gluedEventList[1:],schedule,gluedEventList[0].getAdjustedEnd(glueStart))
    except StretchedTooFar:
        gluedEventList[0].adjustGlue(glueStart,schedule,-1)
        setGlue(gluedEventList[1:],schedule,gluedEventList[0].getAdjustedEnd(glueStart))
```

There is a problem with the above algorithm. It fits the first event into the first place it will fit, and then it will try to fit the rest of the sequence based on the position of that first event. If it fails to fit the rest of the sequence, it will fail completely; it doesn't ever try to fit the first event in a different place (later). I hacked around this with a wrapper function; it's a bit rushed but it usually works (hey this is Python):

```
def setGlue_wrapper(gluedEventList, schedule):
    sillyOffset = 0 # seconds
    worked = 0
    while (not worked) and sillyOffset < maxLenOfLesson:
        try:
            setGlue(gluedEventList, schedule)
            worked = 1
        except StretchedTooFar:
            # try harder
            sillyOffset = sillyOffset + 10 # *** needs to be a constant
            gluedEventList[0].glue.length = sillyOffset
    if not worked: raise StretchedTooFar()
```

So now we can write a function that will book a list of `GluedEvent`s into a lesson, adjusting the glue as necessary:

```
def bookIn(gluedEventList,schedule):
    setGlue_wrapper(gluedEventList,schedule)
    glueStart = 0
    for i in gluedEventList:
        i.bookIn(schedule,glueStart)
        glueStart = i.getAdjustedEnd(glueStart)
```

And now the lesson itself, including the function to play the events (I've cut this down a bit - taken out all the diagnostic messages and the functionality to write the lesson to a sound file as well as play it through the speakers - I don't know how big this article is getting)

```
class Lesson:
    def __init__(self):
        self.schedule = Schedule()
        self.events = [] # list of (time,event)
        self.newWords = self.oldWords = 0
    def addSequence(self,gluedEventList):
        bookIn(gluedEventList,self.schedule)
        glueStart = 0
        for i in gluedEventList:
            startTime = i.getEventStart(glueStart)
            i.event.addToEvents(self.events,startTime)
            glueStart = i.getAdjustedEnd(glueStart)
    def play(self):
        self.events.sort()
        runner = sched.scheduler(time.time,time.sleep)
        for (t,event) in self.events:
            runner.enter(t,1,play,(event,))
        runner.run()
def play(event):
    event.play()
```

Now we need a type of `Event` that will actually play something:

```
class WavEvent(Event):
    def __init__(self,file):
        self.file = file
        header = sndhdr.what(file)
        if not header: raise IOError("Problem opening wave file")
        (wtype,wrate,wchannels,wframes,wbits) = header
        divisor = wrate*wchannels*(wbits/8)
        o = open(file,'rb')
        fileLen = len(o.read())
        o.close()
        seconds = math.ceil((fileLen + 0.0) / divisor) # approx
        Event.__init__(self,seconds)
    def play(self):
        if winsound: winsound.PlaySound(self.file,winsound.SND_FILENAME)
        else: os.system("play %s" % (self.file,))
```

And now all we have to do is make those `GluedEventLists` in a sensible manner. I'll leave that to the next issue.

Silas S. Brown

Write for ACCU!

Pete Goodliffe <pete@cthree.org>

We need members to write articles. And that means you! It's the way the ACCU works. If you enjoy reading C Vu and Overload, and want to see them continue then please get writing something. Don't rely on the same few people. You have something you can share. Really, you do.

And now there's an added incentive. 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 spangly new ACCU awards.

Prizes! Prizes! Prizes!

This year the ACCU is offering awards 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. Never written an article before? Get going!

What to write?

Here is a small selection of suggested titles. They've been published a few times, but have been specifically asked for by ACCU members. Please look at the list and consider if you can write something on a topic.

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

Don't let this list constrain what you write. What are you doing right now? What do you know about? Please write something about this for the ACCU journals. You can write on any topic that will be of interest to ACCU members, language features, tools, libraries, problems you've solved...

Don't hold back

We really need articles, and at all levels. 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.

Reviews

Bookcase

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

Francis Glassborow writes:

One feature of being 'Book Review Editor' is that I get to notice which publisher's books disappear out to reviewers most quickly and which hang around until, in despair I try to do the job myself (normally I try to catch the real lemons and review those. I will also review titles I really want to place on my own bookshelf).

Two publishers very definitely have a lot of books within our general subject areas that fail to attract reviewers. Perhaps that is, in itself, a comment on their brand image. WROX Press used to be popular among our volunteers but much less so over the last couple of years. A! Apress is relatively new and seems to be having difficulty in attracting our reviewers to look at their titles.

Does this matter? I think so because it suggests that the dominance of some publishers' brand images is causing problems elsewhere. Those of you familiar with the structure of books on computing may realize that Pearson Education (owners of Addison-Wesley, Prentice Hall etc.) is close to a monopolistic position. While the constituent parts were being run independently that was not a big concern. However I have recently noticed a blurring of the edges with books from one 'imprint' carrying an ISBN from another.

Excellent though Addison-Wesley is as a publisher of books on C++, I think we need to start sounding warning bells.

Incidentally, that was one of my reasons for taking my book to Wiley, even though Addison-Wesley expressed very strong interest. O'Reilly was the other publisher I considered, but I marked them down heavily because I do not think they are currently applying sufficient quality control and selectivity.

Francis

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

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

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

C & C++

C++ All-in-One Desk Reference For Dummies by Jeff Cogswell (0 7645 1795 3), Wiley, 826pp+CD @ £29.95 (1.16)
reviewed by Francis Glassborow

I was amazed to discover that C++ for Dummies outsells the total of the next three titles aimed at newcomers to C++. Of course this is good news for me because it means that there really are many people out there who believe they know very little but want to learn to program in C++.

Now the title of this book might suggest that it was for the person who has already learnt C++ but wants a good reference volume in which they can look up the things they only vaguely remember. Wrong!

This book is largely a classical introduction (i.e. it does things like using raw C-style arrays for 500 pages before it introduces you to `std::vector` (and the author forgets about the `std::` part so the code from the book will not work as written.) to learning C++. On page 1 we find:

> This book is not a big rant about C++. Rather, this is a hands-on, roll-up- your-sleeves book, where you will truly learn C++.
>
> At the very beginning, I start you out from square one. I don't assume any programming experience whatsoever. Everybody has to start somewhere. You can start here. Not to brag, but you are in the hands of a highly successful C++ user who has shown thousands of people how to program, many of whom started out from square one.

On the positive side, the author's understanding of C++ is streets ahead of Herbert Schildt's but that is not saying much. On the negative side much of his code is littered with bad things such as magic numbers, and dynamic resources handled by raw pointers. He actually writes about magic numbers on page 198 but does little to avoid them in his own code.

