Contents

Reports & Opinions

Reports

Dialogue

ACCU Spring Conference April 14-17 2004

Programme and Registration Form on centre pages

Features

Reviews

Copy Dates

C Vu 16.2: March 7th

C Vu 16.3: May 7th

Contact Information:

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

Advertising: Chris Lowe
ads@accu.org

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

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

Secretary: Alan Bellingham
01763 248259
secretary@accu.org

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

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

Membership fees and how to join:

Basic (C Vu only): £15

Full (C Vu and Overload): £25

Corporate: £80

Students: half normal rate

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

There are 6 issues of each journal produced every year.

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

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

Any questions - just email membership@accu.org

Reports & Opinions

Editorial

It Could Be You

Back in 2001, the then-editor of C Vu, Francis Glassborow, announced his intention to pass the editorship of this journal on to a new volunteer, and so at the start of 2002 I took the reins. In fairness I should say that Francis gave me considerable support in putting together my first few issues, and to this day continues to invest a lot of time in preparing various sections of C Vu. Since the time I took over, a lot has happened. I personally have moved from Bournemouth to Bristol, from Bristol to San Francisco, and from San Francisco to San Diego. The last two moves are not entirely unrelated to a spirited young lass by the name of Désirée, who was also closely involved when I became engaged and then married. Between those changes and others, it is now time for me to step aside and look for a new editor for C Vu; I can no longer give the job the time and energy it warrants.

Is this something you think you could do better? You might well be right. There's only one way to find out; get in touch, register an interest. Don't worry that there's magic to this job – there are many helpful people around to help, and the fine visual appearance of the journal (in fact, both journals) is down to the hard work of our production editor, Pippa. While I'm grateful to all those who have contributed material to C Vu over the last two years or so, the smooth running of the journals would not have been possible without Pippa's dedication.

Please do get in touch with our Publications Officer, Tom Hughes, if you would like to be involved with the future of C Vu. Details can be found in Tom's report later in this issue.

Prizes and Nominations

Early in 2003 we announced plans to award prizes (as well as fame) to the authors of the best articles in ACCU's journals during the year. Now that 2003 is over, it's time to determine which articles those are. Do you have opinions? (Who am I kidding – all programmers are overflowing with opinions...) Make them known to `cvu@accu.org`

How Good is Good Enough?

It's no secret that the declaration syntax of C and C++ is more complex than that of most other languages, and not only because it prefers symbols such as `*` and `[]` to more verbose Pascal-like notation such as `POINTER TO` or `ARRAY OF`.

Interestingly, this declaration syntax was something of an experiment, the idea being that declaring a name would look like its use. For example, derefencing the third pointer in an array looks like `*array[3]`, so declaring it looks like `element_type * array[3]`. A neat idea, but in an interview available from his website[1] Bjarne Stroustrup is quoted as saying that it is an "experiment that failed".

I introduce this here not to discuss the pros and cons of the syntax of these languages we know and love, but as an illustration of a question I find important: how much should a competent professional know of the syntax of the programming languages in which they claim competence? In particular, if these questions arise in a job interview, how much can the interviewee reasonably be expected or required to know? How much does it matter, for example, if they can describe the exact meanings of

```
int * const p;
int const * q;
```

and

```
const int * r;
```

The answer, of course, is that it depends. A different level of accuracy might be expected for a junior position than if the interview is intended to find a mentor for other programmers' language knowledge. It's also fair to expect more from a candidate who describes him or herself as an expert in the use of C or C++. To echo part of a conversation recently revisited on `accu-general`, any programmer bold enough to claim their knowledge of (say) C++ is rated at 10 out of 10, meaning that they know everything about the language, shows that they don't have enough awareness of what they *don't* know, and might well be dangerous through not exercising enough caution.

More important than being able to parse `bool (*(*p)[10])(int,int);` in your head is knowing that you shouldn't write such monstrosities when typedefs can simplify them so effectively, or knowing that should you ever need to figure out such a beast there is a widely-available program by the name of `cdecl`[2] that can translate many of them to English for you.

To take the example above:

```
$ ./cdecl
Type 'help' or '?' for help
cdecl> explain bool
(*(*p)[10])(int,int);
declare p as pointer to array
10 of pointer to function
(int, int) returning bool
cdecl> thank you
syntax error
cdecl> quit
```

Now to get back to my topic – what should an interviewee know to become a valuable part of a development team? They must know how to learn new systems and tools, how to work with existing code, and how to work with other team members. Next, they should have *enough* immediately applicable knowledge that they can increase the team's productivity without too much of a delay. And finally, more important than how much a developer knows is how aware they are of the limitations of their own knowledge, and how much they care to continue learning.

Cost of Keywords

After reaching a certain level of familiarity or expertise with C or C++, it is common for programmers to find that there are certain features they would like to add to the language. (Some of us would also like to remove features, but that's a tale for another time.) After deciding on the functionality that is to be added, its syntax must be considered. At this stage it is often natural to suggest using a new keyword; after all, that is likely to give the most explicit notation.

Getting a proposal requiring the addition of new keywords past the ISO standards committees, however, is not an easy task. The cost of adding new keywords weighs heavily on the minds of committee members. What costs are these? – surely any new feature has these costs in terms of compiler vendors implementing it, testing it, documenting it, and maintaining it, and users being educated in its use.

The additional cost of keywords is that they are likely to "break existing code", and that is a big no-no. As the rationale to the 1989 C standard said: existing implementations of the language are viewed as less important than code written to use the language. That makes sense: there are maybe dozens (at most, hundreds) of implementations of C and C++ combined, but there are literally millions of programs written using the languages, totalling many billions of lines of code. Somewhere in those billions of lines of code, almost any proposed keyword is likely to have been used as an identifier, and because C and C++ reserve their keywords in all contexts, those programs would become invalid when such a new keyword was added. The C committee was very conservative when C was revised in

1 The full interview text is available from `http://www.research.att.com/~bs/devXinterview.html`

2 `cdecl` has been extended to copy with both C and C++ declaration syntax, and is trivially ported to most platforms. I was torn over whether to give a reference such as `http://ftp.unicamp.br/pub/unix-c/languages/c/` to a website from where the `cdecl` source code can be downloaded or just to note that your favourite search engine can find it for you. The versions of `cdecl` I found had some flaws – missing `#include` of `<errno.h>`, use of the C99 reserved word "`restrict`" as a variable name, not knowing about the "`bool`" type or new types from C99 being among them. Maybe one of you would like to find fame updating it, or writing to `cvu@accu.org` to report a more up to date version already in the wild. In the meantime, you can download the source code I have used (and modified to add support for "`bool`") from `http://www.jamesd.demon.co.uk/c%20vu/cdecl-2.5-bool.zip`

1999, ten years after its original standardization by ANSI, but a persuasive case was made to introduce the new keyword `restrict` (to grant C compilers the license to assume function arguments are alias-free, an assumption Fortran compilers have long been able to make, and one of the reasons why Fortran has continued to out-perform C in much of scientific computing).

It happens that as I was preparing this editorial I came across an example of a program using `restrict` as an identifier: `cdecl-2.5`. So, have a little sympathy for the resistance to new keywords, and maybe even for the merciless overloading of existing keywords. I won't be surprised to see C++0x giving additional meanings to the keywords `explicit` and `typename`, and maybe even `default`. Any guesses what they might be?

ACCU Conference

By the time you read this, bookings for the ACCU Spring Conference 2004 should have opened. Details can be found on the centre four pages of this C Vu issue – they are also online at `https://www.accu.org/conference/` It truly is an exceptional conference. If anyone knows of any event offering comparable value on this side of the pond, I would love to know of it. To risk one cliché: if you can make it to one technical conference this year, this is the one. I hope to make it back to the UK for the next one.

James

From The Chair

Ewan Milne <chair@accu.org>

As you will read elsewhere in these pages, James Dennett has decided to stand down as editor of C Vu. James has been an effective and diligent editor, and has persevered with the job despite the considerable recent changes in his circumstances. I'd like thank him for all the hard work he has dedicated to the job, and to wish him all the best in his (relatively) new life in the States. Not, I hope, that we have heard the last from him: I hope that some of the extra free time he will now enjoy may be spent writing for the journals.

His departure does, of course, leave a vacancy. Please read Tom's Publications Officer's Report if you feel you have the time, skill and energy to take on this important role.

As reported in the last issue, we have now finalised a deal with the online journals database service EBSCO (`www.ebsco.com`), and so the current issue of Overload will be the first issue to be held in their database. As well as providing a modest revenue stream, we hope that making ACCU material more widely available in this way will raise our profile.

Preparations continue for the Conference, which by the time you read this will be Coming Soon. It's shaping up to be an unmissable event, I look forward to seeing you all there.

Membership Report

David Hodge <membership@accu.org>

Now we are in 2004 the renewals are all complete leaving us with 985 members, which is 140 down on the end of last year (end June 2003). I expect the membership to be similar to last year at the end as we tend to pick up quite a few members in the run up to the Conference in April. We have members in 40 countries, the largest groups being USA(98), Germany(30) and The Netherlands(17).

If you would like to make renewal a bit less painless you might like to consider setting up a standing order, just email me for a form or the details so you can set it up yourself.

Standards Report

Lois Goldthwaite <standards@accu.org>

This column in the last issue of C Vu reported that the standards group Ecma has chartered a new Technical Group to develop a C++ "binding" for the Common Language Infrastructure underpinning the .Net environment (there are already Ecma standards for CLI and C# which have been adopted by ISO under the fast-track rules for standards developed elsewhere). Although the principal objective is to develop another standard for ISO to fast-track, the original plans did not include a role for WG21, the international C++ standard committee in the ISO world. Some people, including myself, objected that this was a deliberate attempt to bypass the consensus-based process that makes ISO standards so highly respected.

This month I am delighted to report that a liaison relationship has been negotiated, so that interested WG21 experts can participate as observers in the Ecma process. They won't have a vote, but can receive the working papers from TG5 and join in reflector discussions and even face-to-face meetings without becoming members of Ecma. Of course the WG21 experts will have a chance to influence the vote of their national bodies, if and when the new standard is submitted for ISO approval.

This kind of a relationship is a historical first, so far as I know. Open standards are A Good Thing, but rivalry between standards organisations has undermined too many of them (remember the Unix wars?). Ideally, this collaboration will merge the nimbleness of the Ecma group – their aggressive schedule calls for producing the new standard in a little over a year – with the broader-based consensus process of the ISO committee. ISO has justifiably been criticised for its delays and red tape, but it is still esteemed as the "gold standard" of the standards world.

The CCLI efforts could be good for C++ itself, so long as the portability of standard C++ is not compromised by too many changes to adapt it to the .Net environment. In the computing world of the near future, the emphasis will be on distributed systems, quickly developed, whose various modules are written in a variety of languages, relying on maximal error-checking by compilers and run-time systems, and with the flexibility and resiliency to run for very long periods of time and even to evolve to meet new circumstances. But the old-fashioned virtues embodied by C++ – maximum speed, minimum resource consumption, and complete control over every low level detail – will continue to be important. If CCLI can harmoniously blend those philosophies, C++ will continue to be the multi-purpose, multi-paradigm language choice well into the future.

If you want to find out more about CCLI, Herb Sutter's overview presentation is at `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1557.pdf`

The draft CCLI specification can be downloaded from:

`http://download.microsoft.com/download/9/9/c/99c65bcd-ac66-482e-8dc1-0e14cd1670cd/C++%20CLI%20Candidate%20Base%20Draft.pdf`

For deeper involvement in the standards process, you can join the UK panels for C++ or C#/CLI. Please write to `standards@accu.org` for more details.

Publication Officer's Report

Tom Hughes <tom@accu.org>

After two years in the job our esteemed C Vu Editor, James Dennett, has unfortunately reached the conclusion that he can no longer afford the time required to edit the journal properly and has therefore decided to step aside.

As a result the committee is now looking for somebody to take over the role of editor. For anybody that might be interested in the role, the job involves soliciting and/or collecting material from contributors and doing whatever editing is necessary to make it ready for publication.

It doesn't include turning the finished articles in laid-up pages ready for the printers as Pippa Hennessy performs that job for us. All (and it's a fairly big all) the editor has to do is get the edited articles to Pippa on time every two months.

If anybody is interested in being considered for the editor's job, or would like more information about what the job entails, then they should contact me to discuss it.

Dialogue

Letter to the Editor

James,
I thought it was about time I wrote and introduced myself to ACCU members – it's probably long overdue given that I've been production editor for the journals for a couple of years now (just over two years for Overload and eighteen months for C Vu, to be exact).

Thank you, Francis!

Before I start, I must first say a big thank you to Francis Glassborow, who gave up his free time to teach me how to use Quark Express to produce professional-standard copy, and has been available since then to answer my many questions and on occasion to rescue me from a mess of my own making. Until I took over the job I wasn't aware of quite how difficult it is to produce a journal that not only has interesting and relevant content, but also looks like it contains material written by professionals for professionals. And Francis not only did that job, he was also C Vu Editor and contributed a lot of its content for many years!

So who am I, anyway?

OK, a bit about me. I'm a lead software developer working for a big IT company in Nottingham, where my responsibilities include running (mainly internet) projects, writing code, managing teams... all the usual stuff for someone who's been at the same company for over eight years. I actually prefer the management side of things now – I'm obviously not a true nerd! My technical skills include C++, HTML, JavaScript and ASP, all to a reasonable but by no means expert level. OK, now I feel like I'm writing my CV, but hey, it'll give some context. Before I started the production editor's job I didn't really have much experience in the skills required. I edited the Notts County Bridge Association bulletin for a couple of years in the early 90s – a Word document that I personally had to photocopy and staple together every other month – and (possibly due to not having a TV for the first 14 years of my life and hence reading an awful lot) I'm pretty good at copy editing and proof-reading. Apart from that, I'd never used desktop publishing software, and had never even thought about the hundreds of issues that need to be considered and problems that have to be sorted out when putting a journal together.

What does a Production Editor do?

Phew, where to start? I guess if I were to write a one-sentence summary (or is it a mission statement? I never thought I'd have to write one of those!) of the job of ACCU Production Editor it would go something like:

> Collect, proof-read and lay out all copy provided by the Journal Editors, and ensure that this and all other relevant material and information is provided to the printers and distributors according to a pre-defined schedule.

Oh, and I'm the one who defines the schedule too – what power! Of course, that sentence only just begins to cover what I have to do to get your words (you do contribute to the journals, don't you? if not, why not?) onto the printed page. The production process for a typical pair of journal issues goes something like this:

One month in advance of my copy deadline (which is itself two weeks in advance of the printer's deadline): Phone Parchment (printers) and Able Types (distributors) to book printing/distribution dates. Email all contributors (editors, advertising officer, membership secretary, cover picture contributor) to notify them of deadlines.

Any time after that: Receive copy from editors, proof-read and copy edit as necessary, apply standard formatting, pull into Quark Express and notify editors of page lengths.

Two weeks in advance of my copy deadline: Email contributors to remind them of the deadlines.

My copy deadline: Nag editors for copy where necessary. Notify them of current status of the journals – usually how many pages short we are of a sensible-length issue.

After that: Start to put final copy together – this can involve anything from straightforward text formatting to chasing authors for comments on how to lay out complex tables or diagrams and fiddling with large chunks of code to present it in a readable manner. Once a final draft is ready, email it to editors (if time) for comments and do a final proof-read.

Colour copy deadline: Generate front covers and send them to Parchment. Make sure all advertising copy has been sent. Inform Parchment of numbers required and ad placement.

B&W copy deadline: Generate PDF versions of the Quark Express documents and send them to Parchment, editors, ACCU webmaster, and anyone else who needs them. Generate Quark Express document containing book reviews and send it to the ACCU webmaster.

The following week: Deal with any queries from Parchment or Able Types, and gnaw my fingernails down to the elbow worrying about how serious my one big mistake will be (I've made one big mistake every issue so far – see how many you can spot!)

By the time the journals arrive on my desk I'm a nervous wreck – but so far it's been well worth it – I get a real sense of pride when I see the results of all my hard work.

Writing for the Journals

I have to admit, I haven't contributed much to the journals myself. I believe the sum total is one book review about ten years ago, and an article about developers' backgrounds about four years ago. But the pride I feel when I see the journals these days is nothing compared to the warm glow of actually seeing my name in print. I recently persuaded a couple of my colleagues to write up some work they'd been doing and submit it to C Vu – I'm sure they felt really good when their article was printed, and I'm also sure it's done their career prospects no end of good too, both in terms of personal development and in terms of being seen to be excellent at what they do.

I know the editors are always nagging readers to write something – anything – for the journals. I know you're probably fed up with hearing that for an association like the ACCU it's the members who make it what it is, and one way you can all contribute towards the success of the ACCU is to write about anything you feel may be of interest to others. But in my opinion the ACCU would die without the journals. And the journals will die without your contributions. You don't have to be able to craft superb prose, the editors, readers and myself are more than happy to help convert seemingly random sequences of mis-spelled words into articles anyone would be proud of. You don't even have to worry about looking stupid – editors and readers will spot any mistakes before articles even get as far as me. What we need from you are ideas, techniques, tools, reviews, commentaries... literally anything you feel at least some of your fellow ACCU members might be interested in.

I've been meaning for a while now to write something like this letter for C Vu. What's triggered me to actually sit down and put fingers to keyboard is that C Vu 16.1 is, at the time of writing (5 days away from the final deadline), desperately short of material. Overload 59 was also short – thanks to Kevlin Henney allowing us to reprint a series of articles he originally wrote for the C/C++ Journal Experts Forum it's not quite so much of a problem, but we're getting close to the end of that series...

Required Reading

I had many excellent Christmas presents last year. One of the best was "Eats, Shoots and Leaves: The Zero Tolerance Approach to Punctuation" by Lynne Truss (Profile Books, ISBN 1861976127). If you want to write articles, and want to improve your knowledge of how to use punctuation (or are just a pedant who hates the Greengrocer's Apostrophe), I can't recommend this book highly enough. Hey, perhaps I should write a review of it for C Vu!

Pippa Hennessy
<dipsy_x@oldbat.co.uk>

Student Code Critique Competition 26

Set and collated by Francis Glassborow
Prizes provided by Blackwells Bookshops & Addison-Wesley

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

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

Before We Start

I have become accustomed to an ever growing stack of books waiting for me to get round to reading them (for the first time forty-five years I have failed to read a serial in Analog as soon as the last part was published). What came as a surprise was to realise that writing that I am committed to is going the same way – the stack is growing faster than I can reduce it. Some things have to be done by me but other things could be done by others if they had the will to do so.

This column is one of the latter. Its contents are largely written by others. Putting it together takes roughly a day's work. Is there anyone reading this who would be willing to take it over? I can continue to manage the supply of prizes but freeing up another day would be much appreciated.

Student Code Critique 24 revisited

For some reason some of my email has been disappearing before it reaches me (and as my ISP does not yet apply any filters, though they will soon, it isn't caused by some form of false positive for spam). I keep complete logs of all mail that reaches my mailbox and can identify several places where people have clearly sent me things that have never arrived (what worries me is the cases I do not know of, but note that I always acknowledge email with content).

It seems that two entries for SCC 24 were consumed by this email eating demon. So here they are but without the restatement of the problem.

From Matthew Towler <towler@ccdc.cam.ac.uk>

This program is simply outputting text so first of all I will suggest coding purely in standard C++ and removing the MSDOS specifics. These are `clrscr()` (more on this in a moment) and `getch()` which I think is being used to wait for a character to be pressed between groups of output. Removing these means we can also remove `<conio.h>`.

The first line of `main()` is `void clrscr();` I guess this is intended to call an MSDOS function to clear the screen, but in fact it declares a function named `clrscr` taking no parameters and returning nothing – but does not call this function.

The simplest way to get the output into a text file would be to redirect the output. Assuming the code compiles to an executable named `Sum.exe`, then this would be achieved from the command line (from the directory containing `Sum.exe`) as follows:

```
Sum > output.txt
```

Then the following should open the complete output in notepad

```
notepad output.txt
```

The output might be enhanced by adding some separators e.g.

```
std::cout << x << '\t' << m << '\t';
```

would output the values of `x` and `m` with a tab character in between.

The headers `<iostream.h>` and `<iomanip.h>` are not standard, they are old pre-standard headers and should be replaced by `<iostream>` and `<iomanip>`. This will mean that `cout` and `setw` will need to be additionally qualified with `std::`. `<math.h>` is standard but deprecated, you should really use `<cmath>` which would place the `pow()` function in the `std` namespace. However in practice compiler support for `<cmath>` is patchy, and all will provide `<math.h>` for C compatibility so I usually stick with `<math.h>`.

My next point is that of named constants. The values `1`, `11`, `25000`, `35000`, `5000` and `2` appear several times each in the code. I am not sure what the code is calculating but it is likely the meaning would be much clearer if the constants were named to indicate their meaning or purpose. This would emphasise the connections between the individual loops and the data they are handling and just as importantly it would reduce the likelihood of a single typo breaking the code.

Some more informative naming of variables would also help. `i`, `r`, `j`, `m` and `x` leave the reader in the dark about what sort of calculation is being performed, and also give a higher probability of errors due to typos.

`Sum` is declared `static`, which for a program consisting of a single non-recursive function does not significantly alter the meaning. The only advantage of this is that it will not allocate 25000 `int`s on the stack; MSDOS often has a fairly small stack and this could be important. `Sum` is not declared with a type, so it is using the deprecated `implicit int` C feature. Most modern compilers will now warn about this and a few will refuse to compile the code.

An alternative to this two dimensional array would be a `std:vector < std::vector< int> >`. This would allow the access syntax to be changed from `Sum[i][j]` to `Sum.at(i).at(j)` which will check all accesses are within limits at run time. This could be a helpful debugging aid in a simple program such as this where efficiency is not paramount. The equivalent declaration of `Sum` is as follows.

```
std::vector< std::vector<int> >
   Sum(5000, std::vector<int>(5000));
```

Use of `std::vector` would also mean that all the integer values in `Sum` would be automatically initialised to zero, as opposed to the static data which will not do this [*actually, you are mistaken, static data is zero-initialised by default – Francis*].

It usually aids readability if variables are declared to exist for the shortest time; and initialised as they are declared. For instance `r` is only used within the first `for`-loop and initialised at the end of the loop; it could be declared and initialised at the top of this loop. The advantage of this is that when reading code there are less lines to search for the variable. `r` is currently being used uninitialised; but I did have to think for a long time to work out (and changed my mind several times) whether or not the loop bounds in the nested loops conspired to initialise it before first use. It is exactly this sort of confusion that this guideline is intended to avoid.

Similarly the loop variables could be declared within the loops e.g.

```
for(int i =1; etc.
```

The code inside the first pair of nested `for` loops would be simpler if the `if(i == 1)` block were moved above the inner loops and the inner loop started at 2. This saves an `if()` per loop, and far more importantly clarifies the intent that this code is setting up the calculation for the subsequent iteration.

Finally, I am not sure about the mixing of floating point and integers in the loop conditions; namely

```
i <= pow(2, j)
```

in the first loops and similarly for `x` and `m` in the second loops.

In `math.h`, `pow()` is declared as

```
double pow(double, double);
```

Standard C++ (but not C) also provides an overload (according to Stroustrup 3rd Ed).

```
double pow(double, int).
```

Which overload is called is irrelevant as the problem lies with the `double` return value. For example if `i = 4` and `j = 2` the expression resolves to `4 <= 4.0`, which may not compare as you expect. To perform the comparison the `int` on the left will first be promoted to a `double`, which will result in an approximation of the integer value such as 3.999999. This small error can in some cases change the result of the comparison [*True in theory but all compilers I know of correctly and exactly represent the floating point equivalents of whole numbers provided as constants – it is the computed value which is suspect not the conversion from* `int` *to* `double`. *Francis*].

The solution to this is to write a function to call `pow()` and perform the comparison with a tolerance to take account of any approximation errors.

From ??

[*Sorry, but I really do not have time to go through my email archives looking for names of authors. If you send me a file I save it for future processing. If you leave your name out of the file, your work will be published anonymously. Francis*]

My comments to Student Code Critique 24 are split into two parts. The first part describes the changes needed to get the required behaviour of the program. the second shows a number of ways the code could generally be improved.

Solving the problem

Rewriting the program to make it write data to a file requires very little. A few changes to existing code and a couple of new lines does the job:

Taken from the top the changes are:
Add a new `#include` statement:

```
#include <fstream> //file handling
```

The line:

```
void main(){
```

should be changed into:

```
int main( int argc, char* argv[] ){
```

The standard mandates a return type of `int`, the 2 parameters gives us access the command line parameters, which we will use to name the file to write the data to. Insert the following to achieve this. [*If you choose that mechanism you should first check that there is a second command line argument and check that the file is successfully opened.*]

```
// open an output file named by the first
// argument to the program.
std::ofstream output(argv[1]);
```

Change the line:

```
cout << x << m << setw(10)
     << Sum[x][m] << '\n';
```

to

```
output << x << m << setw(10)
       << Sum[x][m] << '\n';
```

and the data will be written to the file named as the first argument to the program on the command line.
This makes the program write data to a named file instead of `stdout`. [*Note I edited the submitter's* `std::out` *– there is no such thing.*]

Improving the code

The code contains a lot of unnecessary noise. Removing this noise can lead to a program that is significantly easier to read and understand. Removing the noise also increases the programmer's trust in the output of the program.

I have grouped the noise into categories, trying to isolate the core of each problem. My hope is that it will help recognizing the type of problems in the future.

#includes:

`#include`s should follow the C++ standard and use the include files without the `.h` extension. The C header for math should be replaced by `cmath`, which includes the math definitions in the `std` namespace.

Some of the `#include` statements in the program are not necessary, and they should be removed from the code. Include statements which are not used add false dependencies.

For large programs with a lot of header files, the false dependencies can forces a recompilation of an otherwise unchanged compilation unit. On a heavily loaded machine the reading and parsing of the include files can be a large part of the total compilation time.

Forward declarations:

Forward declarations of functions and variables should be deferred to the latest possible time. This ensures that the declaration is made as close to point where it is used as possible. A consequence of this is the removal of declarations that are not used.

Following this advice, we remove the lines:

```
void clrscr();
int i, r, j, m, x;
```

Arrays:

One should try to only use arrays of the needed size. Allocating more memory than we need might be masking that we do not really know how memory is used. It could be hiding an error in indexing, which could be hard to spot, even when using memory bounds checkers.

A simple scan of the code that assigns to the array `Sum` shows that the variable `j`, which is used as the second index into the array, only indexes the range [1:11]. If we change the declaration of the array `Sum` to reflect this usage, we reduce the amount of memory required by a factor of 454.

The first index `i` runs in the interval [1:2048], which leads to a reduction of the needed memory by a factor of approximately 2.5, all in all we are able to reduce the memory need of the program by a factor of over 1100.

The indexing into the array is done from 1. The C and C++ arrays are normally indexed from 0, but adapting an existing algorithm that used indexing from 1 can lead to problems.

This is especially important if the code might be used by others, or if the code will be used again and again over a period of years. It will be a lot easier to see the connection between the algorithm and the code if we keep indexing from 1.

The declaration of the array should look like this:

```
long Sum[2049][12]
```

The keyword `static` that was previously used is not necessary [*I think the writer has missed the significance of using static to both ensure default initialisation and to use 'static' memory provided at load time rather than, possibly precious, stack memory.*] The size of the array has taken into account that the indexing starts at 1.

for loops:

The `for`-loops should declare the type of the counting variable it uses inside the `for`-statement. This makes sure that the counting variable is only visible inside the `for`-loop [*As this was for an MSDos platform, that might not be true*].

The inner `for`-loop contains the construction:

```
for(int i = 1; i < pow(2,j); i++) {
// code //
  i = i + 1;
}
```

Because the code block spans several lines, it is not immediately obvious that the `for`-loop step size is two and not one. This should be written as:

```
for( int i = 1; i < pow( 2, j ); i =+ 2 ) {
// code //
}
```

The innermost loop has the following construction:

```
for(i ......) {
  if(i == 1) {
    // code //
  }
  if(i > 1) {
// Code using variable r
    r = r + 1
// code not using variable r
  }
  r = 2;
}
```

From the above it should be obvious that every time `r` is used, it has the value 2, and we might as well change the single usage of `r` to using a properly named variable, like

```
const long magic_constant = 2;
```

Making it obvious that the code depends on a magic constant.

To ease the reading of the `for`-loops writing the data, these loops should use the same variables as was used to generate the data:

```
for(int j = 1; j <= 11; ++j) {
  for(int i = 1; i <= pow(2.0, j ); i++) {
    output << i << " " << j
           << setw(10) << Sum[i][j]
           << '\n';
    output << i << " " << j
           << setw(10) << Sum[i][j]
           << '\n';
  }
}
```

The innermost loop for writing the data two times could be changed to two lines, writing the same output. This makes it more obvious that the data is written twice, at the cost of making all changes to the writing code in both places, but the eye is quite good at catching differences in lines that should look the same.

The post increment operator used in the `for` loops, should be changed to the pre increment operator, if not the code will generate a temporary to hold the value before incrementing, that will never be used.

Readability:

The use of space characters in the code, can significantly increase readability, especially if used in a consistent way, like always enclose operators like `+`, `-`, `*`, `/`, `=`, `==`, `<=` and so on with an equal amount of space before and after. [*However placing spaces after any form of open bracket and before any form of close bracket often reduces readability. In addition there are good readability arguments for not placing spaces either side of a strongly binding operator such as* `*` *or* `/`.]

The rule could read something like: Binary operators have an equal amount of one or more space characters on both sides.
The difference is clearly seen comparing two versions of the first for statement:

```
for(j=2;j<=11;j++)
```

and

```
for ( j = 2; j <= 11; j++ )
```

[*Yet is not:*

```
for(j = 2; j <= 11; j++)
```

slightly more readable? Too much space can be as bad as too little.]
All in all the code ends up looking like this:

[*snipped*]

This code generates the same output as the sample output, except that a space is inserted between the writing of the two array indexes.

Student Code Critique 25 entries

The problems

Program 1

I am little confused about return values of `sizeof` operator.
Here is a simple C program which is putting me in doubt:

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

Output is :

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

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

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

Program 2

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

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

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

From Annamalai Gurusami

<annamalai.gurusami@email.masconit.com>

Provide Prototypes

It is a good practice to include all the relevant header files (meaning, provide the proper prototypes), even though it is not mandatory as far as the C programming language is concerned. (It is compulsory in C++). For using `malloc`, include the `stdlib.h` header file, so that the necessary prototypes are provided to the compiler.

In the absence of an appropriate prototype, the compiler would assume that the return type is an integer (while `malloc` function actually returns a void pointer). This is fine in a 32-bit system, but will be a problem on a 64-bit system. The reason is that in a 32-bit system, an integer and a pointer occupy the same amount of memory space (four bytes), whereas in a 64-bit system, an integer occupies 4 bytes, but the pointer occupies 8 bytes. So provide prototypes. It might save you a lot of trouble later.

[*The above paragraph contains a number of misconceptions. There is no requirement on any system that any integer type have the same amount of storage as any pointer type. Note that not only does both C & C++ allow pointers to different fundamental types to be different in both size and layout, but some implementations actually use this license. The requirements are that* `void*` *and* `char*` *be identical in size and layout;* `void*` *must be able to store the value of any data pointer without loss of information and that all pointers to* `struct` *(and, in the case of C++,* `class`*) only need a declaration of the type name. Different types of pointer can, and sometimes are, different sizes. Pointers can, and sometimes are laid out differently to the layout of an* `int`.

Lastly, as of the 1999 release of C, implicit `int` *and implicit function declarations are no longer supported.* ***Francis***]

Array Name Is Not A Pointer

An array name is not exactly a pointer [1], even though in many circumstances it degenerates into one. An array is a single entity, in the sense that its name and its memory location are inseparable. When we declare a variable like `int a[10];` we have an object that can hold 10 integers and it is named `a`. There is no other memory associated with that object. When we just use an array name where a pointer is expected (type being appropriate), it is interpreted as a pointer to the first element of the array. In our case, if we use `a`, where an `int*` is expected, then it is evaluated as `&a[0]`. This dirty work is done by the compiler. Also note that this association cannot be changed. For example, we cannot do something like

```
int a[10];
a = malloc (10);
// *ERROR* cannot reassign another object to a
// a is not a pointer variable.
```

This is why `sizeof(a)` gives the size of the array object named `a`. The size of `a` in our case is `10*sizeof(int)` which is often equal to 40.

The confusion mainly comes because of the applicability of the subscript operator to an array and a pointer pointing to a `malloc`'ed memory location. For example,

```
int a[10];
int *p = malloc(10*sizeof(int));
// initialize a[0]...a[9] and p[0]...p[9]
for(int i=0; i < 10; ++i) {
  printf("p[%d]=%d\n", i, p[i]);
  printf("a[%d]=%d\n", i, a[i]);
}
```

The similarity between these two usages is the main reason for the confusion that an array name is a pointer. But one should remember that the "array" `p` is not a single entity. There are two entities involved – one is the pointer variable `p`, and another is the allocated memory that can hold 10 integers. Also the association between the allocated memory and the pointer variable is only temporary, in the sense that the pointer variable can be made to point to another memory location. For example:

```
int a[10];
int *p = malloc(10*sizeof(int));
// initialize appropriately
p = a;  // correct: evaluated as &a[0]
a = p;  // error:  a cannot be re-assigned.
```

I leave it as an exercise to the reader to convince himself/herself that an array name is not a const pointer.

The behaviour of `sizeof` operator

The `sizeof` operator is evaluated at compile time. And it gives the size of the object that is given as an operand. So:

```
int a[10];
sizeof(a);
// evaluates to the size of 10 integers
char *p = malloc(n);
sizeof(p);
// evaluates to the size of the pointer p
// which is independent of n.
```

Note that the type of the result of `sizeof` operator is `size_t` (which is `unsigned int` or `unsigned long` [*Or some other unsigned integer type*]). So check for the appropriate format specifiers in your system and use it in the `printf` statements.

The argument to malloc

In the provided program, the argument to `malloc` is 10 (which is in `unsigned char`). The size of an integer [*on this system*] is 4 bytes. So the value provided to `malloc` is not in multiples of the size of an `int`. This most probably will result in hard-to-find bugs (because of memory alignment problems. [*Not likely, malloc is required to return suitably aligned memory for any type*]) Also the behaviour might depend on the implementation of `malloc`. Whatever the after effects, the provided code doesn't seem to reflect what the user wants. If the user wants to store 10 integers, then (s)he has to calculate the required size manually, by doing `10*sizeof(int)`, and passing that to `malloc`.

If it was C++, things are more clear. For an array of 10 integers, we can just write

```
int *p = new int[10];
```

Casting the void* returned by `malloc`

Since the function `malloc` returns a pointer of type void (a `void*`), there is no need to cast it explicitly. A `void*` can be assigned to other pointers without any explicit cast [*In C but not in C++*]. In C++, it is recommended to use the `new` for allocating memory, which returns a pointer to the type asked for. operator [*Not really, the memory allocation tool in C++ is* `operator new`*, the creation of dynamic objects is done with a* `new` *expression*] Since no generic pointer types are involved, there is no question of casting.

Problem 2

Compilation Errors

The use of `hypotenuse` instead of `hypo` in the last `printf` statement will result in compilation error. Also if it was C++, the compilation would fail because the return type of `main()` is omitted. [*Also true in C99*]

Comment on Comments

Comments are normally used to explain what a particular piece of code does (if that cannot be easily understood, just by reading it) and why. Comments like `// Start of Main Program` are really superfluous, and do not add value. Also wrong comments can be very harmful. For example, the comment `// Define Libraries` is not correct. The header files just provide declarations. Any declared functions are defined elsewhere.

Flushing the standard output

When a prompt needs to be displayed to the user, it is always prudent to flush the standard output (because a prompt normally doesn't terminate with a new line). If this is not done, then it is not guaranteed to be displayed on the terminal, because most of the terminal drivers are line buffered. [*But most, if not all, flush* `stdout` *before collecting input from* `stdin`.]

Input Validation And Error Checking

There is no validation done on the provided input. The following checks can be performed on the input. Check if they are legal numbers. Check if they are greater than zero.

Check if there has been any error while calling the various functions (at least for `scanf`, `pow` and `sqrt`.) For example, while calling the `pow` function, it is possible that an overflow occurred. This can be determined by checking the `errno` variable. Such validations would help to determine the precise cause of a failure.

Also note that `scanf` doesn't provide much error checking. For example, if a value overflows, `scanf` doesn't report it. For this reason, a combination of `fgets` and `strtod` is preferred. The `strtod` function returns the error status through the `errno` variable.

The format specifier in scanf

For a `double`, `scanf` requires that `%lf` be specified as the format. [2]

Linker Error

Including a header file doesn't "define" the functions of a library. They just declare them. The function definitions of a library are probably available elsewhere in the system as shared/static libraries. This information should be provided to the compiler (or more precisely, to the linker). Normally the option is '-l' (l as in Link). For more details on this, look at the compiler/linker documentation and man pages.

References:

[1] `http://www.eskimo.com/~scs/C-faq/q6.2.html`
[2] `http://www.eskimo.com/~scs/C-faq/q12.13.html`

From ?? [see my comment earlier]

I'll first answer the main concern of the student, and then address some other issues. Certainly, there's a misunderstanding about arrays and pointers. The first definition `int a[10];` creates an array of ten `int`s, that is, it reserves space somewhere for ten consecutive `int`s. That 'somewhere' can be identified as `a`. On the other hand, `int *p = (int *) malloc(10);` creates a pointer to an `int`, named `p`. Where does it point? To the memory location [*for a block of ten* `unsigned char`] returned by `malloc()`, or to `NULL`, in case the request couldn't have been granted. So now let's tackle the size issue: `sizeof()` is an operator which returns the number of bytes of an expression or a type passed as argument. So if `sizeof(int)` is 4 in your environment (which happens to be a common size on most 32-bit architectures) then `sizeof(a)` will be 40 (10 integers each occupying 4 bytes) In the second case, you get the size of a pointer to `int`, which in your system turns out to be 4 bytes.