The code in the book is also littered with typos. One of the commonest is using '>' where '>>' is needed. Indeed this is so common that I wonder if there is some typesetting problem shooting his code.

While I am greatly in favour of getting students to type in source code and thereby learn the kind of errors that result, I am not in favour of presenting novices with large numbers of lines that simply send text to the screen. Learners have better things to do with their time.

Had this book been written and published six or seven years ago I would have been praising it because the author makes a good job of introducing C++ as it was in the past. He has even acquired new skills by learning about such things as UML and the STL. What he has not done is to re-appraise his teaching method in the light of such things as the STL, exceptions and namespaces. Most of his code will not compile with a modern C++ compiler because he does nothing to handle the namespace problem. He does not appear to appreciate the value of introducing exceptions early so as to encourage the programmer to validate input. He still pursues the traditional approach of introducing all the complexities of arrays and pointers long before using `std::vector`.

In other words his teaching/presentational method has simply accreted new bits without going through a fundamental review to see how Standard C++ could be used to help the newcomer get started with the minimum of complexity.

Finally, I find almost all his code examples crushingly boring. I simply cannot imagine why I would want to spend time as a novice typing in his code examples.

Not a bad book but it could have been so much better, and those interested in learning to program in C++ deserve much better. And perhaps it would be better not to try and write a book that tries to tackle so many different things.

CodeGuru.com Visual C++ Goodies by Nigel Quinnin/ Subject Editor (7897-2777-3), Que, 600pp+CD @ £28-99 (1.38)
reviewed by Francis Glassborow

If you program for windows you should be familiar with `www.codeguru.com`. If you are not then you are missing a useful resource that could reasonably save you more time than it takes to keep an eye on what material has been posted there. Of course if you do check that site you will already know about this book and will probably have already decided if it is a suitable addition to your reference library.

The book is a collation of material from that site. As you might expect, much of the C++ source code is below the quality of what I would wish to see held up as an example. Right near the start we have a brief (yes ridiculously brief) introduction to what the collator describes as C++ essentials. In there you will find very little but even then we have flawed code. For example on page 11 we find:

```
class CcolorLine : public
Cline {
public:
   void Draw(long color);
};
```

Leaving to one side possible problems with deriving from a class that does not have a virtual destructor we have a second problem in that the base class has multiple constructors which all those reading this will know are not inherited so the authors claim:

> Well, with this class we have all the functionality of our other class, but now we can use this other `Draw()` function that allows us to set the color.

That section is littered with raw pointers to dynamic objects and very little if any of the C++ source code in the book is exception safe, even at basic level. However much of the code isn't even C++, but is scripting code to program the Visual C++ IDE. All very useful to those that need it, but why not just place it on the CD, reduce the page count and focus more on explaining?

This book has little to offer those that do not use Visual C++. It also represents the quality and understanding of C++ that the above average user of VC++ has. However, many of those reading this will guess that I do not think that says very much. Their expertise with using VC++ is unquestioned, their understanding of how to get the IDE to work for them is excellent but the quality of their C++ is poor even when allowing for the things they must do to meet the needs of programming Microsoft Windows in a GUI, multi-threaded environment.

In general I think most people in the target audience would get better value by using the electronic source of the material. That way they could choose what they looked at from all that is available rather than rely on the selections of a single editor.

I have to admit that I am curious as to who is getting paid royalties for this book. I am also curious as to whether all those whose work is included gave permission for it because I cannot find any copyright information in the front matter.

Definitely not a book for anyone not using VC++, and I am less than enthusiastic for those that are. It may help you with some problems but it will not improve the quality of your work.

C Programming in Easy Steps by Mike McGrath (1-84078-203-X), Computer Step, 192pp @ £9.99

reviewed by Pete Goodliffe

This book has not been sent to the ACCU to review. However, a friend of mine was given it for Christmas and after looking through it I thought that it was worth commenting on. It does appear very interesting, there aren't that many C books being published any more. But more than this, the price is only £9.99.

At first I was pleasantly surprised. There's a lot going for it, not just that it is cheap. It's generally well written, with an even tone and neat layout, making good use of call outs to draw attention to important points.

It's short and concise, containing no extraneous waffle. Each item is explained once in sufficient detail. It is "cross platform", with examples for Windows and Linux. Indeed a lot of example screen shots are shown of terminals running the example programs on both platforms. Up front the book describes where to get gcc for linux and mingw for windows. Mind you, it's a shame that it never suggests a better text editor than notepad!

It's a shame to see a number of simple items missing that would have greatly improved the book: there are no exercises to test understanding, and it would have been beneficial to include some pointers to more information. Although there is a "Where to go next" section at the end with the Borland/MS/free Bloodshed IDEs mentioned, there are no other books, websites, or other sources of information recommended for the curious reader.

Up to here it would have been an "adequate" book. But unfortunately, it contains a number serious let-down features. There really isn't much technically inaccurate information, but a lot of misleading and incomplete content.

This ranges from missing simple things like a clear differentiation of "==" from "=", through only ever showing `if`/`for`/`while` statements in their form with braces (not wrong but something that will clearly confuse a newcomer when looking at other people's code), to only describing one form of main: `void main()`.

A section on multiple source and header files is dangerously misleading. In the example it shows a header file that actually contains the implementation of a function. Then in each `.c` file this function is prototyped separately. I really can't understand what the author was thinking.

The book proudly claims it claims it "conforms to ANSI C" on the cover, and there's another problem. There is no discussion of C standards. It happily describes an older variant of the language. It also describes some perspectives that aren't really applicable to 21st century C programming. For example it carefully explains the register keyword when these days the compiler is going to pick the best place for each variable regardless. You are left with the impression this is an old text on C re-presented. This really isn't excusable in a 2002-dated C book, it belongs more around the early 1990s.

So this could nearly be a recommendable book, especially considering the price, but it misses the mark. Perhaps this is what should be expected from a book with a smaller budget – in some ways I was impressed by how good parts of the book are. A subsequent edition could bring great improvements.

Unfortunately, the newcomer at whom this book targeted (with the low price) will not appreciate the flaws. As a case in point, after trying to explain to my friend why this book has serious issues, he thought I was being unnecessarily pedantic and elitist. He thinks these things don't matter. He wants a cheap book to teach him to program at home – he has no intention of getting into the programming industry. But this book could also mislead a set of novice C programmers who do have their intentions set on a career in programming. This is a serious issue and shows the level of responsibility technical authors have.

C#

C# and the .NET Platform by Andrew Troelsen (1 893115 59 3), APress, 970pp @ £42.99 (1.39)

reviewed by David Nash

This thick book has 900 pages and is designed to bring existing VB, C, C++ and Java programmers up to speed on C# and the whole .net platform, as indicated by the title.

It begins, as seems to be common with books on this subject, by describing the philosophy of .NET. Chapter 1 contains a basic introduction and the whys and wherefores of .net. In order to justify the existence of C# and .NET, the author seems to be trying to find something wrong with every other language. For example, he says that 'Programming with C++ remains a difficult and error-prone experience', although the only problem he could find with COM is 'very complex'.