Now with this information in hand, we have a better understanding to answer to your questions:

Q: "My understanding says even array name is a pointer. If so why it does not show `sizeof(a)` as 4?"

A: Because an array is not a pointer! In many cases, its name is converted to a pointer to the first element (this is generally called decay), but the object itself remains being an array! The only exceptions when this decay doesn't take place are when being an operand of the sizeof operator (which is our case), the & unary operator (address operator).

Q: "Or if `sizeof` shows the total allocated memory then why `sizeof(p)` does not show 10?"

A: Because `sizeof` just returns the size in bytes of the operand passed, in this case an identifier for a pointer to `int`.

Now let's see briefly a couple of errors in your code. Let's see `malloc()` usage. First of all, the cast is unnecessary, since a `void*` is automatically converted to any other pointer type. That alone isn't a big issue by itself, but your code shows a good reason not to do it: you forgot `#include <stdlib.h>` which is where `malloc()` prototype is declared, thus letting the compiler assume the function returns an `int`. With this assumption, assigning the `int` returned by `malloc()` to the pointer to `int`, would cause a diagnostic unless you force the conversion, which is done by a cast, which is exactly what you're doing. See how this innocent-looking cast turns out to hide a potential bug in your program. So the advice here is not to cast but #include the appropriate header.

Another error is the omission of a `return`-statement, given that `main()` must return an `int`, as your definition clearly states.

The last error has to do with the data type used to represent the size of an object. `sizeof` yields a `size_t` value, which is a `typedef` to a basic type, such as `unsigned int` or `unsigned long int`. So what specifier (`%u`, `%lu`) should we use? Well, it depends on the system you're working on, depending on `sizeof` definition (check `stddef.h`) On the other hand, in C99 you can use `%z` for `size_t`, thus avoiding the above mess. Lastly, I suppose your intention is allocating space for 10 integers, so if that were the case, you should do:

```
int *p = malloc(10 * sizeof(int));
```

or

```
int *p = malloc(10 * sizeof *p);
```

that is, reserve space for 10 objects of the type pointed by `p`, which in this case is an `int`. Note that even though both statements are valid, the second one is preferred, because the data type is not fixed, but expressed by the content of the pointer. So if you later happen to change of pointer type, you won't run into any problem.

So our program would look like this:

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

int main() {
  int a[10];
  int *p = malloc(10 * sizeof *p);
  printf("Size of a = %lu\n", sizeof a);
  printf("Size of p = %lu\n", sizeof p);
  return 0;
}
```

Note that `sizeof`'s operand doesn't need to be put inside parentheses, when it isn't a data type.

Now I'll show you the same code in C++, and briefly discuss some useful facilities the language provides us.

```
#include <iostream>

using std::cout;

int main() {
  int a[10];
  int *p = new int[10];
  cout << "Size of a = " << sizeof a << '\n';
  cout << "Size of p = " << sizeof p << '\n';
}
```

The two most important differences relate to memory allocation and printing output to `stdout`. First note that using `new`, you don't have to think in terms of raw bytes as you do with `malloc()`. You just ask memory for `n` objects of type `T`. Also note that you don't need to use bizarre specifiers such as `%d`, `%u`, etc., you just specify what you want to output and there you go. One minor detail here is that the compiler inserts automatically a '`return 0`' if you don't provide it, but note that this is done only in `main()`.

[*However note that this program, like the C one, forgets to release the memory obtained dynamically*.]

Program 2

This problem has to do with misuse of conversion specifiers. Unfortunately, the same specifier (`%f`) means different things in these two generally interrelated functions. In the case of `printf()`, '`%f`' specifies a `double` argument, and applying the length modifier '`l`' doesn't affect its meaning at all. So in our case, using '`%f`' or '`%lf`' hold the same meaning [*] So what if you want to print a single precision floating point number? (a `float`, to be precise). You have no other choice other than using `%f`! The key issue here is automatic conversion: when you pass a `float` to `printf()` (and to any variadic function, the compiler automatically promotes it to a `double`. Thus `printf()` always receives `double`s! Note there isn't any problem with this, as we're not losing precision. But in the case of `scanf()`, '`%f`' and '`%lf`' are two different things. In the first case you ask for a `float`, in the second for a `double`. Why the distinction here? Because `scanf()` arguments are pointers, which are not promoted.

This is also a good occasion to point out a better way to receive input from a user. Actually, `scanf()` is targeted to read formatted input (for example, from a configuration file, table, etc) rather than from an unforeseeable input source as a human being. The problem with `scanf()` relates to its poor support of error detection, among other things. Instead you should use something like the `fgets()` and `strtod()` in combination:

```
fgets(buffer, sizeof buffer, stdin);
strtod(buffer, NULL);
```

You should also add some error checking to make sure you get what you're expecting. `fgets()` makes it easy; returning a `NULL` pointer in case of error. `strtod()` is a little more complicated, because it returns you the value parsed, or `0` if there was any problem. But as you might have already guessed, `0` is also a valid value! So to set apart an error from a legitimate input, `strtod()` makes use of the infamous `errno`. Fortunately for us, we won't deal with that mess. In our problem domain, we're dealing with sides of a triangle, which means that positive values are the only valid values we're looking for! [*What about degenerate triangles?*]

This simplifies a lot our function, at the expense of getting more detailed error messages:

```
double read_side(void) {
  char buf[32];
  if(fgets(buf, sizeof buf, stdin) != NULL)
    return strtod(buf, NULL);
  else
    return 0;
}
```

Now let's see some other problems, in a 'top to bottom' approach:

```
// Define Libraries
#include <stdio.h>
#include <math.h>
```

This is wrong. You're not defining any libraries at all. You're simply including declarations to be used by your program.

```
main() {
```

`main()` returns an `int` by definition, so you should state `int main()` instead, and place the corresponding `return 0` at the end of the function.

```
printf("Enter base:");
scanf("%f", &base);
```

You need to add `fflush(stdout)` to ensure text is displayed before the `scanf()`. [*I cannot remember ever finding that necessary*.]

```
printf("hypotenuse is %f", &hypotenuse);
```

You're passing the address of a `float` variable to `printf()`, instead of the value expected. Besides, the identifier you're passing doesn't even exist. In case you meant `hypo`, you should pass the object itself, not its address, to comply the requirement imposed by the `%f` specifier. You should also add `fflush(stdout)`, or suffix a `\n` to ensure the string is flushed. [*They do entirely different things, the first forcibly flushes* `stdout` *the second appends a new-line to the output buffer.*]

Lastly, not an error but a better practice. In this case, it's better to manually square the numbers you received, that is, use something like `sqrt(base * base, height * height);` which will surely give you better performance.

This is how a possible implementation might look like:

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

double read_side(void) {
  char buf[32];
  if(fgets(buf, sizeof buf, stdin) != NULL)
    return strtod(buf, NULL);
  else return 0;
}
int main(void) {
  double hypo, base, height;
/* Enter base and height */
  printf("Enter base: ");
  fflush(stdout);
  base = read_side();
  printf("Enter height: ");
  fflush(stdout);
  height = read_side();
  if(base > 0 && height > 0) {
    hypo = sqrt(base*base + height*height);
    printf("Hypotenuse is %f\n", hypo);
    return EXIT_SUCCESS;
  }
  else {
    fprintf(stderr,
            "Both sides must be positive\n");
    return EXIT_FAILURE;
  }
}
```

[*] This only applies to C99. In C89, the behaviour is undefined, so we'd better use `%f` for maximum compatibility.

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

Problem 1

The program needs to include `stdlib.h` for the declaration of `malloc()`. Without this declaration it is assumed to return an `int` rather than a pointer to `void`.

If the student is trying to dynamically allocate the same amount of storage as a ten element array, then he/she should replace the argument 10 in the call to `malloc()` with `10 * sizeof(int)`.

The cast to `(int *)` of the return type from the call to `malloc` is not required since the coercion of `(void *)` to `anytype *` is automatic. It may be harmful to do so if `malloc()` is not declared to return `void*`, as in this case because of the missing header file. The explicit case was necessary in pre-ANSI C and is still necessary in C++. In the case here, the cast may be erroneous because only 10 bytes have been allocated.

Minor stylistic point: No check is made on the return code of `malloc()`. `malloc()` returns `NULL` if the memory cannot be allocated. In production code this should be checked.

The function `sizeof()` [*actually it is an operator not a function*] produces an unsigned integer object of type `size_t` (declared in `stddef.h` or `cstddef` for C++). This header should be included for completeness.

The C Standard (ISO/IEC 9899:1999(E)) states in 6.5.3.4/3

"When applied to an operand that has type `char`, `unsigned char`, or `signed char`, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding."

The C Standard (ISO/IEC 9899:1999(E)) states in 6.3.2.1/3

"Except when it is the operand of the `sizeof` operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type ''array of type'' is converted to an expression with type ''pointer to type'' that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined."

In this case the student is asking for the size of the array "a" (40 bytes) and the pointer p (4 bytes) – not the size of the object to which p points. The program is therefore reporting the correct values.

The C++ Standards (ISO/IEC 14882:1998(E)) states in 5.3.3/2

"When applied to a reference or a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The size of a most derived class shall be greater than zero (1.8). The result of applying `sizeof` to a base class subobject is the size of the base class type.) When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element."

You have to be careful about using `sizeof` with arrays. You have to be sure that you use it only at a point in the program where you are dealing with the actual array and not a pointer to its first element. In many, but not all, contexts an array `T a[N]` will implicitly convert to a pointer to its first element `T *p = a`. This is not the same as saying that an array is a pointer.

In C++PL (Special Edition) B.2.1 Bjarne Stroustrup notes that in C the `sizeof` a character constant and of an enumeration equals `sizeof(int)`. In C++, `sizeof('a')` equals `sizeof(char)` and a C++ implementation is allowed to choose whatever size is most appropriate for an enumeration.

Problem 2

The final `printf()` should refer to `hypo` not `&hypotenuse` (the address of a non-existent variable).

There is a missing return in `main()`.

The function `scanf()` can return a `double` into its argument, but the format flag should have been specified as %lf. By specifying a `float`, the results will be unpredictable.

`scanf()` is evil. As pointed out in the responses to SCC21, calling `scanf()` multiple times in the same module can lead to excess characters being left in the input buffer. Everything after the valid number (including the crlf) is left in the buffer. The second call to `scanf()` tries to interpret that according to the input format. The student would either have to program in assignment to a string for the remainder of the buffer or be introduced to the delights of non-assigning `scanf()` calls such as

```
scanf("%*[^\n]"); // skip to end of line.
scanf("%*1[\n]"); // skip a newline character.
```

to clear the input buffer.

If the student is determined to use `scanf()` then he/she should have checked the return code which specifies how many variables have been assigned.

The call to `pow()` to create a square is overkill and can be replaced by simple multiplication.

The attached code sample is a minimally improved version using non-assigning `scanf()` calls and another with the data gathering moved to a separate function.

```
#include <stdio.h>
#include <math.h>
#define BUF_SIZE 100
double getDouble(char * prompt);

int main() {
  double hypo, base, height;
  base = getDouble("Enter base");
  height = getDouble("Enter height");
  hypo = sqrt(base*base + height*height);
  printf("hypotenuse is %lf", hypo);
  return 0;
}
double getDouble(char * prompt) {
  double num;
  char buffer[BUF_SIZE];
  int gotDouble = 0;
  while(!gotDouble) {
    printf("%s :", prompt);
    if(fgets(buffer, BUF_SIZE, stdin)) {
      if(sscanf(buffer, "%lf", &num) == 1) {
        gotDouble = 1;
      }
      else {
        printf("Error in sscanf()\n");
      }
    }
    else printf("Error in fgets()\n");
  }
  return num;
}
```

The Winner of SCC 25

The editor's choice is:

Annamalai Gurusami

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

Student Code Critique 26

(Submissions to `francis@robinton.demon.co.uk` by March 10th)

It is generally worth sending in a late entry, depending on my work load it may or may not be considered for the prize but it will eventually get published. However this time please make an extra effort to get entries in on time as I would like to get this column done before the ECMA TG5 (C++/CLI binding) in Melbourne.

This time the problem is still about some student code however it is code that works but how do you help the student to move forward? Yes, I know the simple answer but I am looking for something more.

The following program (below my question) produces the results I want, but I am trying to create a `for`-loop to replace all of the `cout` statements.

What I have so far is:

```
for(int i = 0; i <= len; i++)
  cout << str[i + 1];
```

This will produce: "eed help with C++."

Could someone clue me in on how to create the `for`-loop? Should I be using a nested for loop? Any help would be appreciated.

Student's Program:

```
#include<iostream>
#include<cstring>

using namespace std;

void main (){
  const int arraySize = 20;
  char str[arraySize] = "Need help with C++.";
  int len = strlen(str);

  cout << "The sentence \"Need
                    help with C++.\" has "
     << len << " characters."
     << endl << endl;
  cout << str << endl;
  cout << str + 1 << endl;
  cout << str + 2 << endl;
  cout << str + 3 << endl;
// 13 similar statements snipped
  cout << str + 16 << endl;
  cout << str + 17 << endl;
  cout << str + 18 << endl;
}
```

Francis' Scribbles

by Francis Glassborow

Repository of Projects

We need to program in order to develop our programming skills. Anything more than the most trivial program takes time and effort. Most students (in the broadest sense of someone who is studying) find it hard to motivate themselves with projects whose end product is of little use or interest to them. It is much easier to put in the hours doing a job properly if the result is something we have a personal interest in.

However even the most capable teacher has a limited range of interests and domains of expertise. Those indulging in self-development are unlikely to even have the advantage of a teacher, let alone one who can suggest fulfilling activities that will both develop their skills and produce a useful end product.

Students often have difficulty with selecting an achievable objective from their own areas of interest because they simply do not know what programming can achieve. For example in my days as a teacher I often had pupils who wanted to write a program to play chess. They had little idea about how difficult it is to write such a program. However there are many potential programs round the topic of playing chess. Such simple ones as creating an electronic board can develop into tools of value. For example once you have an electronic chessboard you can feed it a file of a games you are studying, create alternative branches (getting some ideas about version control on the way) and easily backtrack to earlier moves.

I am in the process of creating a repository of potential programming projects (`www.spellen.org/youcandoit/projects`) as a resource for anyone studying programming in any language of their choice. By the time this is published enough should be in existence so that a visit will give you the idea of what is wanted for further development.

I am limited in the range of my interests and so need contributions from as far afield as possible. My hope is that everyone who reads this or who visits the site will contribute just one project suggestion. Contributors will be acknowledged.

This repository is a form of open source project without any source. It will only work if many people contribute a little to it. It helps if you have some idea about how hard the problem is and how demanding it will be of a program language. It also helps if you can identify one or more suitable places on the Web where relevant domain knowledge is available.

Let me be blunt, failure to contribute at least one project simply labels you as lazy because you cannot be a programmer without having some idea of at least one way that programming can be applied to a subject of interest to you. If you are only interested in programming (an unlikely event) there are a myriad ways that programming can be used to develop tools for programming.

Please note that I do not intend to publish source code for solutions on the site as that would degrade its value as a teaching resource. However I think that some way that instructors could obtain good source code in one or more languages might be useful. Anyone have any ideas as to how we could achieve that objective without spoiling projects for students?

If you teach programming, even on an informal basis, I hope you will find the repository useful for enthusing your students, useful enough that you will provide a link to it from any relevant pages that you manage.

The Myth of Homeland Security

This is a book (0-471-45879-1) about the various US government reactions to 9/11 (I still wonder if that date was chosen deliberately; 911 is the US emergency phone number.) It is well worth reading both because of the author's understanding of the US and because of his lack of understanding of the rest of the world and factual inaccuracies with regard to things external to the US (he thinks the Basque terrorists are French.)

If intelligent, well-educated US citizens who are willing to spend time researching on the Internet remain so profoundly US-centric we continue to have serious problems for which I can see little hope for solutions.

About eighteen months ago my wife and I had a problem when visiting the US. The cause of the problem was that they had no record of my wife having left after a previous visit so she was listed as having outstayed her six months on the visa-waiver program. Fortunately the date they had for her prior entry to the US was 1909 (yes, really). Worse, she had a new passport and so the old documentation was gone. It was so manifestly an error that it only took us half an hour to sort it out. But how does such a ludicrous error get into the system?

Part of the problem was highlighted on our most recent visit where no one collected the exit part of our visa-waiver, well, not until I drew their attention to this as we were about to board for the final leg of our return journey. Is it any wonder the best estimate for illegal immigrants in the US is in the millions?

Actually the whole green card visa-waiver procedure was (I do not know what it will be like the next time I visit in the Autumn this year) ludicrous in an electronic age. Clearly they should have been completed before embarkation and many of the details could be pre-entered by the airline (flight number, date, and points of departure and arrival). Indeed, with electronically readable passports the passport number should also be inserted automatically. Such pre-processing would be in everyone's interests. What airline wants to find that one of their passengers cannot enter the country of destination only after they have flown them there?

The processing of immigrants and visitors to the US is inconsistent with their belief that they are the most advanced technological society in the world. Requiring biometric passports is fine, but only if you have the infra-structure in place to manage data correctly and consistently. Those on the ground trying to manage the problems are generally courteous and thoughtful and can see what a mess is provided by inappropriate or incorrect use of technology. Those responsible for the systems are, at best, out of their depth.

Anyway, read the book but do not get too irate by the author's lack of understanding of how the rest of the world views US interventions.

C++ and the CLI

CLI stands for 'Common Language Interface' (I think) and is basically the .NET mechanism whereby processes written in different languages can interoperate.

Several years ago Microsoft produced something called 'Managed C++' which even their own C++ experts admitted was pretty awful. Recently Microsoft proposed that there should be a set of standard bindings from C++ to CLI. At one level this makes perfectly good sense. The trouble starts when new keywords and structures are necessary for C++ to support the CLI object model. In an ideal world we would modify CLI to better match major features of C++ (such as the concept of `const` qualified objects). However we do not live in an ideal world. We also have the problem that CLI is already a Standard (via ECMA) and is currently going through a revision. Obviously we want to have a major influence on that so that there is a better match with C++.

This leads to the problem that the 'correct route' to such a C++/CLI binding via a Technical Report from WG21 is too slow to have any chance of influencing CLI.