Since the readers of this book are expected to be programmers already, chapter 2 runs very quickly over the core language and the C# fundamentals such as Objects, data types, the difference between value and reference types, UDTs, overriding methods, looping and flow control constructs, operators, basically all the core C# language.

In chapter 3 we meet the Object-Oriented features of the language, i.e., class-related stuff like inheritance and support for polymorphism. Comparisons are made with C++, Java and VB, which is helpful, although if you are an experienced OO programmer you won't need the description of the 'Pillars' of OO - Encapsulation, Inheritance and Polymorphism. The next couple of chapters build on this and introduce more of the object-oriented features.

Following these the book moves on to .NET features, describing assemblies and attributes, then a few chapters on windows rather than plain 'console' features. The book refers to Visual Studio and other tools where appropriate but also tells you how to code Windows forms, for example, by hand.

Finally the book describes some optional parts of C# programming, with chapters such as 'Interacting with unmanaged code', 'Data access', 'Web development' and 'Building web services'.

The reviewed copy sported a flash on the cover stating it was for Beta 2 of .NET, although the illustration of the book on the publisher's web site showed no sign of this, as one would expect them to have updated it by now. Source code for the examples in the book is also available from the web site.

Generally this book was very easy to read, with a friendly style that came across well. I reviewed it at the same time as Wrox 'Professional C#' and would add that this book is easier to handle, at 'only' 970 pages. The examples in this book are necessarily trivial, as with most programming tutorials and at times a bit silly ('Bitmaps as brushes! Way cool...'), but generally they illustrate the point being made.

Professional C# 2ed by Simon Robinson et al. (1 861007 04 3), WROX*, 1222pp @ £43-99 (1.36)

reviewed by David Nash

I reviewed this book at the same time as reading 'C# and the .net platform' by Andrew Troelson. That book has nearly 1000 pages, but this one takes the prize with a weighty 1233 pages. In contrast to that book though and in common with most Wrox books, this book has multiple authors: in fact it has 14. So it's not difficult to see how they came up with so many pages.

It begins like most C# books seem to, describing the architecture and rationale behind

C# and .net. Unlike Troelson's book they don't seem to think they have to find shortcomings of all the other main languages out there in order to justify C#.

The book moves on to describe most features on C#. It starts, as is usual, with an analysis of the 'Hello World' program. Following that come all the usual things like types, control structures, console I/O and also includes XML documentation at this early stage, which I feel is good as it makes the programmer think about documenting the software!

Next comes OO stuff (as it relates to C#, not generally) including classes/structs (and the differences), inheritance and interfaces, method overriding, constructors and destructors, properties etc.

All this has appropriate comparisons with C++ where necessary.

The next chapter finishes off the general C# part of the book with what they consider to be 'advanced C#' including exceptions, delegates and events, user defined casts, the preprocessor and such like.

After this the book moves on to more general .NET stuff plus various extras usable from C#. It begins with the .net base classes, which are cross-language, then we encounter the first chapter that relies on Visual Studio .NET (the previous ones used the plain SDK) showing how to develop windows apps in C#, including custom controls.

Following these chapters we learn about web services, SOAP and remoting (like Java RMI).

Really this book has it all. It is easy to read, despite being written by about 14 different authors and just about everything seems to be here. It is not organised as a reference though, but a tutorial and I would have preferred that, if a book is going to have this many pages, that some of them could be used to include some material in reference format. As it is the only real criticism I have is the sheer size of the book, which makes it physically more difficult to read.

GDI+ Programming in C# and VB.NET by Nick Symmonds (1 59059 035 X), Apress, 589pp @ £42-99 (1.39)
reviewed by Max Palmer

This book aims to provide an introduction to programming GDI+ in both C# and VB .NET. The author approaches this task by splitting the book into two parts. The main part of the book explores four key areas of GDI (graphics device interface) programming — vector graphics, images, text/font handling and printing, while the final two chapters try to pull these areas together through the development of two relatively simple applications. There is also a good introduction to .NET and a brief overview of GDI at the start of the book for those that might be unfamiliar with either.

One of the more unusual aspects of the book lies in its language neutral approach to explaining GDI+. The author works through a series of examples in each chapter, nearly always providing 'equivalent' example code in both VB .NET and C#. While this is interesting for those who want to see how both languages are structured and deal with the .NET framework, it has the obvious drawback that a fair amount of time is spent explaining the same concepts in both languages. While repetition can be good, especially when dealing with new concepts, I couldn't help feeling that this dual approach diluted the GDI content. However, the title makes it clear that this is to be expected, so the reader shouldn't be too surprised.

Overall, the book is well written, nicely presented and the concepts that are covered are clearly explained. Not being familiar with GDI+, it is difficult to say whether any areas have been neglected, but what is covered is dealt with well. There is a lot of example code included in the text and projects are also available for download from the publisher's web site. Occasionally I felt the author could have used more meaningful variable names in the code, but other than that it was clear. The book adopts a hands-on approach and it would suit someone who is relatively new to .NET and graphics programming. However, if you have more experience, you might find the pace a little bit too slow and there was too much code (and repetition of code) present in the text for my liking.

Learning C# by Jesse Liberty (3-596-00376-5), O'Reilly, 354pp @ £24.95 (1.40)
reviewed by Paul Johnson

The C# language has its heritage in C++ and Java and if you are familiar with either (or both) of these languages, you should be able to pick up C# without a problem. If you are a linux user, then you can still make use of the book as just about every example is compilable using mcs/mono from Ximian Inc.

If you are new to OOP, then you will have problems as the description of OOP and its principles are not clearly explained. A newcomer would be better off with a basic OOP book in either Java or C++, then come back to this to learn C#.

The book is well written and very clear in both writing style and chapter objectives. The only problems come in some very important parts, which are not fully explained. For instance both `void Main()` and `int Main()` are used, but it is never explained when you would use the differing return types. Fortunately, this sort of problem does not occur too often.

The additions to the language over C++ and Java are well described and make the basis of OOP a lot simpler. One of the simplest, yet best ideas is that all variables have to be initialised — the compiler will not compile any source where a variable is not initialised (unless the `out` modifier is used). Passing by reference is made clear by use of '`ref`' instead of &

The main problems I have with the book are three-fold.

1. The book describes itself as being .NET programming. While C# is designed for .NET, the book itself really does not cover .NET to justify being called a .NET programming book
2. The method classes are not very clearly described. Descriptions are very minimal and unfortunately, there are not enough examples of methods to show how to properly use the methods
3. The book is not very well organised. The first part is very basic, the second part covers OOP with just a chapter on the VisualStudio .NET debugger separating them (why that is in the book, I don't really know!). The only problem is that the coverage of aspects that really could have done with being in the first part is covered right at the end of the second part.

Unusually for a programming book, there is a comprehensive appendix giving definitions to C# keywords. It is more usual to just have a listing of the keywords. I suppose that it is because the language is new.

Overall, I would give this a recommended rating. For £25, it is a fairly good introduction to C#.

C# for Experienced Programmers by Deitel & Deitel et al. (0 13 046133 4), Prentice Hall, 1382pp @ $54.99
reviewed by Peter S Tillier

As I am a full-time mainframe/USS/UNIX/PC middleware programmer using various languages I was interested in reviewing this book for one main reason; its relevance to my job.

I don't generally like books of this size. Nearly 1400 pages is rather too many for my taste and suggests that the authors may have tried to put in too much material. In this case my feelings are confirmed, it isn't the sort of book that you can dip into while commuting - it wouldn't even fit in most briefcases.

As an experienced programmer who has used many different languages, I asked myself before reviewing it what would I expect from a book entitled 'C# for Experienced Programmers'. My answer was that I would expect that a book that showed how C# differs from, or is similar to, other languages that I know or have used before. Such a book does not necessarily need to be either long, or to contain many 'run-able' examples. What it does need to do is to highlight the syntax of the language and, in particular, any areas where this is unusual when compared to other commonly used languages such as C, C++ and Java. I have mentioned C, C++ and Java because they are modern language implementations that are likely to have been used by most of the intended audience for this book. Older languages such as COBOL, FORTRAN and PL/I are rather different in style and programmers who are only experienced in these would need, in my opinion, a different type of book.

What this book provides is not what I would have expected. In fact I have to ask myself just what do the authors think 'experienced' means? There are about 150 pages describing Object-Based and Object-Oriented Programming with copious example code and coverage of how to use an object browser. A further 100 pages describe GUI concepts with yet more code and pretty GUI windows. So there are 250 pages describing concepts that anyone who claims to be an experienced programmer in any sort of environment must, at least, have been aware of and hardly needs a tutorial such as this to show them the way forward.

It gets better! There are 50 pages about XML and DOM: more concepts that any programmer worth their job would be aware of, especially if they are experienced. What next? Database concepts and SQL - well no experienced programmer would know about those, of course. The appendices are worse - how many experienced programmers need to know about number systems, job opportunities, HTML and XHTML?

If you are getting the impression that I didn't like this book you are right - but only because it not a book for what I would call experienced programmers. They simply don't need the masses of code examples and in depth explanations that it provides.

Deitel and Deitel ought to decide what this book is about. Is it about C# or the .NET platform? Is it about XML or HTML or XHTML? Or maybe it is intended for ex-mainframe programmers, in which case I don't think that one book, even of this size, would be adequate. Somewhere in the introduction or the book jacket the target audience should be made clear.

I think the book might be better titled 'All you ever wanted to know about C#, .NET and then some', but even then it fails in some ways. I wouldn't recommend it to anyone that I consider an experienced programmer. No doubt some people will like it and buy it - I wouldn't have looked at it in a bookstore just because of the size. It's handy as a doorstop, though. Not recommended.

The Complete Idiots Guide to C# Programming by David Conger (0 02 864378 X), Alpha*, 346pp @ £14-99 (1.33)
reviewed by Francis Glassborow

Those of you familiar with the general form of a *Complete Idiot's Guide* will know what to expect from this book. This one sets out to teach the complete novice how to program in C# but actually teaches them C# while avoiding teaching them to program. This is a common failing. Authors confuse the process of writing syntactically correct source code with the process of writing a program. All that source code does is to express your solution to a problem in terms of a computer language. That is the easy part. The hard part is teaching the newcomer how to crystallise their ideas into a form that can be expressed in a programming language.

By the time you have finished reading this book you will have a reasonable idea as to how to write syntactically correct C#. You will have reached, in programming terms, about the equivalent of being able to write 'The cat sat on the mat." The unfortunate thing is that you do not need the whole panoply of a modern computer language in order to write a novel program that means something to you. Indeed much of the mechanism of a modern language gets in the way of the newcomer trying to do just that.

I was recently discussing the difference between a book aimed at teaching a newcomer to program and one aimed at teaching someone a new language. One of the participants commented that the trouble with newcomers is that they kept wanting to know what something was good for, while those who already know another language keep wanting to know why you do not do it that way. I think many authors would be well advised to think carefully about that.

As I worked through this book, I kept hearing the ghost of a student muttering 'But what use is it? What problem does it solve?' The fundamental problem with this book is summarised by the front cover where it says:

Expert advice for both beginners and advanced programmers.

The style of writing would drive the average advanced programmer up the wall. On the other hand breadth of information would simply leave the average newcomer mired and confused as to how to write a program that did something that was actually useful.

If you have some knowledge of programming and have written successful programs of your own design in at least one other language but you do not consider yourself an experienced programmer then this book might work for you. But if you really have no programming background you will need to look elsewhere and if you really are an advanced programmer you would not appreciate the style of this book.

Java

Mastering Enterprise JavaBeans 2ed by Ed Roman et al (0 471 41711 4), Wiley, 640pp @ £33-50 (1.34)
reviewed by Christer Lofving

Two years ago I purchased the first edition of this title about maybe the hottest Java technology right now, that of Enterprise Java Beans. The number of pages has almost doubled with this edition, which is justified by the additions to the technology. The previous edition covered the EJB 1.1 standard, which in fact most real world applications are built upon. EJB 2.0 comes along with fresh new features like a local fast performance interface and message driven beans.

If you are unfamiliar with J2EE you aren't given any soft start. Instead you are from the first page thrown head-first into this complex technology and to get the most out of the text you need to be at least a little familiar with Java beans, J2EE or any similar technology.

At first glance the book gives the impression of being a read-through, but it isn't. It is divided in many different parts, chapters and appendixes, each of them suitable to be read on their own and there is a useful introductory text to describe the organisation of chapters and when/why to read them.

Part two, 'The triad of Beans' is the core of the book and explains Enterprise Java Beans in such a great way that it is worth the price of the book by itself. The extra dark squared text boxes on the pages are small gold mines. They delve deeper into key points and hard bits, such as JDBC and the peculiar handling of exceptions in EJBs. Illustrations like schemas describing a bean's life cycle are clear and easy to read. The appendixes cover, among other things, the API in a foreseeable way, deployment descriptor reference and Corba interoperability.

The mysteries of the Environment Naming Context, a vital part of EJB technology, are explained in a way that really makes sense in one of these separate appendixes. One can say you are growing with this book. It is suitable for the advanced programmer as an introduction to the EJB world and also in daily work and as a long lived reference. The code samples are good and complete, but how are they supposed to be used? One of the special things with EJB technology is that a lot of work is done by the abstract so-called application 'container'. This container is handling issues like transactions and pool handling almost transparently for the developer. Therefore you need support of a complete J2EE application server to run even the simplest hello world code.