The 'solution' is to use ECMA as a fast mechanism, fast enough that we have some remote chance of getting some changes to CLI along the way. However the trouble with fast processes is the potential for mistakes along the way. This is quite worrying, not least because despite the successful efforts of Herb Sutter and Tom Plum to get the ECMA participation/liaison rules modified (or at least interpreted) so that National Body experts can have oversight of the proposals and input to them (even though without a vote) only the UK (plus France through a single expert) appears to have taken up the opportunities.

It is interesting to note that just about all the participants in TG5 (the ECMA group dealing with this) are actually also members of WG21. The real problem is getting TG3 (responsible for CLI) to listen and respond. This is another problem related to the fast timetable that ECMA groups tend to set. Even when you are using electronic communications getting consensus is time-consuming.

My biggest concern is that we know from bitter experience that getting things right takes time, and even when we think we have managed it we are too often mistaken. The experts reading this might like to think about the problems with ADL (argument dependent lookup, sometimes called Koenig Lookup) and the problems that both C++ and Java have had with exception specifications (even though they tried different approaches).

For example, in my opinion, even when we want to provide something such as 'interface classes' in the C++/CLI binding and use a keyword for the purpose we should stick to the underlying C++ syntax (properly declared pure virtual functions) and just use the keyword to instruct the compiler to diagnose breaches of the CLI requirements (i.e. in this case, only public members, no data etc.) The result of omitting 'interface' should mean as near as possible the same thing in pure C++.

I am sure that the result of the UK participation through the BSI will be that the work being done to support CLI in C++ will be much better. It is only a pity that there are not more people involved in the effort. Despite what I read in one book recently, Standards are not dead and work on them does improve the ordinary user's lot.

Australia

I will be in Australia for the last two weeks of March and for the first week of April. I will start in Melbourne and attend the ECMA TG5 (C++/CLI)

meeting there before moving over to Sydney for the WG21 (C++) meeting followed by WG14 (C) which is the reverse order to the normal one.

It is disappointing that after many of us have made a special effort to go to Australia, it seems that few if any Australian experts will make the effort to attend the meetings. Worse, the Australian National Body is no longer even an O (for observer) member of SC22 which is the parent committee of WG21 and WG14. I can think of some other places I would like to visit if local interest in Standards is not required.

The other issue is that those dates are exactly the time when I would normally be working on my contributions to the next issue of C Vu. Please make a special effort to get your contributions in early because I would like to get as much as possible done before I leave for Australia. I know I can do much of the work on a laptop but I am happier working with my reference books close to hand.

My Book

It went to reprinting five weeks after its release in the UK. US book chains have been upping their orders even before it goes to distribution there (on February 9th). As the first reprint was before a single review had been published, clearly many of you have been doing your bit to tell others about it. Please keep up the good work.

Problem 13

Look at the following C code. Why does `qsort()` fail? Please note that the `compare()` function does rank any two objects of type `X`.

```
struct X{
  int i;
  int j;
  int k;
};
int compare(void * p1, void * p2){
  struct X * x = p1;
  struct X * y = p2;
  if(x.i > y.i) return 1;
  if(x.j > y.j) return 1;
  if(x.k > y.k) return 1;
  return x.i + x.j + x.k – y.i – y.j – y.k;
}

int main(){
  struct X array[10];
/* code initialising array */
  qsort(array, sixeof(X), 10, compare);
/* etc. */
  return 0;
}
```

Commentary on Problem 12

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

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

I wonder how long you stared at that code before the penny dropped? Even given the advantage of seeing that the result was of writing out a palette and then reading it back resulted in a completely different palette it took me an embarrassing time to realise what the problem was. In a way I was lucky that the error was so visible and that it happened immediately with the compiler I was using.

The problem is that the `HueRGB` constructor takes three arguments. The fact that they are all identical is not important except that it guarantees that nothing has a chance to detect the error other than the human eye.

Remember that the order of evaluation of sub-expressions is unspecified. The arguments of a function call can be evaluated in any order. The compiler I was using (the MinGW version GCC) evaluates arguments right to left. The data was written to the file left to right. The result was that red and blue values were exchanged by the process.

Elegant though the `canvas.set_entry` might seem to some, it is fatally flawed by trying to do too much in one gulp and must be replaced by something such as:

```
int const r(read<int>(in));
int const g(read<int>(in));
int const b(read<int>(in));
HueRGB const p(HueRGB(r, g, b));
Canvas.set_entry(i, p);
```

It is generally better to evaluate the arguments of a function call like this as it avoids the possibility that an order of evaluation will arise. Making all the intermediate values const variables allows the compiler to optimise but keeps control of the order of evaluation.

Cryptic

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

The result if Tao were valued in Greece.

Counting wild oats in a singularly Greek fashion.

And for the intended value (271):

A Greek tap is particularly apt in a numerical way.

A Greek tap is numerically indistinguishable from a pat on the head.

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

Christmas Competition

So far I have had two entries but the actual selection of prize winners will have to wait till the deadline (either the production editor will manage to sneak it in at the last minute or the announcement will be in the next issue.

I have this from John Kewley:

Txt me "happy" one Christmas!

I'll try and think of something less seasonal for 3 * 3 * 17 * 27960259

[Try: *The text of Happy Xmas*. Francis]

And Richard Blundell sent me this:

OK, how about this for a clue for your number:

"Bubbly hooligans phone to wish seasonal greetings (10 digits)"

The "seasonal greetings" bit is obviously the phrase you were encoding, namely "Happy Xmas". But an alternative answer (at least on my phone) is also "Gassy Yobs", hence the first part of my clue.

And that really gets into cryptic clues by giving the answer two ways.

Thanks to both for making the effort to come up with good clues and remembering to brighten my day by sending them in.

Time For

Another form of numerical clue is to use the time or date of some event that is well known to the readership. Clues actually using a time tend to be very much local to groups but dates can be more general:

When next Julian has one that Gregory skips. (4 digits)

Francis

Comment on "Problem 11"
(Francis' Scribbles, C Vu 15.5/15.6)

Bill Clare <BillClare3@aol.com>

The first step here in finding problems in the code is to identify the problem the code is trying to solve. The discussion in the C Vu article is basically about curiosities in the way in which the C++ standard library `std::istream` is defined, but I will make the perhaps unwarranted assumption that what the problem the code is really about is not the uses of `std::istream`, but rather, more generally, how to write a read routine that can effectively and safely capture data from an input stream. Actually as the first problem below illustrates neither of these issues can be effectively addressed without the other.

Problems

The proposed improvement to the templated `read` function is that it starts an approach to handling different input conditions by having the user distinguish between two types of stream ending conditions, reading just an end-of-file and reading a carriage return along with end-of-file. (Do I have this right?)

This is a start, but only useful to illustrate idiosyncrasies of STL `istreams`. It still has problems with `std::istream`, but as a lesson in reading computer input it is deficient in the following ways:

- The most basic problem here is that of "separation of concerns" and for separate routines that each do one function and do it well. This is particularly unfortunate here, since it is especially important to avoid tight coupling between system support routines (reading input) and client application routines (processing input).
  This basic problem is manifest here in multiple ways:
  - The client routine is expected to test multiple stream ending conditions, reported with different syntax and in two different domains; one in that of the input mechanism, one in that of the read routine.
  - The test for a dummy value is a clever, but is, at best, an awkward and somewhat dubious general approach of detecting particular conditions (should we perhaps label this a hack?).
  - Such approaches can easily lead to error prone code.
    As implemented here, the two conditions to test are redundant, since a dummy value has to be returned for end-of-file, whether a carriage return was present or not. Thus not only is the client code overly complex, but the strategy is faulty. Also, if the "dummy value" actually happens to be present in the input stream, it will indeed be treated as is any other value.
  - Detecting different ending conditions is relevant to the input processing domain; processing different ending condition is relevant to the client domain.
  - Testing multiple conditions in multiple ways will not scale well, when other conditions are considered. The example considers a special case, but, with slight extension for instance, the read routine might be adapted easily to process console output directed to a file, where there may be end-of-line, and possibly carriage return characters, separating data items.
- The error handling is rigid with no flexibility for adaptation to either the application environment or the client needs.
  The read routine throws an exception for stream errors; but even worse the routine buries its own private `fgw::bad_input` exception. On the other hand, the client routine may well wish to continue processing for bad input, which may be either unreadable for the specified type (input stream domain failure) or invalid (either as defined in the data, the read routine or the client processing domain).
- The `in.bad()` condition is not tested, which is the one more deserving of an exception. Actually for a pre-standard library the `fail` bit may cover this case. But then, the read routine would throw a `bad-data` exception, when the error actually is failure to read the data, whether good or bad.
- For beginners especially, the code fails to take a valuable opportunity to demonstrate basic and consistent mechanisms for preventing invalid data values from getting past the application external interfaces.
- In any case, there needs to be consistent support for applying both general overall application, as well as client routine specific policies for both error handling and for error reporting. Developing those policies is another subject, but the basic interfaces can be made reasonably simple and crucial.
- The input data appears to be constructed twice, once in the read routine and once in the client routine, and probably with different constructors. Typically this may not actually be a problem, but this behavior can lead to subtle problems.
- If, as suggested here, the client code needs to be abstracted from the details of `std::istream` error conditions, why have any dependency on `std::istream`? Perhaps, even more useful than templating the input data type, is abstracting the concept of an input source.
- Names are critical. Here the routine does not read the input stream; it reads the next item in the input stream. Hence the routine could be called `readNext`.
- A simple, but important, advantage of abstracting the input source type is that now the function of the routine is not merely `readNext`, but more generally `getNext`.
  And, we already have a powerful and applicable mechanism in C++ for `getNext` processing – in the form of iterators, which are applicable here.
- The routine is at too low a level for many uses, forcing the client to devise one of many possible iteration constructs. In the face of multiple exit conditions, these are too often error prone.
- The routine can only read input of one data type. This is appropriate for "self-defining" streams, which, for instance, provide tokens to identify the next item in the stream. There are numerous other approaches to data type extension, probably well beyond the intent here, but the applicability and limitation should at least be noted.

Solution Steps

The problem issues above can be addressed systematically in a series of steps. These are not all meant for one lesson, but each is straightforward enough, even for beginners. They are all also invaluable in their own rights for other problems. In fact, the process here goes far towards an objective of teaching programming based on principles and practices, rather than just belabouring syntax and semantics.

1. Provide a status variable parameter, which reports all conditions that the application may or may not want to consider.
   In its simplest form this a string of bit flags, although supplementary data about the condition may be of interest also. A higher level might introduce predicates, such as `status.isValid()`.
2. Rather than directly reporting the failure codes particular to a specific source, conditions need to be mapped to categories of concern to the client.
   Here, some such conditions might include: invalid parameters (e.g., invalid port or URL), inaccessible input, un-initialized (e.g., un-opened, un-connected ported) input or un-initializable input (e.g. open or connect failures), insufficient security permissions, source failure, source warning, unreadable data, special delimiter (carriage return, end-of-line, white-space, other), invalid data, along with provision for two or three additional conditions to be used for specific implementations.
3. Allow the interface to set the conditions to abort on, to return to the user, or to just skip over, and the conditions to be reported to the application environment in any case.
4. Parse all errors reported by the source.
5. Issues of memory management, references, pointers, multiple constructors – with possibly different behaviour, and data object copying, all rear their awesome heads here as elsewhere. Better, and simpler, is for the client routine to specify where the data is to go.
6. Use the convention of returning a `null`, or invalid `end()`, pointer, rather than attempting to define dummy values. Think of all the fun, the C convention of terminating strings with `\0` has caused.
7. Use a template parameter for the input source type as well as the data type, and introduce template specialization to show `std::istream` handling.
   Parameterizing the input source type is important, since it is, or should be, an incidental focus of the application routine. In particular, consistent handling of all input sources is invaluable for an application and makes possible extensions to files, communication protocols, database interfaces, GUIs, and sequences in general.
8. Represent the source as a forward iterator parameter that wraps either the actual source or an existing iterator.
   It is useful to illustrate a complete templated iterator solution, but it is only necessary to develop details for the basic template components, and here only for `std::istream`. The rest can be left for reference to the standard definitions. On the client side, `begin` and `end` iterators, `for`-loops, and dereferencing idioms are simple and natural.
9. A fundamental extension, is for the template code to test both the input source parameter type and input data parameter type for `isValid` routines, and use these to check the input data values.

10 Both the error status conditions and the exception flags are now better included in the iterator template class, rather than the function parameter list.

11 Have the template code also test the iterator parameter type for an `onError` interface and report errors to that interface.

12 Actually there are two parts (handling and reporting) to an `onError` routine and hence the possibility for two routines:
- The first maps the conditions from a particular environment into the more general client interface. It may also need to set a flag to indicate if resuming input is possible and providing such a mechanism.
- The second, which may be part of the input routine itself, passes information identifying the details to a common higher level application reporting mechanism, for appropriate logging and recording.

13 A small, but valuable generalization is to look for an input mapping routine in an interface borne by the iterator. This allows data types and values in the input domain to be directly transformed to data types and values in the client domain.

14 Similarly a filter routine can be used, if present, to bypass unneeded source data.

15 Illustrate support for `to_string` and `from_string` serialization routines, for use with operators `<<` and `>>` for derived types.

16 When adapted for output, the iterator can also contain formatting flags and delimiters.

17 This leads to raising the level of the routine.
Better, for many but not all purposes, would be a copy routine (or move routine, if the input is consumed) following the STL syntax – here, with `end()` to be set for the iterator return of conditions flagged by the caller. For some applications, which need a lower level involvement in handling special conditions, selected `end()` conditions can be processed by the client routine, with `begin()` used to allow an attempt at resumption of input.

And these seventeen progressive steps, I think, provide an outline of a fairly complete solution to the problem of creating a code structure for simply, safely, and effectively transferring input data into an application framework, and by simple extension output data (the homework exercise?). Various interfaces can be made more general and more sophisticated as necessary, without impact on client code. Alternatively, if client code needs to adapt to additional conditions this can be added in a consistent and compatible manner.

Lessons

The final result, or outline for a result, is considerably more complicated than the initial small example, but there are many valuable pedagogical reasons for developing it. In particular, it should be emphatically taught when not to use code that is error agnostic.

The fundamental lesson here is that there is a considerable difference between production code and code for beginning exercises or prototyping. This is easily spouted as a general principle, but is difficult to teach effectively. The sample problem here provides an ideal basis for illustrating this issue systematically and indicating approaches to dealing with it.

The next most fundamental lesson is to assign responsibilities appropriately, then to design interfaces that handle the responsibilities, and finally to allow flexibility by providing mechanisms to delegate responsibility for policies appropriately. Here there are separate responsibilities in several places:
- for the input routine, in being complete in some definable sense,
- for the client interface, in specifying a request,
- for a higher level routine, in parameterizing the request according to design parameters and constraints,
- for the input data class, in maintaining consistency and integrity constraints according to class invariants,
- for consistent error handling and reporting policies at the application level, and for flexibility for appropriate interventions by the using client.

Understanding tradeoffs of where and how to apply generality, simplicity, ease of use, and allowance for specific conditions is fundamental. The solution should illustrate use of templates, constructors, default parameters, and environment variables and routines (including exception handlers), as appropriate, to design and apply constraints and policy.

Also fundamental, is the realization that error handling is basic for any significant code that is to actually be employed for useful purposes. By analogy perhaps, with a numerical analysis computation, the result is generally not of value, other than as a guess at usefulness, unless error analysis has been performed to determine how good the result actually is.

One basic tenet about error handling, that emphatically applies here, is that applications need to catch all erroneous inputs at the external interfaces. This can then limit significantly the data validity testing needed later.

Since the student will undoubtedly be exposed to them, the lesson might include tradeoffs in various approaches to error returns through special values, (e.g., `end()`), through `pair`s, through bit flags, through status objects, through exceptions, etc. The lesson can emphasize the dangers, particularly for critical application interfaces, of starting with more limited approaches that are inflexible and that do not scale.

The final result may seem more complex than needed for what seems like a simple problem, but I would respectfully disagree with the premise. The problem posed is not trivial; and ignoring basic issues makes for an incomplete solution, not a simple solution. Reasonably simple solutions can still be arrived at by dealing with each issue separately and appropriately.

Techniques

The lessons here are general but the implementations, if they are to be illustrated in C++ code, are admittedly non-trivial. As examples though, the techniques can be easily taught as idioms, to be imitated, and these idioms are also useful in many broader contexts.

From a teaching and learning perspective, there are only two roads to writing useful code in C++. The first is to understand the C++ language and library standard, and particular compiler deviations from it in detail (not particularly to be recommended), The second is by extensive reading and following of useful models (which is what all the worthwhile C++ beginner and intermediate texts provide). Ideally this accomplished with a mentor.

The basic techniques here include:
- Basic bit flag masks to indicate status or state; supported by `enum`s that are powers of two, operations on sets of flags, and by `status.isXXX()` type predicates.
- Rudiments of exception handling.
- Type generalization through templates, with basic template specialization.
- STL iterator concepts, at at least a high level, and their use in general algorithms such as `copy`.
  In particular, a strong preference, if only for consistency, for using STL constructs and concepts where appropriate can be inculcated. For instance, encapsulating iteration (here `copy`) in a library routine, rather than using a variety of `for`, `while` and `do` constructs is worthwhile.
- Parameterization options through template parameters, `typedef` statements, constructor arguments, default function parameters and environment support (here, at least, exception handlers).
- Testing types and objects for extended interfaces through compile time (template based) and run time (dynamic cast) techniques. Here, the solution tries to allow existing data objects and iterators to be used, but takes advantage of additional capabilities if provided.
- And yes, idiosyncrasies of various input mechanisms also can be explored.

Perhaps the final lesson is my perception of C++ as a really ugly tool for developing beautiful constructs. As one mentor, once said, "You don't ask a cow why it works the way it does, you just learn to milk it."

Summary

The goal of making C++ more accessible to novices is admirable, but oversimplifying the issues does not appear useful; nor does dwelling on details of `std::istream` to the exclusion of more basic issues.

The discussion above leads to approaches to that goal on two levels:
- At the client level, the final copy routine is indeed simple, and can illustrate the power of the tailoring mechanisms to provide a significant range of underlying functionality including: comprehensive handling of unusual conditions, full reporting of error conditions, the ability to adapt to any input source, the ability to map data from different sources to common types, scaling, formats and representations, and the ability to filter extraneous input.
- At the development level, the analysis of problems and solutions illustrates both design considerations needed for building code that can adapt to a broad range of application needs, as well as coding considerations in the use of C++ facilities for accomplishing this. This surely is a worthwhile introduction to what programming is all about.

Bill Clare

Features

do...while (more_to_say())

James Dennett <jdennett@acm.org>

What can be said about C's everyday `do...while` loop? It just does *something* while *some condition* holds. End of story, right?

No, of course not. That would make the title of this small article silly, so let's cover two topics.

Firstly, the description above of what `do...while` does is wrong, for one simple reason. `do...while` always does its *something* at least once. It would be better described as:

> "do the thing once, and then continue doing it as long as some condition is met".

That added complexity makes formal reasoning about programs using `do...while` more difficult; their invariants can easily end up being of the form:

> "x always holds, and y is guaranteed to hold except for the first time around".

A `while` loop, by testing its condition at the top of the loop, gives simple invariants. This isn't just abstract nonsense (we're not talking about category theory here). With practice, designing programs around `while` loops as the default choice really does make for simpler, more robust code. Only when you find the code looks like:

```
do_x();
while (condition()) {
  do_x();
}
```

should you consider re-writing it as the equivalent `do...while` loop:

```
do {
  do_x();
} while (condition());
```

Secondly, let's talk about idioms. Actually, about idioms and anti-idioms. To get the audience on my side, I'll say that I don't like code using `goto` statements. I might accept that one time in a million they provide a real advantage, but to me the consistency of a simple, absolute ban on their use pays off many times over. So, we're all together here, no `goto` statements allowed.

Now for the anti-idiom. On hearing that coding standards were commonly disallowing `goto` statements, some ingenious programmers found a way around the restriction (at least for coding standards that didn't also ban `break`ing out of loops). Here's their trick to write what is in effect a goto without typing '`g`', '`o`', '`t`', '`o`':

```
do {
  function1();
  if (condition1())
    break;
  function2();
  if (condition2())
    break;
  function3();
} while (FALSE);
```

(assuming, of course, that `FALSE` is a constant evaluating to `0`). Hang on a minute... `do while(FALSE)` sounds a bit like `if (FALSE)` – a way to stop code running. Except that (did I mention this above?) `do...while` always executes its body at least once. So `do while(FALSE)` is an odd way of saying "please run this statement at least once, and not more than once". In other words "please run this (possibly compound) statement". In this situation `do...while` is not being used as a loop, but only so that `break` statements can be used to simulate `goto`s. Restricted, forward-only `goto`s, to be sure, but they're still `goto`s. When you write code using a control structure such as `while`, `for` or `do...while` intended for looping, make sure that looping is the concept you really wanted to communicate to the (mostly human) readers of your code. Don't use `do...while` when you mean `goto`.

Except in one situation. Take a look at the following code, which defines a function-like macro (by the name of `MACRO`, to showcase my imagination) which is intended to be used like a function. Here is the idiomatic use of `do while(FALSE)`.

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

#if defined DEFINE_A_BROKEN_MACRO
#define MACRO(x) \
{ printf("Bad"); printf("\n"); }
#else
#define MACRO(x) \
do { printf("Good"); printf("\n");} \
while (0)
#endif

int main()  {
  if (0)
    MACRO(x);
  else
    MACRO(y);
  return EXIT_SUCCESS;
}
```

(To the pedants: yes, I know that the `return` statement is optional according to the current C standard, but (a) almost no compilers yet implement the current C standard, and (b) it's good to be explicit. That's also why I write:

```
return EXIT_SUCCESS;
```

instead of just:

```
return 0;
```

even though any programmer competent with C should know that returning `0` from `main` is another way of reporting the successful exit status of a program. It's more explicit.)

This program fails to compile if the first `MACRO` definition is used, but compiles quite happily if the second is used. To use the "broken" macro successfully, one of two strategies can be followed. The first is to omit the semi-colon in the

```
MACRO(x);
```

lines, making them read

```
MACRO(x)
```

which looks somewhat unnatural. The second is to stick to a rule that says that you always use `{ }` in your if statements. If that is done, so that the `if...else` loop reads

```
if (0) {
  MACRO(x);
} else {
  MACRO(y);
}
```

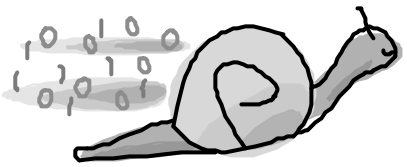
then either of the macro definitions will work without problems.

James Dennett

Professionalism in Programming #24

The need for speed (part one)

Pete Goodliffe <pete@cthree.org>



There is more to life than increasing its speed

Mahatma Gandhi

We live in a fast food culture. Not only must our dinner arrive yesterday, our car should be fast, and our entertainment instant. Our code should also run like lightning. I want my result. And I want it *now*.

Ironically, writing fast programs takes a long time.

Optimisation is a spectre hanging over software development, as W.A. Wulf observed. *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.*

It's a well-worn subject, with plenty of trite soundbites bounding around, and the same advice being served time and time again. But despite this, a lot of code is still not developed sensibly. Programmers get sidetracked by the lure of efficiency and write bad code in the name of performance.

In these articles we'll address this. We'll tread some familiar ground and wander well-worn paths, but look out for some new views on the way. Don't worry – if the subject's *optimisation* it shouldn't take too long...

What does it mean?

The word optimisation purely means to make something better; to improve it. In our world it's generally taken to mean 'making code run faster', measuring a program's performance against the clock. But this is only a part of the picture. Different programs have different requirements; what's 'better' for one may not be 'better' for another. Software optimisation may actually mean any of the following:

- speeding up program execution,
- decreasing executable size,
- improving code quality,
- increasing data throughput (not necessarily the same as execution speed), or
- decreasing storage overhead (say, database size).

The conventional optimisation wisdom is summed up by M.A. Jackson's infamous laws of optimisation:

1. Don't do it.
2. *(for experts only)* Don't do it yet.

That is, you should avoid optimisation at all costs. Ignore it at first, and only consider it towards the end of development when your code's shown not to be running fast enough.

In reality this is far too simplistic a viewpoint – accurate to a point, but potentially misleading and harmful. Performance is really a valid consideration right from humble beginnings of development, before a single line of code has been written.

Code performance is determined by a number of factors, including:

- the execution platform,
- the deployment/installation configuration,
- architectural software decisions,
- low level module design,
- legacy artifacts (like the need to interoperate with older parts of the system), and
- the quality of each line of source code.

Some of these are fundamental to the software system as a whole, and an efficiency problem there won't be easy to rectify once the program has been written. Notice how little impact individual lines of code have, there is so much more that affects performance. Optimisation, whilst not a specific scheduled activity, is an ongoing concern through all stages of development.

Think about the performance of your program from the very start – do not ignore it, hoping to make quick fixes at the end of development. But don't use this as an excuse to write tortured code, based on your notion of what is 'fast' or not. A programmer's gut feeling for where bottlenecks lie is seldom right, no matter how experienced he or she is.

What makes code suboptimal?

In order to improve our code, we have to know the things that will slow it down, bloat it, or degrade performance. Later on this will help us to determine some code optimisation techniques. At this stage it's just helpful to appreciate what we're fighting against.

Complexity

is a killer. The more work there is to do, so the slower the code will run. Reducing the amount of work to do, or breaking it up into a different set of simpler, faster, tasks can greatly enhance performance.

Indirection

is touted as the solution to all known programming problems, but also blamed for a lot of slow code. This criticism is often levelled by old-school procedural programmers, aimed at modern OO designs. Whether any of it is actually true is debatable.

Repetition

can often be avoided, and will inevitably ruin code performance. Repetition can often be avoided, and will inevitably ruin code performance. It comes in many guises; for example, by failing to cache the result of expensive calculations or of remote procedure calls. Every time you recompute you waste precious efficiency. Repeated code sections extend executable size unnecessarily.

Bad design

will lead to bad code. For example, placing related units far away (say across module boundaries) will make their interaction slow. Bad design can lead to the most fundamental, the most subtle, and the most difficult performance problems.

I/O

is a remarkably common bottleneck. A program whose execution is blocked waiting for input or output (to/from the user, the disk, or a network connection) is bound to perform badly.

This list is nowhere near exhaustive. But it gives us a good idea of what to think about as we proceed to investigate how to write optimal code.

Why not optimise?

Historically optimisation was a crucial skill, since early computers ran very, very slowly. Getting a program to complete in anything like reasonable time required a lot of skill, and the hand-honing of individual machine instructions. That kind of skill is nowhere near as important these days; the personal computer revolution has changed the face of software development. We often have a surplus of computational power, quite the reverse of the days of yore. So it would seem that optimisation doesn't really matter any more.

Professionalism in programming?

Another new year, another new volume of C Vu – it's time to remind ourselves what this column is all about. What does the *Professionalism in Programming* title mean?

The aim of this column is to investigate the apparent oxymoron of the ACCU's mission statement: *to promote professionalism at all levels of programming*. These articles are aimed at both people who consider themselves 'professionals' and people who aspire to be professional.

I want to impart skills, but more than this: to impart wisdom. These articles don't focus on the single word 'professionalism', but discuss topics ranging from coding style, to working practices, to our attitudes. Maybe some of the topics don't seem directly related to the banner 'professionalism' (or at least no more than any of the other C Vu articles). However, what I am trying to pull out is the professional perspective that we should adopt behind each particular issue.

I'm grateful to those readers who have taken the time to comment and reply to what I'm writing here. Any author appreciates hearing from their readers – it proves that there's actually someone reading! I hope you continue to enjoy this series. Feel free to email me with comments, suggestions, or encouragement.

Pete

Well, not quite. We'll see that there are still many situations requiring high performance code, but it is preferable to avoid optimising code if at all possible. Optimisation has a *lot* of downsides.

Lightning performance and heavy optimisations are seldom as important as you think – it's either acceptable to put up with 'adequate' performance, or you can work around performance issues in other ways (more on this later). Before you even consider a stint of code optimisation, you must bear this advice in mind: *Correct* code is far more important than *fast* code. There's no point in arriving at the wrong answer quickly.

You should spend more time and effort proving that your code is correct than getting it fast. Any later optimisation must not break this correctness.

Presuming the code wasn't absolutely terrible in the first place, there's a price to pay for more speed. Optimising code is the act of trading one desirable quality for another. Done well, the (correctly identified) more desirable quality is enhanced.

These are the top reasons to avoid optimising code. You'll see that a number of them are examples of code writing tradeoffs:

Loss of readability

It's rare for optimised code to read as clearly as it's slower counterpart. By it's very nature, the optimised version is not as direct an implementation of the logic, or as straightforward. You sacrifice readability for performance.

Most optimised code is ugly and hard to follow. Optimisation destroys neat design. Here is a simplistic example: you'll see plenty of C code constructs like this: `int value = *p++`. It's hard to read, sadly even for experienced C programmers. There's nothing wrong with separating that into two statements like: `int value = *p; p++;`. The initial version may be more concise, but the second version is far, far clearer to read and understand.

This is a simplistic example for two reasons. First, it's a code construct issue. Most optimisations are concerned more with logic than syntax. Second, whilst this might have generated more efficient code in the Good Old Days (when dinosaurs wrote preprocessors) modern optimising compilers will generate identical code for both versions. However, the general principle is clear.

Increase in complexity

A more 'clever' implementation – perhaps utilising special 'backdoors' (thereby increasing module coupling) or taking advantage of platform specific knowledge – will add complexity. Complexity is the enemy of good code.

Hard to maintain/extend

As a consequence of increased complexity and a lack of readability, the code will be harder to maintain. If an algorithm is not clearly presented the code can hide bugs more easily.

Optimising working code is a surefire way to add new subtle bugs – these will be harder to find because the code is more contrived and harder to follow. Optimisation leads to dangerous code.

This also stunts the extensibility of the code. Optimisations often come from making more assumptions, limiting generality and future growth.

Introducing conflicts

Often an optimisation will be quite platform specific. It might make certain operations faster on one system, at the expense of another platform. Picking optimal data types for one processor type may lead to slower execution on others.

The software world moves fast. Technologies change rapidly; today's optimisation might be tomorrow's bottleneck. But gnarly optimised code hides the original algorithm's intent, so it will be hard to unpick the optimised code.

More effort

Optimisation is another job that needs to be done. We have quite enough to do already, thank you. If the code's working adequately then we should focus our attentions on more pressing concerns.

Optimising code takes a long time, and it's hard to target the real causes. If you optimised the wrong thing, you've wasted a lot of precious energy.

Too expensive/unnecessary

Often optimisation is not really worthwhile, or uneconomical. A few extra percent points in speed trials doesn't justify a year's extra work.

Inappropriate

Do you really believe that you can optimise better than a modern compiler's optimiser? Trying to perform code level tweaks can be a big waste of time.

For these reasons, optimisation should be some way down your list of concerns. Balance the need to optimise your code against the requirement to fix faults, add new features, or to ship a product. If you take care to write efficient code in the first place you're less likely to need to optimise anyway.

Alternatives

Often code optimisation is performed when it's actually unnecessary. There are a number of alternative approaches that we can employ to avoid destroying code. Consider these solutions *before* you get too focused on optimisation:

- Can you put up with this level of performance – is it really *that* disastrous?
- Run the program on a faster machine. This seems laughably obvious, but if you have enough control over the execution platform it might be more economical to specify a faster computer than spend time tinkering with code. Given the average project duration, you are guaranteed that by the time you reach completion processors will be considerably faster.
  Not all problems can be fixed by a faster CPU, especially if the bottleneck is not execution speed – a slow storage system, for example. Sometimes a faster CPU can cause drastically worse performance; faster execution can exacerbate thread locking problems.
- Look for hardware solutions: add a dedicated floating point unit to speed up calculations, add a bigger processor cache, more memory, a better network connection, or a wider bandwidth disk controller.
- Consider reconfiguring the target platform to reduce the CPU load on it. Disable background tasks, or any unnecessary pieces of hardware. Avoid processes that consume a huge amount of memory.
- Run the slow code asynchronously, in a background thread. Adding threads at the last minute is a road to disaster if you don't know what you're doing; but careful thread design can accommodate slow operations quite acceptably.
- Work on user interface elements that affect the user's perception of speed. Ensure that GUI buttons change immediately, even if their code takes over a second to execute. Implement a progress meter for slow tasks; a program that hangs during a long operation appears to have crashed. Visual feedback of operation progress conveys a better impression of the quality of performance.
- Design the system for unattended operation, so that no one notices the speed of execution. Create a batch processing program with a neat UI that allows you to enqueue work.
- Write time critical sections in another faster language – conventional compilers still beat JIT code interpreters for execution speed.
- Try a newer compiler with a more aggressive optimiser, or target your code for the most specific processor variant (with all extra instructions and extensions enabled) to take advantage of all performance features.

Why optimise?

So that's it – we should all give up on any foolish notion of optimising code, and put up with mediocre performance, or only ever attempt round-about solutions? Well, not quite...

There are plenty of situations where optimisation is important. And contrary to the popular wisdom, some areas are guaranteed to require optimisation:

- Games programming always needs well honed code. Despite the huge advances in PC power, the market demands more realistic graphics and more impressive artificial intelligence algorithms. This can only be delivered by stretching the execution environment to its very limits. It's an incredibly challenging field of work; as each new piece of faster hardware is released, games programmers still have to wring every last drop of performance out.
- DSP programming is all about high performance. *Digital Signal Processors* are dedicated devices specifically optimised to perform fast digital filtering on large amounts of data. If speed didn't matter you wouldn't be using a DSP. DSP programming generally relies less on an

[concluded at foot of next page]

Code in Comments

Thomas Guest <thomas.guest@ntlworld.com>

We have all seen comments in source files which look more like executable code than documentation.

The first line in the body of the `for` loop below is such a comment: you might expect to be able to remove the leading slashes and have code which compiles and runs, but functions slightly differently.

What did the author of this comment intend?

Example 0

```
for (Surfaces::iterator sf = surfaces.begin();
     sf != surfaces.end();
     ++sf) {
  // std::cout << "Drawing: " << *sf << "\n";
  sf->draw();
}
```

OK, I'm being disingenuous. I'm aware that the comment isn't really a comment, it's commented-out code. And, like any tolerant and capable programmer, by examining the surrounding context I can guess why this code has been commented out.

This article examines how to comment out code, then describes various problems which lead to code being commented out, before finally arguing that there's often a better solution to these problems.

How to Comment Out Code

Usually it's as simple as using your editor to select a region then instructing it to comment out that region.

If using C-style comments – by which I mean comments delimited by `/*` and `*/` – then bear in mind they do not nest, so you may run into problems with real comments in the code you want to comment out, or even with commenting out code which has already been commented out.

I came to C++ from a C background, and remember attending a training course at which the presenter pointed out how easy it was to comment out C++ comments using C comments:

```
/*
for (Surfaces::iterator sf = surfaces.begin();
     sf != surfaces.end();
     ++sf) {
  // std::cout << "Drawing: " << *sf << "\n";
  sf->draw();
}
*/
```

Oh dear!

If your editor's syntax highlighting makes it obvious that the whole loop is commented out, great. Would it still be obvious if, for example, you were viewing the source file on a remote computer via a telnet session? Or if you were code-reviewing a printed version of the file? Or if you had hit a commented-out line while searching? Or even if you were at a customer site and didn't have access to your favourite editor?

The more useful thing to say about C++ comments is that they do not delimit regions and therefore cannot nest. So a C++ comment can be commented out by a C++ comment!

```
// for (Surfaces::iterator sf
//                         = surfaces.begin();
//      sf != surfaces.end();
//      ++sf) {
//   // std::cout << "Drawing: " << *sf
//   //           << "\n";
//   sf->draw();
// }
```

If using C-style comments, then the following style makes it clear which lines are part of the comment:

```
/* for (Surfaces::iterator sf =
surfaces.begin();
 *      sf != surfaces.end();
 *     ++sf) {
 *   // std::cout << "Drawing: " << *sf
 *   //           << "\n";
 *   sf->draw();
 * }
 */
```

If your editor does not allow you to easily comment out a lengthy region of code in this way then either use a better editor or read the rest of this article and see if commenting out the code is really what's required.

A more heavy duty way to stop a block of code from executing is to instruct the preprocessor to skip past of it.