As always, you need to compile and run the source code to really understand it. You need both the JDK and J2EE (Java 2 Enterprise Edition) downloadable from `Java.sun.com`, but also a J2EE server. Sun have their own reference application server, but I recommend Blazix for this and other tutorial purposes. (`www.Blazix.com`). It is a small but fully J2EE compliant application server which comes with a complete tutorial. Very suitable for the purpose to test out this books and other sample code in real work. Blazix has free trial versions for both Windows and Linux, is easy to set up and has an amazingly small footprint.

I've saved the best thing until last. You may also visit the excellent Web site `www.TheServerSide.com`. Among other goodies you are able download this complete reviewed title as a PDF-file. For free!

Java 2 By Example 2ed by Jeff Friesen (0 7897 2593 2), Que, 830pp @ £25-50 (1.37)
reviewed by James Gordon

This is a nice smart looking book with enough white space so that it is easy on the eye. It has icons in the margins for examples and output, I especially like to see the output from a program. It starts with a section describing Java (452 pages worth) and then steps into the second part of the book, which is the APIs.

This book is aimed at the beginner in programming for Java, with clear, short, simple examples to follow. Luckily the examples are downloadable from the quepublishing web site as there are almost 200 examples. Except for a very large example in the beginning the examples are short, uncluttered and complete. They contain the output from running the code, which other books stupidly miss.

The examples explain things ranging from Class Instance Initalisers, nested classes, inheritance, scheduling of threads, etc. in the first part of the book. Then in the second part of the book it goes into the API and shows examples of collections like lists, hash sets and iterating over them. It also includes streams, files, random numbers and other maths examples. It seemingly covers all of the standard API classes with enough to be able to help yourself to the other, lesser used, classes.

This looks like a very good book with a good index, plenty of examples and if you like them, additional things to do at the end of each chapter, with the answers to the additional question in the book. Nothing worse that trying to figure out where you have gone wrong without the answers. It covers the latest Java 2 SDK 1.4.

I've enjoyed reading this book as it has taught me about Instance Initialisers and creating classes using Interfaces, both things I need for some of my projects. If you have just started, or struggling to get off of the ground with Java, then this book will certainly help with that initial learning curve.

Great Ideas in Computer Science with Java by Biermann & Ramm (0 262 02497 7), MIT, 528pp @ £29-50 (1.53)
reviewed by Mathew Davies

This book introduces the reader to a selection of ideas from computer science, encouraging him or her to experiment with these ideas using simple Java programs. In just over five hundred pages, the authors conduct a whistle-stop tour of web pages, Java applets, software engineering and development, OOP, simulation, machine architecture, assembly language translation, security, communications, parallel computation, non-computability and artificial intelligence. As I said before, it's a whistle-stop tour.

This book is unarguably very interesting. However, I'm not sure about the intended audience. Indeed, the authors' assumptions about the reader seem to change as the book proceeds. For example, the opening chapter explains how to assemble some basic web pages, using HTML. The explanation here seems to be aimed at someone who's perhaps never programmed before and needs a bit of hand-holding. Yet when it comes to getting connected to the Internet, the authors suggest that 'You may need to get help from a friend to get started ...' Similarly, when the authors expand their web pages to invoke Java applets, the reader is offered the following advice on setting up Java on their machine: 'You may have to get some help in adapting to your specific situation.' What's more, when I tried typing in and running a few examples of Java source code, I found some confusion over subroutine names between different parts of the book and my browser display didn't look like the pictures in the book. OK, I'm whinging but, hey, either we're catering for newbies here, in which case we should be making sure that the programs run as advertised, or we're not, in which case why labour the explanations?

I agonised long and hard over how best to advise other members of ACCU on purchasing this book. Eventually, I decided on the following. If you're looking for an informative overview of selected computer science concepts, at an introductory level, then this book should interest you. However, if you're looking for a book to teach you either Java programming or hard-core computer science, there are probably more suitable texts.

WebLogic Server 6.1 Workbook 3ed by Greg Nyberg (0 596 00417 6), O'Reilly, 233pp @ £17-50 (1.43)
reviewed by Peter S Tillier

A number of my colleagues are WebLogic Server (WLS) developers and I wanted to review this book in order to 'keep in touch' with the technology.

This book is a companion to 'Enterprise JavaBeans, 3rd. Edition' (EJB3), also published by O'Reilly, which discusses how to create portable components that can move between application servers. It covers the implementation of EJBs under BEA's WebLogic Server 6.1, with updated examples for v7.0 being available from O'Reilly's web site. Other workbooks in the series cover IBM's WebSphere, JBoss and Sun's J2EE Reference Implementation.

The EJB3 book is not platform-specific and so particular installation environments need further documentation. The workbooks perform this function, each showing how its examples can be built to run on the different platforms.

The workbook is readable on its own, but for the maximum benefit needs to be read alongside the EJB3 book. I do not have a copy of the latter book, so I was not able to assess fully how well this book meets its objectives.

Despite the limitations of reading just this book it is possible to use it to download, install and try out the BEA examples under a 30-day free licence. This is a very useful idea as it gives the potential user of the WebLogic software enough time to try out the software in a real environment.

Some knowledge of XML, Java and JSP is really a pre-requisite for using the examples in this book, but if the reader doesn't have the EJB3 book enough information can be found using WWW searches and information from the BEA WWW site.

I would recommend this book to WLS users, provided that they are willing to buy the EJB3 book or to do some additional research on their own. Recommended with reservations.

Developing Mainframe Java Applications by Lou Marco (0 471 41528 6), Wiley, 430pp @ £35-95 (1.39)
reviewed by Peter S Tillier

The book starts out by discussing why a mainframe programmer might want to learn and use Java. In this respect the approach is good; many mainframe programmers that I have met tend to believe that COBOL, PL/I and/or FORTRAN are the only languages worthy of consideration. It then goes on to describe at a fairly high level how Java differs from COBOL and PL/I. This makes quite a good case for the move from traditional mainframe languages to Java.

The next chapter in the book discusses how to obtain a version of the Java SDK and install it on a PC. This is a little surprising since it tends to assume that all mainframe programmers wanting to use Java will have access to a PC - they are very likely to, but it's by no means guaranteed. Java on the mainframe is left until later in the book. The remainder of this and the following chapter discuss creating simple Java programs and using the various tools in the JDK.

The remainder of the 250 pages that constitute the first part of the book provides an introduction to the Java language, Classes and Objects, Encapsulation, Inheritance, Interfaces, Threads, Exceptions and an example of a course scheduling system. The introduction to OO concepts and Java are necessarily brief, but cover the ground pretty well.

There is also a short case study discussing the user interface approach taken by Java when compared with the traditional mainframe style. This study reappears briefly in the second part of the book when some Java code is compared with the COBOL equivalent.

The second part (about 100 pages) covers how Java and the OS/390 Architecture fit together, including a brief introduction to UNIX System Services, running Java in MVS batch programs, JRIO and interfacing Java with databases and CICS. This should be very useful for any beginning Java mainframe programmer.