```
#if 0
for (Surfaces::iterator sf = surfaces.begin();
     sf != surfaces.end();
     ++sf) {
  // std::cout << "Drawing: " << *sf << "\n";
  sf->draw();
}
#endif
```

If this technique is used, the preprocessed-out code blends perfectly with the executable code. Even syntax highlighting does not expose the fact that the code will not be executed.

More Examples

Example 1

```
void Session::registerClient(Client const &
                             /* client */) {
  // if (!registered(client)) {
  //     m_clients.push_back(client)
  // }
}
```

[continued from previous page]

optimising compiler, since you want to have a high degree of control over what the processor is doing at all times. DSP programmers are skilled at driving these devices at their maximum performance.

- Resource constrained environments, like deeply embedded platforms, can struggle to achieve reasonable performance with the available hardware. You'll often have to rewrite code to achieve an acceptable quality of service.
- Real time systems rely on timely execution, on being able to complete operations within well specified quanta. Algorithms have to be carefully honed and proven to execute in fixed time limits.
- Numerical programming – in the financial sector, or for scientific research – demands high performance. These huge systems are run on very large computers with dedicated numerical support, supporting vector operations and parallel calculations.

Perhaps optimisation is not a serious consideration for 'general purpose' programming, but there are plenty of cases where optimisation is a crucial skill. Performance is seldom specified in a requirements document, yet the customer will complain when your program runs unacceptably slowly. If there are no alternatives, and the code doesn't run fast enough, you have to optimise it.

Clearly there is a shorter list of reasons to optimise than not to. Unless you have a specific need to optimise, you should avoid doing so. But if you do need to optimise, make sure you know how to do it well. Understand when you do need to optimise code, but prefer to write efficient *and good* code in the first place.

Next time

We'll look at good techniques for optimising code.

Pete Goodliffe

Example 2

```
switch (table_selector) {
  case table0:
    /* convertTable0(data);
    break; */
  case table1:
    convertTable1(data);
    break;
  ... ...
}
```

Why Comment Out Code?

The obvious answer – to stop it from executing – begs the further questions: Why should the code not execute? Did it ever execute? Will it ever execute? There are several possibilities:

1 The commented-out code has been superceded by something better, but the author of the new code wants the old code to stick around for a while, perhaps as a reference, perhaps out of historical interest, or perhaps because the new code might not turn out to be better after all.
2 Commenting the code out appears to fix a bug, but no-one understands why – hence the old code is left in comments.
3 The code was commented out during an experiment, maybe an attempt to reproduce a bug, and should never actually have been checked in in such a state. In other words, a bug has been introduced: in fact the code should still be executed.
4 The source file belongs to a third party library which has required modifications to work in-house. The lines which have been commented out represent the source code in its original form.
5 The commented-out code produces irritatingly verbose debug diagnostics which needed silencing.
6 The commented-out code represents work in progess which the author has sensibly checked in to the source repository for safe keeping.

These are all reasons for turning code into comments, some more reasonable than others. Categorising our examples: Example 0 appears to be silenced debug, Example 1 work in progress, and Example 2 a hacked-up experiment – though we really cannot be sure. Example 2 might equally well remove the symptoms (if not the cause) of a bug and Example 1 might represent a superceded method.

The important point is that unless the author of the commented-out code explains what's intended, it is impossible for future readers to accurately guess. How, then, should the author explain? With a comment!

```
for (Surfaces::iterator sf = surfaces.begin();
     sf != surfaces.end();
     ++sf) {
  // std::cout << "Drawing: " << *sf << "\n";
  // Work in progress.
  // Surface stream output not yet
  // implemented.
  // Tom Guest, 12-Dec-2003.
  sf->draw();
}
```

Depending on circumstance, the comment might read:

```
// Do not inflict verbose debug output
// on everyone.
// Tom Guest, 12-Dec-2003.
```

or even:

```
// Commenting out the above line of
// code appears to cure the random
// crashes reported as PR666. I am
// not yet sure why, but am leaving
// the code commented out while I
// investigate.
// Tom Guest, 12-Dec-2003.
```

I have included an explicit name and date in the comment – redundant data, strictly speaking, since they duplicate information held in the source management system – because I do not intend the code to persist in its current state. The comment has a sell-by date. Anyone reading it will immediately understand what's going on and who to hassle if they don't like it.

Why Commenting Out Code is a Bad Idea

Commented-out code – like any other comment – ages badly. It needs maintaining if it is to remain in a state where it can be uncommented and compiled, should the need arise, and of course it won't be maintained in this way since the need is unlikely to arise. There is more than enough live code to maintain without devoting attention to half-dead code in comments.

Once code has been commented out it becomes hard to remove: someone at some point obviously thought the code worth leaving in, so future programmers working on the file honour that decision, although they may well consider the code smells a bit off – code which someone once found problems with, attempted to cure, never really got to the bottom of, and left for someone else to sort out, or, more than likely, ignore.

What To Do Instead

The source management system is the proper home for old versions of source code. If, for some reason, it really seems necessary to explicitly note that an alternative version of the current code once existed, then this can be indicated using indirection.

```
// Previously, surfaces were stored in a
// vector, not a list.
// A list is now used to support efficient
// re-ordering.
// Refer to version 1.22 of this file for
// the vector implementation.
```

There is a direct parallel with the change history of a document: at times it may be very useful to review who changed what, when, or to examine side-by-side differences with the previous version of the document, but this is achieved by accessing the document's revision history, not by leaving obsolete wording lying around in strike-through style. The up-to-date version of the document should be up-to-date.

Ideally, then, the dead code can be cut away. To achieve this ideal requires proper use of the source management system. Check-in comments should be of the same standard as any other project documentation – they need to be clear, accurate, and must include relevant cross-references (to bug report numbers, for example). An iterative approach to check-ins works well: i.e. take the code in steps towards its new form, checking in after each step is complete.

We may not be dealing with dead code, though. Perhaps the code represents work in progress – functions which are being written and which do not yet work, but which nonetheless belong in the source code repository. In this case, the code needs to indicate why it has been commented out, as already mentioned. An alternative would be to store the developing code on a branch until it matures.

Similarly, if the commented-out code represents the original content of a third-party file, then an in-line explanation is required to make this evident to anyone inspecting that file. Even in this situation, I would argue the original content should be cut and the source control system used to manage the differences.

Debug output often gets commented out because it degrades performance or fills up screens. In this case, either the debug code really was a one-off, and should be deleted after use, or it should be carefully integrated into the trace system (the trace system being the module which provides the facility to efficiently filter debug output at run time).

Conclusions

What would happen if you were to cut commented-out code from your source tree? My guess is that you would have significantly less code to maintain, that much of the remaining code would be cleaner (and therefore easier to maintain), that old versions of code would remain accessible, and that functionality and efficiency would be unaltered.

Thomas Guest

Reviews

Bookcase

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

Francis Glassborow writes:

Thanks and Welcome

Those that are sharp of eye will notice the first change to the head of this column for a very long time. Indeed I cannot remember a time when Michael Minihane was not quietly getting on with managing a great deal of the routine that brings this column into existence. I am sure you will all wish to join me in thanking him for the way he has reliably delivered reviews to me for final organisation and processing.

Now we have a change and Christopher Hill (not to be confused with Chris Hills) has taken on the job. I hope you will all make his life easy by volunteering to review books (the procedure has been cleaned up a bit) and keeping a steady flow of books from my office staircase and a regular supply of reviews for this column.

Help Wanted

I will shortly be settling down to write my second book. This one will be an introduction to C++ for people who already know the basics of programming. It will address the problem of programming from a modern C++ perspective (i.e. no contamination from either C or early 90s C++). However it will use the library I developed for my first book which supports simple graphics, raw keyboard reads and a one-button mouse.

I am looking for someone who is familiar with using C++ on Linux to check that all the code will run correctly using the Linux implementation of my library. Any volunteers?

Later I will be looking for test subjects but not yet as I need to get a substantial amount of the book in first draft first.

Spelling

I know my spelling can be pretty shaky at times but since when has 'modelling' been spelt 'modeling'? If my spelling checkers are to be believed this is yet another perversion from the other side of the Atlantic. I know that the English rules often have exceptions but they largely make sense. What rule does the US have that allows me to correctly choose 'sitting' as opposed to siting?

Finally

What is missing from this column that makes it a first ever?

Francis

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

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

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

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

C & C++

Borland C++Builder 6 Developer's Guide by Jarrod Hollingworth et al (0-672-32480-6) SAMS, 800pp @ £43-99 (1.36)
reviewed by R.D. Hughes

This is a vast tome at nearly 1100 pages, which together with the range of topics covered, helps to justify the fairly high price. The authors have managed, however, to reduce the load from the previous edition of the book for C++Builder 5. This was a similar length, but had almost as much additional text on its accompanying CD. Just in case any important topics were cut, the full texts of both this volume and the previous edition are included on the CD accompanying this book.

This guide sets out to cover a broad range of topics in sufficient detail to get you started. For example, it covers: the basic Builder IDE, the VCL GUI library, COM, XML, database connectivity, a range of graphics and multimedia technologies etc. It is very successful in providing a reasonable level of information to get users started, both with potentially unfamiliar technologies, and with using these within Builder.

Although I have not used this version of the book extensively as yet, I have used previous versions and have never found any significant errors. All in all, this book is recommended to most Builder users, whether highly experienced or newcomers. The only note of caution is that if you require an in depth discussion of a particular technology with respect to Builder, this book probably won't serve all your needs, but nonetheless, should get you started.

Visual C++.NET Bible by Tom Archer and Andrew Whitechapel (0-7645-4837-9) Wiley, 1200pp @ £37-50 (1.33)
reviewed by Jon Steven White

My last experience of author Tom Archer was his "Inside C#" book. I liked that one, and so I embarked upon the Visual C++.NET Bible with high expectations. The first thing to cover in this review is who this book is aimed at, as it seems that the title of this book misguides some people. Whilst reviewing it, a number of friends and colleagues picked it up and expected it to be a Managed C++ book. This is not the case, hence the lack of the word "managed" in the title. Managed C++ does make a brief appearance, but this book is primarily a comprehensive guide to Windows application development using Visual C++.NET, from beginner to advanced levels.

Not only is the physical size of this book huge (it weighs in at a hefty 1214 pages), I found it to contain an equally broad content. The first half of the book is devoted to all aspects of MFC programming, and I was very impressed with the level of detail the author goes into. The content easily matches and probably surpasses most of the best MFC books that are currently on offer. Then follows Data I/O, which is dealt with well with some nice chapters on ODBC, ADO, DAO and file I/O.

I really liked the next set of chapters on COM and ATL, which represent a good quarter of the publication. I found them to be clear and well written, and got more from them than I have managed to consume via multiple other ATL and COM specific books. I would recommend this book on its COM and ATL content alone. Finally, the book draws to a close with an introduction to Managed C++ and Windows Forms.

I feel that this book is certainly a worthwhile purchase, especially for developers moving their MFC applications into the Visual C++.NET environment. At £37.50 I think it is really good value as this book is almost guaranteed to help you out sooner or later.

C++ Gotchas by Stephen C. Dewhurst (0-321-12518-5) Addison-Wesley, 384pp @ £34-99 (1.29)
reviewed by John Mullins (second review)

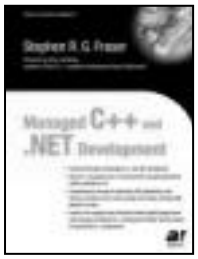
Steve Dewhurst was one of C++'s earliest users and has been a trainer and consultant for many years. This book represents a collection of problems and their solutions that he has encountered in that

time. The book is divided into nine chapters starting with Basics and moving through such things as syntax, resource management and the preprocessor through to class and hierarchy design.

Some of the Gotchas are not unique to C++ (e.g. Gotcha #3 – Global Variables), but crop up often enough to warrant inclusion. Much of what Dewhurst says cannot really be argued against but there are also items that many would consider controversial (for instance, Gotcha #18 discusses just where you should place `const`). There are also occasional items that look out of place; certainly Gotcha #12 (Adolescent Behaviour) doesn't seem to belong in a book about C++.

Much of this book is about communication, about what source code says to its reader (this to some extent probably explains the inclusion of Gotcha #9 – Using Bad Language). Dewhurst is forever emphasizing that source code should be written for the reader and not the compiler, as it will probably be maintained and updated many times. Much is made of idiomatic use of the language, use of the common form of expressions help to clearly express the intent of the author. The same can be said of the use of the standard library. Dewhurst also touches on design patterns, explaining how their use helps to document tried and trusted techniques. Other than a brief description of the mechanics of a particular pattern though, the author leaves detailed discussion to more specialized texts.

The book is aimed at neither experts nor beginners but at working C++ programmers, pretty much everybody can learn something from it. Of course this is not the first book of its kind and having a full collection of Scott Meyers' books on my bookshelf I was a little concerned there may have been very little new here, however I was pleasantly surprised to find only about 20% of the items overlapped and besides the approach of the two authors is very different. While having misgivings about one or two items this has been a welcome addition to my library, Recommended.



Managed C++ and .NET Development by Stephen Fraser (1-59059-033-3) Apress, 951pp @ £43-00 (1.40)
reviewed by Jon Steven White

The primary audience for this book is the C++ programmer who wants to write .NET programs. Stephen Fraser covers a comprehensive range of Managed C++ topics and successfully demonstrates to the reader that Managed C++ is just as important as the traditional .NET languages. Presently, there is a rather limited range of Managed C++ books available, and as an experienced C++ programmer this one is by far the best I've read so far.

The structure of this book is a refreshing alternative to that of most .NET books. The author begins with the usual framework overview followed by a selection of chapters covering the fundamentals and basics of the language, but leaves Windows Forms and Visual Studio .NET development until midway into the book. Following this, there is an excellent chapter covering Graphics with GDI+ and a handful of chapters giving the reader an adequate introduction to ADO.NET, XML, Web Applications (ASP.NET) and Web Services. Some may feel that these latter topics are not detailed enough, but these topics are much too broad to be covered in a book which aims to cover general Managed C++ development. The book gives an excellent introduction of each, which can then be pursued further in a more specialized publication. After covering Multithreaded Programming to a level that should keep C++ programmers happy, the book concludes nicely with a detailed chapter on .NET assemblies.

Overall I found this book to be well structured, clear and accurate with solid code examples throughout. I would highly recommend it as a first Managed C++ book purchase, and although it carries a price tag of around £40 I feel it's worth it. I would have liked the book to cover the basics of interoperability with legacy code and COM components, but to be fair the author does state on the rear of the book that it covers only new .NET program development. Developers looking specifically for a book that will help them to migrate legacy code and components with Managed C++ should look elsewhere.

.NET

.NET and COM The Complete Interoperability Guide by Adam Nathan (0-672-32170-X) SAMS, 1579pp @ £43-99 (1.36)
reviewed by Max Palmer

At nearly 1500 pages this is quite an intimidating book, as is the subject that it covers. The author was part of the interop test team at Microsoft and as such has an excellent grasp of the inner workings of COM, .NET and how both technologies can be made to work together. Fortunately, unlike some developers who write, he is able to explain an otherwise difficult and complex subject area, with both clarity and authority.

The focus of the book is on COM and .NET and how the two technologies can be made to work together. The book is split into eight parts, covering topics such as using COM components in .NET applications, .NET components in COM applications, designing .NET components for COM clients and PInvoke. This is useful, because it allows a developer to dip into the part of the book that is most relevant to their particular project's requirements and quickly gain an understanding of what is possible and some of the design issues they should be aware of. It is also not quite as overwhelming as having to read the entire book from cover to cover.

The book contains a large amount of source code, which is used with good effect to illustrate particular issues, for example, marshalling certain types of data from COM to .NET. These examples are written in a number of different languages, including C#, VB 6 and .NET, C++ and idl, as is necessary to cover different scenarios that the interop team (and developers) have had (and will have) to deal with. Indeed, the advantages and drawbacks of different languages form one of the more revealing aspects of the book, highlighting issues for the unwary such as differences in the default threading models of the .NET languages and how these affect interaction with COM components.

In what is possibly the only drawback of the book, some chapters contain numerous consecutive examples that can, at times, be quite difficult to follow. This is not the fault of the author, however, rather a reflection of the sheer variety of (data) types that need to be discussed. Otherwise, the book is well written, comprehensive and clearly explained. Developers will also find the information contained in the detailed appendices (some 300 pages) very useful, such as a comprehensive list of PInvoke signatures. Recommended.

.NET Multithreading by Alan L.Dennis (1-930110-54-5) Manning, 328pp @ £31-50 (1.11)
reviewed by Max Palmer

At only 328 pages, ".NET Multithreading" is a fairly short book which aims to teach an 'intermediate level' developer the basics of .NET multithreading. Like most .NET books, it takes a somewhat language-neutral approach, with examples throughout the text alternating between C# and VB .NET, while the final chapter discusses how threading applies to J#. It assumes no prior knowledge/experience of multithreaded development and only briefly discusses multithreaded programming using earlier Microsoft development tools (Visual C++), although there is a chapter devoted to COM interop.

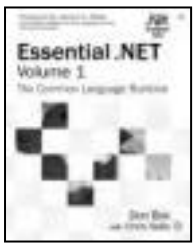
Given the above, ".NET Multithreading" should be relatively accessible, in assuming no prior knowledge, and also readable, given its length. The subject matter should be of interest to most developers, as the ability to write reliable multithreaded code is a very useful skill to have.

So is the book any good? Unfortunately, the best answer I can give is 'sort of'. Certainly, it covers most of the topics I expected to see - deadlocks, race conditions, locking mechanisms, etc. - as well as many others that were new to me - thread pools, CPU affinity and the .NET threading namespace in general. Also, topics are introduced which build upon one another as the book progresses without clouding the earlier discussions with caveats. Reading it also made me more receptive to using threads in my own .NET projects, because Microsoft seem to have made the use of threads much more accessible.

However, the book is inconsistent. For instance, some chapters contain examples with code that lack a through explanation. More worryingly, by the time I reached the end of the

book I had noted a number of mistakes and occurrences of poor formatting, certainly more than I would have expected to find in a book of its short length. I was also surprised to find some relatively basic programming concepts, such as bitfields, being discussed in some depth in an 'intermediate level' book.

That said, the author explains some difficult concepts reasonably well and the book is certainly interesting to read. It is just a shame that its quality and content are so variable.



Essential .NET Volume 1: The Common Language Runtime by Don Box & Chris Sells (0-201-73411-7) Addison-Wesley, 464pp @ £37-99 (1.32)
reviewed by Huw Lloyd

This is a guide to the CLR, the core technology underpinning .NET. Readers should take heed of the preface by seeking initial .NET exposure elsewhere and to be prepared to reread some chapters. Familiarity with C#, COM, design patterns and CIL/Metadata will assist the programmer in tackling this concise yet challenging book.

Of the '.NET Development Series' books I have looked at, this is the densest. There are a few places where I felt more elaboration or diagrams may reduce the difficulty for readers. However, for a subject previously deemed too big for single book a distilled content of 400 succinct pages is a remarkable achievement.

A language neutral description of the CLR is provided, relying mostly on C# for examples. This permits more detail and scope coverage than other language-specific CLR related books, such as 'Inside Microsoft .NET IL Assembler', which is, incidentally, a good complement to this book.

Each chapter focuses upon incremental building blocks of CLR fundamentals to deliver a rounded view of the CLR. The authors provide valuable insights regarding meaning and typical usage of key classes and good judgement regarding details thought to be misleading.

Although the publication date is relatively early, November 2002 for version 1 of the CLR, I found the book consistent with .Net 2003; care is taken to delimit descriptions of implementation details that may change in future releases.

In summary, this is a well thought out guide that conveys clarity to an illusive yet core facet of .Net programming technology. Recommended.

Java

Java 3D API Jump Start by Aaron Walsh & Doug Gehringer (0-13-034076-6) Prentice Hall, 245pp @ £27-99 (1.25)
reviewed by Alan Barclay

This book encourages the reader to discover how the Java 3D API unleashes a new generation of 3D programs for the desktop and the Web. Aimed at the Web professional this book is an introductory text and not an extensive reference manual, although the Java 3D API documentation itself assists in this area.

After a slightly confusing and protracted beginning the book settles down into a relatively straightforward rambling description of the main components of the Java 3D API. It provides the reader with useful knowledge but perhaps leaves them facing extensive trial and error in order to achieve the 3D scenes that they desire to create. At times some extra detail would have been useful but this would have added to the workload of learning this quite extensive subject area.

With a reasonable sprinkling of code snippets and illustrations the book attempts to show the major capabilities and drawbacks of the API but unfortunately I was sometimes left unconvinced by the examples.

There are some useful embedded Notes and Tips sections within the main text and a link to download the author's very nice Java 3D Explorer demonstration application. For best results the reader would benefit from some further 3D education, perhaps in the form of an OpenGL text, to complete the picture. Recommended with reservations

Java 3D Programming by Daniel Selman (1-930110-35-9) Manning, 376pp @ $44.99
reviewed by Alan Barclay

This book claims that novice programmers will gain fast entry into Java 3D development and that experienced Java 3D developers will benefit from the state-of-the-art techniques described within. While I certainly found this to be a comprehensive text about Java 3D development I think that a novice programmer would find the road quite hard going and that experienced developers will still have some questions left unanswered.

All of the major portions of the Java 3D API are covered in a reasonable order and depth along with numerous figures, illustrations, tips about design issues and likely pitfalls. Unfortunately there are no colour illustrations in this book, which is slightly disappointing considering the nature of the topic and its relatively high cost.

There are also a good number of code snippets to help show the relevant points but these often seem to stop just short of making complete sense. It occurs to me that Java 3D development is sufficiently complicated that it can only be well comprehended by examining the source code of full examples and reading much of the Java 3D API documentation itself. Thankfully the author has provided a wealth of good example applications (complete with full source code) on his website and I would strongly recommend that these be examined in conjunction with reading the book.

The sections on 'integration with Swing applications' and 'key navigation behaviour' were particularly helpful for comprehending how a complete Java 3D application (or even a game) might fit together. Almost certainly this whole subject is best understood through experimentation at the computer while working through the various topics of the book. Recommended

Comparison

A comparison between Java 3D API Jump-Start (J3D-JS) and Java 3D Programming (J3D-P) ...

J3D-JS is an easier read and is presented slightly better (in general as well as having colour illustrations) than J3D-P. However the volume and depth of material covered in J3D-JS is not as great as in J3D-P and therefore the reader is likely to be left with many unanswered questions although the reader may still be left with unanswered questions after reading J3D-P too. After working with Java 3D for the past two months I can say that I am very impressed with the results that I have achieved but your mileage will vary depending on the problem you are trying to solve. Without these books I doubt that I would have gotten very far with Java 3D - they are both quite useful and continue to be referred to.

Java 2 Primer Plus by Steven Haines & Steve Potts (0-672-32415-6) SAMS, 800pp @ £32-99 (1.36)
reviewed by Robert Pauer

The intended audience is novice programmers upwards. Before I read this book I knew almost nothing about Java and I have reviewed it from that point of view. It has 5 sections covering basic language, OO concepts, GUI programs, Advanced topics and Web technologies.

It covers the basic language features in a workmanlike fashion but occasionally gets too verbose. For example it is not really necessary to list every single method for the String class taking up 3 pages of text. Lists of class methods can easily be put into an Appendix.

For a programmer from a traditional background it would have been useful to have more explanation of the topics that are different from normal programming. For example: Interfaces and events.

While discussing the collection classes the use of the "import" command mysteriously creeps into the code examples. There is no commentary and no mention of it in the index either (maybe it was generated from the text). We are left to guess its meaning. This is not an isolated instance as earlier the same thing happens to the comment syntax – it just appears in some code. And later I am left wondering what the "package" command does.

The book does contain many complete examples of how to tackle various topics and for this alone it may prove beneficial for some readers.

The two authors have contrasting writing styles and seem to have written different

chapters. I have never noticed this before in works by more than one author but in this case the combined effort does not quite knit together as well as it should.

There is a website reference from where you can download the code examples but there is no list of errata (I noticed quite a few). It would also have been helpful to have more complete instructions on installing and using the SDK, which you must download from the Sun website if you want to try anything out.

I wanted to like this book but it tries to offer something for everyone and that is its main downfall. It is not quite good enough for the complete novice and is probably not sufficiently focused for the proficient Java programmer.

Most books have some form of misprint but one particularly amusing malapropism was: "...the three tenants of OO programming are..."

Java in 60 Minutes a Day by Richard R Raposa (0-471-42314-9) Wiley, 800pp @ £34-95 (1.43)
reviewed by Paul F. Johnson

Yes, it is one of those large language in a small period of time books. Fortunately, this is one of the better ones, but not by much. It's fine for beginners and those who only ever really intend to dabble.

The book covers just about every aspect of Java, but not in great enough detail and definitely not as clearly as it should. For instance, it sets a task without having covered the methods required to understand it fully – and that's as early as Chapter 2, which spoiled an otherwise fine end of chapter assignment.

It also doesn't tell you how to run the compiled apps. I was caught out by that by trying to run an early example, but included the .class extension! It also assumes the user is working under Windows (though it does mention Unix/Linux being able to use it – it would have been nice if the author had said which Java to get and how to install it on a non-Windows platform).

As a memory jogger, it has it is uses. As a replacement for a decent course at your local college on Java, it is not up to the job.

I wasn't overly hopeful to start with and wasn't disappointed.

Java Oracle Database Development by David J. Gallardo (0-13-046218-7) Prentice Hall, 420pp @ £39.99 (1.25)
reviewed by James Gordon

Being an Oracle developer and heavily into Java I loved this book.

It assumes nothing and starts out easily, explaining the tools, database and a simple guide to tables and data types.

It even gives an overview of SQL and PL/SQL just in case.

After that firm foundation, just in case you needed it, it moves onto object-relational and object features and an introduction to JDBC.

The progression in then to advanced JDBC and J2EE persistence, followed by the use of Enterprise Java Beans and object-relational mapping s to Java Data Objects.

All in all a very good book. Filled with simple to understand examples and the output they generate, which is always a good thing.

Java J2SE 1.4, Core Platform Update by James Hart (1-861007-27-2) WROX, 250pp @ £25-99 (1-35)
reviewed by Rodrigo Barnes

This book tries to introduce the breadth of changes between versions 1.3 and 1.4 – with full chapters on the New I/O, XML processing and cryptography. Shorter compound chapters work through the GUI and utility changes. The chapter on language changes (the addition of assertions) is preceded confusingly with one on changes a lot of us would have liked to have seen (type-safe enumerations and generics) but have not made it into this release.

The exposition is clear and some background is given to all the topics that are useful if the inclusion of a technology in 1.4 will be your first real introduction to that technology. This is especially true with the XML and cryptography sections. The section on I/O, while sufficiently detailed may only interest those who have a specialist need in the area. The retention of a self-contained parallel package of old I/O classes and the enormous body of legacy code probably means that the new advantages will not be exploited.

Code examples are included throughout - for the most part interwoven with the text so you have to download the examples from Wrox to see them in full.

As this book covers the Standard Edition of the Java platform – there is little here on the many changes happening in the wider platform, hence the focus on core technologies that are essential for networked applications. No detailed material on Sun's emerging Web Services platform though for example the XML additions are put in the context of those developments.

Working in commercial organizations, I would like to have seen more on the very real issues of deployment, and although there is much material on exercising API features, it does not attempt to address improving programming style and patterns.

While not providing as much material and cross-referencing as a well-wrought website might have provided, this book is a good narrative to read yourself into the major changes 1.4 brings to J2SE. It's not a reference guide, but there are plenty of those around.

The Sun Certified Java developer Exam with J2SE 1.4 by Mehran Habibi et al. (1-59059-030-9) Apress, 350pp @ £35-50 (1.41)
reviewed by James Gordon

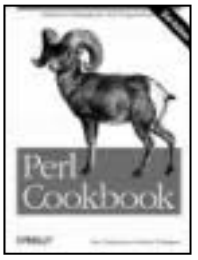
An absolutely cracking book, I couldn't put it down. It was so easy to read, nicely laid out with diagrams, code snippets and program output.

It basically goes through a sample project, a DVD database, and explains what is needed to get the project through to being acceptable for submission for the exam. There is a lot of helpful insight into what is required to pass the exam like whether to use RMI or sockets and the documentation that needs to be packaged with the submission.

It goes through analysis, coding conventions and Javadoc before getting to coding. It then looks at threading, RMI, sockets and the GUI explaining what the examiners would be looking for.

I am looking into doing the first exam, but when I need to pass the second exam I will definitely be getting this book down off of the shelf. This is the first techie book in a while that I have really enjoyed reading.

Other Languages



Perl Cookbook 2ed by Tom Christiansen & Nathan Torkington (0-596-00313-7) O'Reilly, 964pp @ £35-50 (1.41)
reviewed by Paul F. Johnson

If you don't have the first edition, get the second. If you have the first edition – get the second, it really is that good.

The book has expanded on the original with more examples and more pitfalls as well as new chapter on XML and updates on the mod_perl (for the Apache http server) and handling unicode.

I had no problems with compiling and running the code and the explanations of the code and where it can go wrong left me in no doubt that the authors know what they are talking about. Highly Recommended

Windows Programming Made Easy by Glenn Maughan & Raphael Simon (0-13-028977-9) Prentice Hall, 450pp @ $44.99 (no UK)
reviewed by Paul F. Johnson

This is not an MFC book, nor is it one of those books that you pick up and think "Hmmm, another 'Made Easy' book which is anything but". This book actually does what it sets out to do.

Using the Windows Eiffel Library, the book not only documents extremely well the library, but gives plenty of well thought out

and well documented code examples. It really does take the user by the hand and leads them up the tricky path that is programming for Windows.

The CD, which comes with the book, is well thought out and helps no end in learning about this library.

There are two definite advantages to this book.

1. The book assumes you know how to switch on the machine and install from a CD.
2. That you may have a smattering of programming knowledge, but not much.

Based on these two premises, the authors have the basis for a damned fine book – they assume nothing.

Unfortunately though, that does mean that if you do know how to program, then probably this book will seem possibly demeaning.

Now, you may be wondering why with all the good I've said, it's not gone above a "recommended" rating. Simple answer is that while things are well documented, I could not get quite a few of the later chapters' code examples to work on my freshly installed XP box at work – which really drags things down. The attention to detail was marred by the code not working properly.

Python in a Nutshell by Alex Martelli (0-596-00188-6) O'Reilly, 654pp @ £24-95 (1.40)
reviewed by Daire Stockdale

The cover blurb of this excellent book advertises it as 'A Desktop Quick Reference'. I would say that this book is much more then that: it is an informative and interesting treatise on Python, written by someone who clearly has extensive and thorough knowledge of the language. The writing style is extremely terse and factual, which can make the book very heavy reading at times, but I prefer this to the colloquial too often adopted by many computing books. I was very impressed by, and more then a little envious of, Alex Martelli's knowledge and understanding of Python.

The book aims to be a reference work on Python, and so starts at the beginning, explaining installation and theory, the language and syntax. For those already comfortable with Python, these sections might be slightly dull. The book covers 2.1 and 2.2, commenting differences between the versions, and even 2.3 where the specifications are known to the author. The book then deals with core subjects such as object-orientation, exceptions, modules etc, and manages to cover such broad subjects very well.