The final part (50 pages) discusses Applets, The Java User Interface, I/O, J2EE and RMI - all briefly but usefully. This section leaves the reader wanting to know more about the topics covered.

In my opinion this is a useful book for the mainframe programmer moving to Java - it doesn't cover all that they will need to know, but does give a good basis to work from.

There were a few niggling errors such as misspellings, incorrect figure titles and so on, which seem to be present in most first editions of computing books these days. This is a pity because it spoils otherwise good books for the sake of proper proof reading.

On the whole I would recommend it to someone in its target audience, but there may be better-produced books available, hence my 'with reservations'. Recommended with reservations.

Design Patterns Java Workbook by Steven John Metsker (0 201 74397 3), Addison-Wesley, 475pp @ £31-99 (1.41)
reviewed by Silvia de Beer

This is a very good book indeed! Almost eight years after the appearance of the now classic book Design Patterns, another book explains the same twenty-three design patterns. During those years, the general understanding of design patterns has tremendously increased. However, it is still the question how much an average Java developer understands from design patterns and if he is capable of applying them when writing software and discussing with other developers. My guess is that most developers are still struggling with recognising, understanding and correctly applying design patterns. For these developers this book will be a very good aid on this subject.

The design patterns are regrouped in five groups, according to their intent; interfaces, responsibility, construction, operation and extension. Of course, for some patterns, their classification could be debated, but this is even encouraged. Remarks and questions encourage a better understanding of the intent and purpose of each pattern. The author discusses when simple features of the Java language could suffice to meet a requirement, or when it would be a better idea to re-factor code using a design pattern. The design patterns are not explained abstractly, but a concrete example is given based on a fireworks factory. The intentions of each design pattern are clearly stated.

A very useful feature of this book is that exercises appear throughout the text. This encourages a careful studying of the text. If you just casually read the text, you will not be able to answer all the questions. Some questions seem simple, but this may also be to not discourage newcomers. The questions break down into three categories; questions to formulate a response and reflect on what you have read, questions to complete or draw small diagrams and questions to complete or write small parts of Java code. If you are not a java developer, you could still benefit from this book, but you should at least have a minimal understanding of the Java language. I can recommend this book, it is the best book I have reviewed so far, taking into account the importance of this subject for developers.

Functional Programming

Haskell: The Craft of Functional Programming 2ed by Simon Thompson (0 201 34275 8), Addison-Wesley, 487pp @ £34-99 (1.37)
reviewed by Francis Glassborow

Many of us old hands learnt our programming by experience. The younger generations have, in theory, the good fortune to have been taught programming. Those that have gone to universities and colleges with high quality Computer Science Departments will have been taught a wide range of programming with several different programming languages. Unfortunately many will have attended courses where lecturers have little grasp of the way that languages differ and teach all programming languages as if they were just variations on a theme. This no more works for programming languages than it does for natural ones.

Functional programming languages are very popular with Computer Science Departments that have developed as an offshoot from Mathematics Departments. However they are often completely ignored by other Computer Science Departments. I am sure that this is, at least in part, brought about by a fundamental lack of understanding as to what these languages are. Some of you may be thinking that languages such as C and C++ must be functional languages because they rely heavily on calling functions. When you hear someone talk about procedural languages you jump to the conclusion that those must be languages like Pascal that have procedures. This is the typical kind of confusion that arises out of the way computing creates its own jargon.

A pure function 'returns a value' and has NO side effects. A pure procedure does NOT return a value and is entirely side effects. By convention a functional programming language is one where, in as far as is possible, the mechanism is on of using pure functions. Please note that there is no place for assignment in a functional style of programming because storing information is a side effect and so is procedural. Indeed it is one of the curiosities of C and C++ that assignments return values as well as store them. Unfortunately, C++ assignments do not return values but references and so are definitely on the procedural end of the scale unlike C where assignments return a value, that places C closer to the functional end. Perhaps it would be interesting to design a pure procedural language.

Now I am assuming that the reader of this review is a reasonably competent programmer in a language such as C, C++, C#, Java etc. but has not studied a functional language nor functional programming. If you are not in this category this review is not for you.

Haskell is a powerful functional language that has had a good deal of work done on it. Indeed even the basic language supports ideas of genericity and there are versions available that provide powerful extensions to support template-like coding. Those wanting to learn Haskell will find that there are good quality compilers available as freeware so the only up front cost is buying a suitable book and finding the time to study it with an open mind.

But why should a C++ programmer invest valuable time studying a functional language? The answer is that while C++ is not a functional language, and it isn't even a language with specific support for functional programming, it is a language in which functional ideas can thrive. The combination of function objects and templates provides a powerful programming environment, but one that few programmers are prepared for. I believe that exposure to functional programming will help the C++ programmer acquire insights that will help them use those C++ tools more effectively.

This book is an excellent text on functional programming in general and Haskell in particular. Studying (there is no point in just skimming through it quickly, you will miss the deep structure that will broaden your skills) this book will enrich your understanding of programming and add to your ability to program effectively.

One advantage of being a niche language is that the bookshelves are not weighed down with poorly written texts. This book does not have many competitors so rating it as one of the top ten books on Haskell isn't saying anything. However this book would rate well even if it had many competitors. Its author has a solid grasp of functional programming (FP) and this book explains both the basics and the specific mechanics of FP in Haskell. If you are a serious programmer looking to broaden your skills and craftsmanship try to find the time to study this book, you will be rewarded for doing so.

Perl

Beginning Perl for Bioinformatics by James Tisdall (0 596 00080 4), O'Reilly, 368pp @ £28-50 (1.40)
reviewed by Peter S Tillier

As I am a keen user of scripting languages, for applications such as language parsing and report formatting, I was interested in reviewing this book. It is relevant to my job and hobby for two reasons; expanding my knowledge and generalising my experience to the use of programming languages to assist in data processing for chemical and biological research. I have collaborated with programmers using awk and Perl to process such data and wanted to compare that experience with the use of Perl as described in this book.

The author describes his book as 'a practical introduction to Perl designed for biologists with little or no programming experience.' It is certainly that, but a considerable proportion of the early chapters would equally well serve as an introduction to bioinformatics for beginning Perl programmers, or even as an introduction to programming for people with no programming experience in Perl nor experience in bioinformatics.

So I see this book as alternative to many other 'Learn(ing) Perl ...' texts for those who are not employed as programmers, but are interested in analysing data for many scientific, or other, disciplines where there is a need. Sadly, some of the potential audience may be 'put off' by the title.

In my opinion the only people likely to find fault with the author's approach are those Perl veterans whose aim is to write dense and impenetrable code and who don't believe TMTOWTDI (there's more than one way to do it).

The book comprises 13 chapters and two appendices; the second appendix being a brief summary of a subset of Perl's features. I have downloaded and tried out the bulk of the examples both on a PC and my PDA using perl 5.6. The examples work pretty much as advertised, only needing minor changes to get them to run on the particular platforms that I was using.

The author provides a number of URLs and links to www resources for both Perl and bioinformatics. In particular some of the latter are very interesting even for a complete novice in the field (like me).

After writing the above review I took a look at the O'Reilly WWW site to see what others thought about the book. One reviewer thought much the same as I do - that the book is more than it claims to be in many respects. The site also contains a list of errata, which is quite long but not excessively so. It seems mainly to do with the bioinformatics part of the book rather than the programming part. This is, after all, a first edition, so a longish errata list is probably not too much of a problem, although proper proof reading ought to have found most of the flaws.

I would have bought this book if I had browsed it enough before buying, but many will not have the time (or possibly the luck) to do so. Even so I still recommend it because I think that it has a wide appeal as an introductory perl book. Recommended.

Web Based

Mastering XHTML by Ed Tittel et al. (0 7821 2820 3), Sybex, 1019pp @ £29-99 (1.33)
reviewed by Christopher Hill

This book is a disappointment on three very different aspects, the weight of the book, the HTML style presented and the surfeit of poor reference material.

A book is a medium that is to be handled while being used. At 1020 pages, I found that I could not read the book for long periods away from my desk, as the weight put a severe strain on wrists and arms. On the positive the binding did stay open at the required page when laid flat.

The authors do not quite make the jump to XHTML; they describe XML requirements and how this impacts on HTML (well formed, closing tags, etc.), but spend many pages describing deprecated features of XHTML. In fairness they do point out that these features are deprecated and refer to the CSS chapter. I would have preferred the authors spend more time on the CSS and less on the old HTML.

The authors introduce CSS, JavaScript and Multimedia (with a chapter on each) and there is a useful part (4 chapters) on the web site life cycle with useful tips for various styles of web site.

So far I have covered just the first half of the book. The second half is reference material, lifted from the W3C site and 'un-hyperlinked', so every tag has 3/4 of a page describing the class, id style and title attributes, over and over again. This material ought to have appeared in an accompanying floppy or CD (which would have removed the weight problem, as well as making for easier reference).

The HTML (sic) they present would appear to be accurate and the first half of the book could be useful, but check out other books and consider the weight before you buy.

Information Architecture for the World Wide Web by Louis Rosenfeld & Peter Morville (0 596 00035 9), O'Reilly, 461pp @ £28-50 (1.40)
reviewed by Christopher Hill

Very occasionally I read a book that covers current issues in a practical, lucid manner, such that my outlook is radically altered. I gladly confess that I had many 'Eureka' moments while reading (and re-reading) this book.

This book discusses the factors that contribute to a web site that 'works' - navigation, labelling organising, finding - a book about information, not data. Data are facts and figures; databases provide specific answers to specific questions. Within information systems it is a rare occasion when there is 'right' answer. Sometimes you are after a specific answer 'What is the population of Oxford?'; sometimes you have an area of interest 'Locomotives of the 19th century', no right or wrong here, some answers will be of real interest, others dross, but you are not quite sure which is which until you see it; sometimes you want to exhaustively explore a topic, leaving no stone unturned, so you cast your net wide and drag up everything that you can. So different ways of locating information are reviewed along with behind the scene techniques to assist the user.

Browsing - Users don't articulate their queries, but reach their goal through menus and links. Issues raised include Navigation systems, Sitemaps, 'Where am I?', Indexes, Linking Systems, Thesauri and Controlled Vocabularies.

Searching - Users enter their query and are automatically presented with a customised set of results that match their query. Issues raised include Search Interface, Query Language, Search Zones, Search Results, Nothing found, Thesauri and Controlled Vocabularies.

The book goes on to explore how to apply Information Architecture, putting it into practice and rounds off with a couple of comprehensive case studies.

If you want to understand how and why your favourite web sites are so easy to use, this is the book for you. Highly recommended.

PHP Cookbook by David Sklar and Adam Trachtenberg (1-56592-681-1), O'Reilly, 608pp @ £28-50 (1.40)
reviewed by Christopher Hill

The book is aimed at PHP programmers who have a basic understanding of the language, although some of the chapters cover material that I would have considered basic (i.e. chapters on Strings, Numbers and Arrays). Each 'recipe' or code snippet is presented as the solution to a problem, which is followed by a discussion which highlights the relevant points and concludes with a 'See Also' which often refers to the PHP online manual.

Sometimes the Problem - Solution - Discussion format does not help the subject matter, e.g. Problem 'I want to use a trigonometric function' - Solution use sin(), cos(), tan(). At other times the solution does not at first appear to match the problem, e.g. Problem - 'You want to replace the 3rd record in a file of 80 byte records, so you have to write starting at 161st byte'. Solution - fseek($fh, 26);.

Most of the code is presented as one or two liners, with lots of description. There are some code examples that run over a number of pages. These tend to demonstrate the features already described, thus providing useful reinforcement and a practical programme. Thankfully there are very few screen shots and other fillers.

This is not a book that you would read cover to cover and indeed the authors advise against this. There are twenty chapters, with topics as diverse as XML, form handling, Objects, Regular Expressions, encryption and PEAR. The large code examples are well structured and practical.

Many similar examples appear in many other PHP books - so if this were only your 2nd or 3rd PHP book, it would be recommended, otherwise view a copy and ask yourself if it answers questions you have.

Programming .NET Web Services by Alex Ferrara and Matthew MacDonald (0-596-00250-5), O'Reilly, 396pp @ £28-50 (1.40)
reviewed by Huw Lloyd

I think this book is more general than the title implies. Although '.NET' features in the title, the use of Visual Studio .NET is not implicitly assumed. Where .NET specifics are presented, such as in the use of C#, I do not think a programmer will find it difficult reinterpreting the small programs in terms of his own implementation.

Web service fundamentals are narrated first, these are not .NET specific, such as the deployment of files to establish and use Web services; they are well documented, for example, both HTML and SOAP are given exposure.

One caveat; the book does not hold all the answers. My reading concurs with the authors' claim that the remaining text is '... aimed at providing the reader with strategies for web service security, state management and transactions.' However, the strategies provided on security are a little thin especially regarding ticket systems; they inevitably leave more questions open than closed, although they are at least drawn to the programmer's attention. The small closing chapter on web service interoperability issues works to this end.

This version of the book refers to WSDL version 1.1 and represented the state of the technology as of mid 2002.

Overall, I think developers will find the book of practical merit. The content presents the information a programmer needs for a practical understanding of what web services are, what they can do for them and how they are implemented. Recommended.

Methodology & Design

Design Patterns in Communications Software by Linda Rising (ed) (0 521 79040 9), CUP, 547pp @ £37-50 (1.60)
reviewed by Mark Easterbrook

This book is a catalogue of design patterns in telecommunications software originating from TelePLoP and ChiliPLoP conferences. This is a valuable resource for any engineer working in the telecom or datacomms software. A basic understanding of design patterns and pattern languages is required to get the most out of the book although the material is presented in a readable style that allows the pattern concepts to be picked up as you go.

The telecoms and datacomms worlds leave the computer world standing when it comes to acronyms, abbreviations and other terminology, making them difficult to new comers. The book is careful to explain these clearly so that, for example, it is clear when IP means Internet Protocol (datacomms), Intelligent Peripheral (telecoms) or Intellectual Property.

As with any collection of patterns or pattern languages, you cannot just use it as a recipe book to produce software solutions - it takes time to learn the pattern languages and understand where they can be applied - and therefore it is not a book to be read cover to cover, but one to be dipped into from time to time, although it is necessary to spend some time initially becoming familiar with the book structure.

There are three sections to the material, the first two covering large collections and small collections respectively and deal with creating solutions from scratch. The final section is concerned with managing change and, given that most developers spend more time changing existing systems than creating new ones, is possibly the most useful section.

Architecture-Centric Software Project Management by Daniel Paulish (0 201 73409 5), Addison-Wesley, 284pp @ £26-99 (1.30)
reviewed by R N Lever

This is about project management using an architecture centric process, in fact, what it says on the tin. This book is most valuable to those project managers who are already familiar with other methodologies, the IT discipline in general and who are interested in other people's experience and approach. This is because the author describes an approach in this book that reflects his experience with software development within Siemens and is different to the standard methodologies.

The book is organised into key sections that provide an overview and then discuss the key elements; planning, organising, implementing and measuring. Finally, a number of case studies from the author's experience are included. The fundamental idea behind all of this is that by doing architecture work up front more realistic and achievable objectives can be defined within the project. However, anyone who has been involved in managing projects knows that the most important factor is people. It is here that the author really provides value because whether or not you agree with his approach what he says about people within projects is of almost universal application.

Having recently led a global project myself there were a number of comments made in the book that struck a chord. If you are very new to project management then you should start elsewhere first and learn the fundamentals from organisations such as the PMI (Project Management Institute). This is a book by a project manager for project managers. Whether or not you agree that the approach is useful there is enough additional content and nuggets of wisdom to provide value to the reader.

IT Measurement by International Function Point User Group (0 201 74158 X), Addison Wesley, 759pp @ £41-99 (1.31)
reviewed by R N Lever

The usual quotation is along the lines that you cannot manage what hasn't been measured. IT measurement is becoming more important as industry, academics and experts try to understand and improve the success rates for IT projects. This book, sponsored by the International Function Point Users Group, is a collection of articles that pulls together the current and considered opinion on the why, what and how of measurement.

The book is organised into thirteen parts with each part aimed at a particular topic area such as Measurement Program Approaches, Using Software Metrics for Effective Estimating, SEI and ISO-Based Metrics and Impact on IT/Business Measures. There are forty-three chapters spread over these parts with around seven hundred pages and each chapter is from a different industry expert. The content is well laid out, typically comprising an introduction, main content, summary and biography and at the back of the book is an extended bibliography.

Although the breadth of the book is wide the material became somewhat repetitive, a common problem for a collection of self contained articles. Also, whilst there was a lot of good information on the subject it lacked the depth of a practical implementation being initiated and delivered. This gave much of the material a superficial quality that hinted at much more than it actually delivered. Although function points are extensively referred to in relation to determining software product size there are a number of IT areas where it is inappropriate as a measurement tool. Unfortunately even with around seven hundred pages of information there is not much information on those areas that function points are not good at measuring.

For software product development managers who wish to review the current industry thinking on IT Measurement, primarily with function points, then this book collects much of that wisdom. However, it would be difficult to envisage how those same managers would have enough knowledge to be able to initiate and implement their own program, without requiring a significant level of consultancy. So, this is not a self-help book but simply a starting point for further investigation.

Request for Proposal by Bud Porter-Roth (0 201 77575 1), Addison-Wesley, 307pp @ £30-99 (1.29)
reviewed by Silvia de Beer

I chose this book because in the last few months I have been reading a few Requests for Proposals (RFPs), which made me curious about the qualities of a good RFP and the process of writing one. The book is both useful for people writing RFPs and for those who write proposals. It describes the whole process: from the intention to write an RFP to choosing the winning supplier. This book can be of a tremendous help if you are new to writing RFPs and also if you are involved in writing a bid or just have to read an RFP to start the work to fulfil a contract.

The structure of this book is good, with 7 chapters describing the different phases and parts of writing an RFP. Even the 60 pages of appendices provide useful templates and checklists; they have a real value, which is not the case in many books. The author repeats himself a bit in some of the chapters. An annoying fact for me was that the chapters do not contain numbered subsections. Only bold headers indicate the subsections.

A strong point of the book is that it makes you think about the process of writing the RFP and the actual text you write. Are the requirements unambiguous, measurable, meaningful and complete? Many important points are explained, many of which you might not think if you are new to writing RFPs.

By writing your RFP in a very structured way, you facilitate yourself in the evaluation of the bids you receive. The resulting proposals are more likely in a format that you require. The first step is RFP Planning and Preparation. When you write the Administrative Requirements section, you explain the process of bidding and the selection procedures your company is going to follow. An important section in an RFP is of course the Technical Requirements section. One must make sure that the person who writes a bid can easily make a distinction between the specifications and the requirements. In the Management Requirements Section you invite the suppliers to describe their experience in managing projects and in the pricing section you try to invite the supplier to quote his prices broken down, so you can easily compare the prices between the different suppliers.

Professional Development

The Career Programmer: Guerilla Tactics for an Imperfect World by Christopher Duncan (1 59059 008 2), APress, 211pp @ $29.95
reviewed by Pete Goodliffe

This is a book about software engineering in the Real World and how not to die of stress whilst doing it. It's well worth reading; I really enjoyed it and found myself agreeing with almost all that was said.

A great deal of the content is common sense and down to earth tactics for approaching software development in a manner that will work within the harsh confines of the corporate environment. Some of the advice is perhaps a little cynical, but all delivered in order to develop better software better and to have an easier life whilst doing it.

Some of the author's advice is self-defence, but there are also more offensive tactics and canny ways to work around bad project management. It's all delivered in a remarkably readable, conversational tone.

It's a shame, though, that the author relentlessly refers to 'Corporate America' as the problematic system programmers work within. The same issues are encountered in the UK and I'm sure any other country. Whilst this doesn't make the book any less readable, perhaps in a future edition this will be widened.

The book starts off describing why the typical software engineering job is not what a fresh faced software engineer expects, how they get bogged down in politics and poor project management. A frighteningly accurate and funny section describes the different sorts of programmers you will encounter in your career. The author then lays out, not so much a methodology, but a pragmatic battle plan to produce successful software projects.

If you are a programmer, team lead, project manager, or even software company director, I highly recommend you read this book.