Then the author covers specific areas of Python usage and libraries, such as threads, testing and debugging, sockets, CGI scripting, HTML and XML parsing, embedding, extending and distributing Python applications. Lacking professional experience in many of these areas, I can only say that the author's treatment of them appears to be of the same high calibre as the rest of the book, and I doubt many who read this book will disagree. Code samples and their output are given where appropriate, each sample being neat and well structured, advancing the readers knowledge. Occasionally the author highlights 'gotchas' particular to Python, or advises a particular coding practise, and these are so sensible and insightful that I was reminded again and again of Stroustrup's 'The C++ Programming Language'.

I whole-heartedly recommend this book to all Python programmers. It is a very concise and informative book, and its small size belies its information content. I would also recommend and suggest this book to any experienced programmers wishing to begin programming in Python, who would enjoy a straight-to-the-point manual on the Python language.

Design

Packaged Composite Applications by Dan Woods (0-596-00552-0) O'Reilly, 187pp @ £20-95 (1.43)
reviewed by Richard Stones

The title of the book is probably not going to mean much to many people, but the book is about how you can use existing applications to underpin an 'Enterprise Service Bus' and then build new applications on top. In short: add services (possibly, but not necessarily web services) to your existing applications, then by choreographing these services together build new applications that span several existing packaged application areas. This concept will probably be familiar to people who have worked with Enterprise Application Integration (EAI) tools, such as SeeBeyond, TIBCO and others, though PCAs are more than just integrating existing applications. The aim is to do more with your existing applications, because in today's environment you probably can't afford to replace them.

The book is different from the usual O'Reilly book, it's quite short, and aimed at executive/ senior IT manager type level. There is very little technical detail or hard facts in the book.

The book makes a good case for explaining why the PCA concept is important, looks, a little briefly, at (1.43)alternatives and explains why the author thinks PCAs are better. There is most of a chapter on the case against PCAs, though for my money he missed one important snag – doing this stuff well is difficult, and there are going to be some expensive mistakes made in some enterprises.

Having explained the background and argued the case, the book moves on to work though the vision, creating the service bus that underpins PCAs and what PCAs can actually do for you, and where they are best used. This is the point at which I expected the book to give me a few more tangible ideas and rules of thumb, but unfortunately it sticks to very high level concepts, and I found myself desperate for some more solid detail.

Overall an unusual book – it makes a good job of a high level explanation of PCAs, and why they are important and probably a coming IT fashion, but for me was a bit too light on the practicalities, so just misses a 'recommended'. However if you are looking at EAI tools, and implementing services on top of existing applications, but wanting a high level view without technical detail, this book is certainly worth considering.

Data Access Patterns by Clifton Nock (0-13-140157-2) Addison-Wesley, 400pp @ £39-99 (1.25)
reviewed by Huw Lloyd

If you have ever puzzled over how to implement persistence code in a well-factored reusable configurable manner this is a book for you.

Each pattern teaches us about the persistence problems that manifest themselves between object-oriented applications and relational databases. Each pattern tends to deal with one prevalent problem, along with an appropriate class-based solution to deal with it.

Unlike the original 'Design Patterns', where patterns focused on domain-generic software problems, each pattern is framed by a particular domain problem. Many of the patterns are described as specialisations of the original patterns. Indeed, few of the patterns are new class relationships; it is their application and appropriateness to particular problems that gives them their name and value.

The elegance of the patterns is manifest by their clarity and the simplicity with which they are integrated to resolve problems of persistence. I found the text to read smoothly with a consistent level of difficulty – interesting but not demanding.

The examples are provided in Java; a little imagination will be required for C++ purists to reinterpret some of the author's style, such as a tendency to use class Object to pass arguments. The example domain objects are overly basic – a few pages are wasted listing get/set operations for all class members – but their simplicity may reduce the clutter of the patterns' presentation.

You will need to know about OO programming, design patterns, the basics of using relational databases and SQL to benefit most from this book. Recommended.

Patterns of Enterprise Application Architecture by Martin Fowler (0-321-12742-0) Addison-Wesley, 512pp @ £37-99 (1.32)
reviewed by Garry Lancaster

A quick browse through C Vu and Overload shows that programmers are using some very advanced features of their languages in a vast range of application types. However, if we are honest, many of us spend a large proportion of our working hours building applications based around relational database systems. This may

not be the most fashionable kind of programming, but it is not without its challenges. Although many books detail a particular database system or explain how to get started with SQL, this book is a comparative rarity, being concerned with how to write real (not toy) code in enterprise complexity systems that elegantly bridges the gaps between databases, forms, reports and proper object-oriented coding techniques.

The book takes the pattern perspective, popularised by the seminal Design Patterns book by Gamma et al. (often referred to as the Gang of Four or GOF), where the common features of real code are abstracted into prose, UML diagrams and examples, and named as a pattern. The book's examples are written in either Java or C#, but are straightforward enough that intermediate to advanced C++ programmers should have little problem making sense of it. Most of the advice given in the book translates equally well into C++, although there are some areas that are platform specific.

In terms of scope the book does not cover replicated or distributed systems, opting to give more thorough coverage to the areas of user interface design, domain modelling and interfacing with backend databases. Perhaps the most taxing area addressed is concurrency, but primarily this is looking at the concurrency management systems that database systems tend to provide, such as transactions.

Anyone who has read the GOF work will find the layout of the majority of this book very familiar: the inside front cover summarises the patterns covered, the patterns are grouped into convenient categories, and each pattern is described over the course of several pages, together with copious examples. The inside back cover provides a helpful "cheat sheet" which boils down the mechanics of choosing an appropriate pattern to the answering of a small number of simple questions. The biggest difference between Fowler's book and its spiritual predecessor is that here you will find more discussion of how the patterns fit together to form a complete application. I found this a welcome change of emphasis: this practical slant makes the application of the detailed patterns more straightforward.

My only real criticism is the level that some parts of the book are pitched at. Relatively simple ideas are explained in overlong detail. This complaint has a positive flipside of course: this book is very clear and explanations are never rushed. Those who are veterans of a lot of well-structured database-oriented work, and are therefore already familiar with many of the ideas expressed, may well wish for an abridged version.

There is no abridged version of course, nor have I ever read any other book on this subject that does the job better. It's a solid read, I could not find any technical faults and it has good index and further reading sections. Although there is no "wow!" factor to this book, it would be unrealistic to expect one given the subject matter. If you are in the intended audience, reading and understanding Patterns of Enterprise Application Architecture will almost certainly improve the quality of your work and save development time. Therefore this book is recommended.

Software by Numbers by Mark Denne & Jane Clelaund-Huang (0-13-140728-7) Prentice Hall, 190pp @ £31-99 (1.25) reviewed by James Roberts

This book describes a scheme for ordering the development of items within a particular project to maximise the Return on Investment (ROI).

The premise of the book is that following the dot-com crash, getting investment with projected payback over any period other than the short-term is increasingly difficult. Therefore, steps have to be taken to ensure returns are made quickly on investments made, with a minimum amount of capital tied up unnecessarily in effort that does not provide immediate return.

Within its scope the book worked well. It showed how to identify 'Minimal Marketable Features' (MMFs) which are the smallest units that can give a quantifiable return. The book then showed the arithmetic formulae to calculate a true ROI on development of these features, and discussed ways to improve the ordering of MMF development within the project to maximise ROI. It then showed how this financial scheme could be applied to development methodologies such as RUP and XP (the authors were keen to say that this process is methodology independent).

On the downside, I felt that for many projects the book was over simplistic. It seemed to assume that the cost of a release of software from development to a live environment was directly proportional to the cost of development. Hence there was no discussion of how to factor in variable costing depending on the number of drops to live. Similarly it didn't address issues with variable team-size etc, which project reordering would be expected to throw up.

In short, this is an interesting book with some useful ideas, but probably to be taken with a large pinch of salt.

Methodology & Tools

Beyond Software Architecture by Luke Hohmann (0-201-77594-8) Addison-Wesley, 314pp @ £32-99 (1.21)
reviewed by Pete Goodliffe

Just what we need – another book on software architecture! Surely it has all been said already? Well, no – this is a good book, endorsed by Martin Fowler. If it's anything to go by, the rest of the Addison Wesley "Signature Series" is probably worth a look.

This is not a traditional book on software architecture. As the title implies, it goes beyond the standard descriptions of system design, to investigate the issues that drive successful software architecture.

Hohmann shows how software architecture ties in with the realities of product development, and describes how successful architectures marry technical concerns with the real commercial pressures of the marketplace. He details pragmatic approaches to a number of key issues, including: licensing, portability, branding, and upgrading/deployment.

The tone is authoritative; the author writes insightfully, with a clear depth of knowledge and experience. It is thought provoking, readable, and accessible.

Who should read this book? It has a definite thrust towards software professionals making Real World decisions. It is applicable to engineers designing software systems, and also valuable to those who manage such projects. Highly recommended.

Test-Driven Development By Example by Kent Beck (0-321-14653-0) Addison-Wesley, 176pp @ £22-99 (1.30) reviewed by John Mullins (2ed review)

Kent Beck introduces the reader to a technique of software design driven by tests, the aim being to develop "clean code that works – now".

The first part the book consists of two tutorials that demonstrate of examples of organically growing software by first writing a test to exercise a piece of functionality and then writing the code to make that test pass and then refactoring to remove duplication. The author shows how to move in very small steps when working through particularly sticky areas of code or move in larger steps when confidence permits. Beck uses the tests to inspire confidence in the code being developed, and to continually drive the development forward. Beck also emphasizes that tests should be easy and quick to run (if they are not they unlikely to be run as often as they should) and introduces the test framework JUnit, a tool for automating the running of tests.

The first tutorial is fairly lightweight and consists of developing a multi-currency class in Java. Here we are introduced to the various ways the author uses to quickly get failing tests to pass. There is much emphasis on the red/green bar, which refers to the JUnit framework that uses a red bar to indicate failure and a green bar to indicate success. The author does not like having a 'red bar' for any significant length of time and shows how to quickly get to green (often using any means necessary), i.e. to get to the "code that works" stage, before turning that into "clean code". The second tutorial is slightly more heavyweight and describes the development of the above-mentioned testing framework in Python. Here we gain more insights like how do we start to write tests to test the framework we will be using to write tests and why and how to isolate our tests from one another.

The final part of the book discusses various testing patterns, design patterns, refactoring, and where to go from here. There is a whole list of patterns you can add to your arsenal of tools to keep the bar green. Beck describes the refactorings that are common in TDD and how their use differs slightly from the classic view presented by Martin Fowler in his book on the topic. There is also a discussion of where we have difficulty using Test Driven Development, such as GUI development.

The book is very easy to read, the authors style is very relaxed and he is clearly very

enthusiastic about the story he has to tell. The book packs in an awful lot considering it is just over 200 pages long, if you wish to experiment with TDD everything you need to get started is here. Without doubt, one of the best books I have read this year, Highly Recommended.

Lean Software Development: An Agile Toolkit by Mary Poppendieck & Tom Poppendieck (0-321-15078-3) Addison-Wesley, 203pp @ £30-99 (1.29)
reviewed by Anthony Williams

If you're not convinced that Agile Software Development practices are at least worth investigating further after you've read this book, then you'll never be convinced. It is a well-written guide to the ideas behind Agile thinking, with plenty of references to other sources (the bibliography is 8 pages long); both those showing the benefits of Lean thinking in software and manufacturing, and the "original" sources for various techniques and methodologies. The interested reader therefore has plenty of material for ideas on where to go next, having accepted Agile practices as effective development practices.

The book divides lean thinking into 7 key principles, with 22 "tools" to help you adapt agile practices to your workplace. It also features a "try it out" section at the end of each chapter taking you through some simple steps that demonstrate how the particular techniques discussed can be applied to improve your software development process.

Some of these principles are obvious at first glance – "Eliminate Waste" for example – but this simplicity hides profound insight; in this case, the insight is that much of the "work products" of traditional software development processes are in fact waste, produced purely so the developer can "tick the box" and move onto the next task. Not only that, but the very process can itself generate waste – having analysts produce specs from customer requirements, which designers then turn into a high level design for coders to turn into software is very wasteful, because knowledge is lost at every stage; the very act of writing something down means that the understanding and background knowledge held by the author is lost, either permanently, or until the reader has acquired it for himself.

This book is aimed at project managers and lead developers looking for ways to improve their software development process, but I would recommend it to anyone who is serious about producing quality software. Whilst many Agile practices require management buy-in (and if you can get your manager to read this book, it will probably help), others can be implemented by developers as part of almost any process.

Highly Recommended.

Essential CVS by Jennifer Vesperman (0-596-00459-1) O'Reilly, 336pp @ £28-50 (1.40)
reviewed by Nicola Musatti

This book is a practical guide to CVS which addresses both users and administrators. It is composed of an introduction, a user guide, an administrator guide and a reference section. The introduction is the section I liked the least; in my opinion there is too wide a gap between the general description of what version control is and the subsequent quick tour of the features of CVS. Maybe a few pages could have been devoted to a description of the common tasks of version control management. These topics are indeed covered in the subsequent sections.

The user guide and the administrator guide are both fairly complete and combine a description of the functions available with advice on how to make the most of them. The reference section not only covers command syntax but also the file formats and environment variables. I did not have the opportunity to verify it thoroughly, but it seems to be accurate.

The author appears to know her topic very well and manages to present it clearly, even though I found her style a little awkward here and there and many concepts are repeated almost verbatim in different places. However these defects do not detract from the overall quality of this book.

It should be noted that this book only covers the use of command line CVS on Unix/Linux systems; graphical front-ends and solutions for other platforms are briefly covered in an appendix. Nonetheless I am convinced that "Essential CVS" is useful even to those that work on different systems.

Lastly, it may be just a matter of taste, but I find the 300 page format very convenient: I do most of my technical reading while travelling on public transport and I appreciate books that I can carry with me without breaking my back in the process.

This is a book that delivers what it promises to its target audience. If you are a CVS user chances are you will never need another book; if you are an administrator it will still cover most of your needs.

Unix

Automating UNIX and Linux Administration by Kirk Bauer (1-59059-212-3) Apress, 600pp @ £35-50 (1.41)
reviewed by Paul F. Johnson

While automating a system is the dream of all sysadmins, there is such a thing as too much automation. Thankfully, this book recognises it.

It covers just about everything a sysadmin needs to know, using the common Unix/Linux tool-chain available to all as well as the pitfalls and how to ensure your systems stay secure.

It is based around the RedHat flavour of Linux (which is even better for me given my servers at work are all RedHat 9 or Fedora boxes) and demonstrates how to get the best out of the system. The book also covers the sticky subject of if to patch or install from fresh to building an RPM environment.

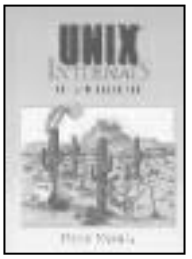
If you're a sysadmin – get this book, you won't be sorry. Highly Recommended.

Unix Internals: A practical approach by Steve Pate (0-201-87721-X) Addison-Wesley, 688pp @ £28-99 (1.72)
reviewed by David Caabeiro

This book should have been titled "SCO Internals", as it discusses different SCO versions (including Openserver R5) in a practical approach, as the title states. Besides that, it's a very good book, which takes you through the shell, processes, memory management, IPC, file services, I/O system, and many other areas typically found in this kind of books.

Again, the problem lies that the author expects you to have a SCO flavour installed for practicing. If that weren't the case, do not expect to practice (in most cases) the examples given in the book, as they rely on utilities found only in SCO, such as crash, adm, etc.

On the positive side, the book contains several screen dumps with captures of program output and code examples, which makes the above requisite dispensable. Recommended with reservations.



Unix Internals: The New Frontiers by Uresh Vahalia (0-13-101908-2) Prentice Hall, 601pp @ $72.00 (new edition due in 2005)
reviewed by David Caabeiro

Albeit being somewhat dated, I think this book is the best available in its category. It discusses in detail most features of current state-of-the-art Unix kernels, including SVR4.2, Mach, 4.4BSD, Solaris 2.x, which makes it a good follow-up to former books such as Bach's Unix Internals.

Each chapter provides descriptions of several topics, with clear diagrams and comparisons among different implementations. Code is provided when discussing implementation issues, but in a brief and to the point style.

At the end of each chapter, you can find a list of references to fathom on the topics discussed. Exercises are also available, to provide food for thought and consolidate the concepts learnt. Unfortunately there is no answer book available.

The author starts with an overview of the evolution of Unix since its inception to the current thicket of different flavours. Then it describes processes related issues in traditional and modern kernels: threads, signals, job control, scheduling, IPC and synchronization in uniprocessor and multiprocessors.

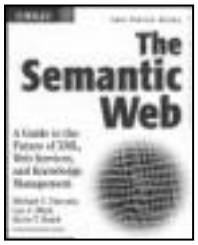
Next chapters introduce the file system interface and the VFS concept, detailing some specific implementations, to introduce you to distributed fs and finally some advanced concepts such as journaling and stackable fs. Then it starts with memory management from the kernel and user point of view, provides a good description of VM and discusses implementations in SVR4, Solaris, 4.4BSD and Mach.

Finally, it describes cache-related issues and its effects. The last two chapters address the IO subsystem, discussing the traditional framework (char and block devices), and STREAMS.

This is not an introductory book. You need some background in operating systems concepts, such as virtual memory, processes, etc. which can be learnt in introductory books for undergraduate courses such as Tanembaum's Modern Operating Systems, Silberchatz, Stallings, etc.

If you are into operating system design, Unix programming or even system administration, get this book you will not regret it. Highly Recommended

Internet



The Semantic Web by Michael C Daconta, Leo J Obrst & Kevin T Smith (0-471-43257-1) Wiley, 281pp @ £24-50 (1.43) reviewed by Alan Lenton

As a programmer I've noticed the term 'Semantic Web' is starting to get bandied around on various technical lists that I subscribe to. When I spotted this book on the list of those available for review it seemed like a good opportunity to fill in the gaps in my mainly self-taught knowledge.

And did it do that? Well yes, after a fashion. The book advertises itself as being for managers and senior developers, but really it is a whiz through the subject for the former, rather than the latter. A manager who reads this book attentively through will have a firm grasp of the jargon, but probably not the technology. A programmer will (probably) have an overall picture, but little to work with.

To be fair, the book does more or less cover its subject, and the authors have impressive qualifications in the field.

For those who haven't come across it before, 'The Semantic Web' is a term coined by Tim Berners-Lee to cover a new vision of the web where the content is machine readable as well as human readable. From a programming point of view you have not merely syntax, but also semantics which allows software to make 'intelligent' decisions.

The technology is, of course, built on XML, but XML is only the foundation. On top of that are the much-hyped Web Services, not to mention RDF, taxonomies and ontologies.

Unfortunately, I can't really recommend this book for a number of reasons. First because the high speed, zoom through the various aspects of the Semantic Web doesn't really justify the investment in reading time. Then there is the problem that a lot of the diagrams look as though they have been drawn by a drunken spider dipped in ink. (See for instance the graphical ontology example on p.183.)

But the real reason why I can't recommend this book is that the authors are boring writers. They are undoubtedly highly qualified, but, sadly, that doesn't mean they are good writers. I really wanted to know more about this subject, but if I hadn't undertaken to review this book I would probably have given up by halfway through it. The prose is really very pedestrian.

And my favourite quote from the book? 'A subject indicator is just a way of indicating subjects.' (p.174) Sorry guys, this one isn't going to make it on to my bookshelf.

Beginning Dynamic Websites by David Sussman (1-86100-792-2) WROX, 545pp @ £28-99 (1.38) reviewed by James Gordon

This book really assumes nothing. It starts with installing Web Matrix from the included CD and then steps through the basics of HTML using ASP.NET.

The first part of the book is an introduction to ASP.NET, Web Matrix and the basics of writing code e.g. variables, collections and error handling.

Section 2 concentrates on getting and storing data in databases, Microsoft databases that is. But no doubt it can be made to work with other databases. The final section is about your website as a whole. It goes into linking pages and files together, reusable code, cookies and session states. It then goes into Web Services and where to go once you have finished the book.

A very neat book; as long as you are using Web Matrix and Microsoft databases. It does show in many instances the code generated by the Web Matrix tool and the full source for the book can be downloaded.

Dynamic HTML The Definitive Reference by Danny Goodman (0-596-00316-1) O'Reilly, 1401pp @ £42.50 (1.41) reviewed by Colin Harkness

I found it hard to fault this book since it does exactly what it says on the tin (well, the back cover). If there is a problem, it is that the book is too big.

This is mainly a reference book. The 200 pages of guidance at the start is not a beginner's tutorial, so if you are new to any of the technologies covered, you should look elsewhere for a gentle introduction. But you will probably want this by your side as well. The guidance on, and widespread indicators of browser compatibility are invaluable and are themselves a good enough reason to buy the book.

The 280 dense pages on HTML/XHTML and 540 pages on the Document Object Model, could easily form their own books and are truly comprehensive. In contrast, the CSS and JavaScript references might not be complete enough for some. The author has another book that covers JavaScript in considerable depth.

Webmaster in a Nutshell 3ed by Stephen Spainhour & Robert Eckstein (0-596-00357-9) O'Reilly, 576pp @ £24-95 (1.40) reviewed by Tony Houghton

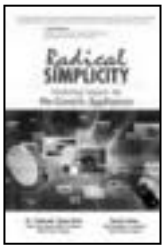
The books in O'Reilly's Nutshell series are neither definitive detailed references nor tutorials. Instead they provide a handy, concise reference for something you've already learnt but can't keep entirely in your memory, and would otherwise need several books for. The Nutshells generally fill this role very well and are highly regarded; this is no exception.

Webmaster in a Nutshell can be considered to have two halves, client-side and server-side. The former covers HTML – with separate chapters usefully dedicated to tables and forms – CSS, XML and Javascript and the latter CGI and Perl, PHP, and Apache – configuration, modules and performance optimisation.

When I first dipped into the book while coding some HTML I wasn't terribly impressed, but having browsed it much more thoroughly for review, I've got a much better impression. It's worth getting a good overview of how the book is laid out before using it as a reference. I felt that if I was only going to have one book on web coding this would be it, and if I had not already read others it would suffice alongside some basic tutorials and code examples from the web. I say coding, because the Nutshell doesn't offer any advice on design or images, or software for site design, editing and management. The latter is understandable, because it would be platform specific. On the plus side it does have useful information about common browser incompatibilities in areas like Javascript and the DOM.

I think on the whole I would give this a Highly Recommended rating.

Other



Radical Simplicity Transforming Computers Into Me-Centric Appliances by Dr. Frederick Hayes-Roth & Daniel Amor (0-13-100291-0) Prentice Hall, 300pp @ £23-99 (1.25) reviewed by James Roberts

This book is about 'me-centric computing', which (according to the cover) relates to the 'next computer revolution'. The general idea is that computers will become more pervasive, and become embedded in more and more appliances as time goes by, which will create new patterns of use for computers.

These appliance-computers will be able to communicate with each other in unexpected and useful ways. This, say the authors will require the embedded software to be more task-oriented and integrated around the user (which is the me-centricity referred to in the title).

To be useable this will require the computers to be far more user-friendly than the current status quo. As an illustration the authors refer (repeatedly) to the difficulty many people have in programming a video recorder. If a watch became similarly complex, would it be useable (and would it require a thick instruction manual to be carried around at all times)?

The book generally seems to be an exercise in blue-sky thinking, peppered through with example of how the future technology might operate – contrasting with how similar situations are handled today.

Although the premise for this book is interesting, in general I was disappointed by the details.

The cover blurb says 'this book shows how', there were few details (XML and Internet I would probably have guessed myself, 'agent' software is rather too generalized a term to help).

Another grouse I had with the book was that I felt that the examples they had were somewhat pedestrian. For example, software agents were to be used to verify the date/location of a meeting. Why do agents not continuously monitor diary compatibilities? The suggested solution seems no better than current diary capabilities (e.g. from MS Exchange).

In summary, this book has some interesting ideas, which it presents convincingly. However, I would have hoped for more details, both in terms of the vision and how it might be implemented.

Oracle PL/SQL 2e Interactive Workbook by Benjamin Rosenzweig & Elena Silvestrova (0-13-047320-0) Prentice Hall PTR, 500pp @ £31-99 (1.25)

Oracle SQL 2e Interactive Workbook by Alice Rischert (0-13-100277-5) Prentice Hall PTR, 500pp @ £31-99 (1.25)
reviewed by Christer Lofving

In my software career I have worked with both Microsoft SQL Server and Oracle databases. I do not want to make any overall comparison of the two systems, but in my opinion one rather non-technical difference is the availability of handy code reference help – the aid you need to write both proper SQL statements and more complex commands like stored procedures, cursors or triggers. I think SQL Server has a clear advantage in this area with its accompanying "SQL Books" in old reliable Windows help file format. When I try to get the same help in an Oracle environment, I feel the need for some external (paper based) reference tool to be more urgent, which is why I was happy to find these books.

Their sub-title is "The independent voice on Oracle", a series of Oracle titles published by Prentice Hall. They both combine the best of computing tutorials with reference manuals at their peak. "Oracle SQL" is more fundamental while "Oracle PL-SQL" is addressing a harder to grasp subject.

But they are so similar in concept, disposition and text that I do not hesitate to cover them as a single entity in this review. The books are about 700-800 pages in length, but seldom in the short history of computer books have so many pages been filled with so much relevant material.

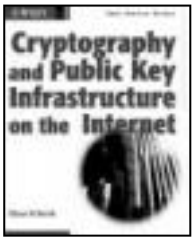
The layout of every chapter is very pedagogic. There are exercises, solutions to exercises and self-review questions at the end. Every single SQL-function/PL-SQL language feature is explained. Generally "unreadable" syntax patterns are avoided - instead small but fully functional code samples are shown. An approach I think most readers appreciate.

Other plus points; includes a well-designed tutorial database, (which the examples are built upon) complete with test data, (downloadable from web site) and a comprehensive index.

The editions I reviewed were Oracle 8 based, but updated for Oracle 9i. That's good, but I think the changes are pretty small on this lower level of Oracle.

These are great references titles but also sophisticated learning tools. That means they are useful both when first learning the subject and a long time after that.

They are not "talk bibles" – they are carefully written by authors expert in their subject, they are learning tools in the words best sense. Rather expensive books, yet highly recommended.



Cryptography and Public Key Infrastructure on the Internet by Klaus Schmeh (0-470-84745-X) Wiley, 472pp @ £34-95 (1.43)
reviewed by James Gordon

This book details a lot of information dealing with why cryptography is important on the Internet and how some styles of cryptography are easily broken allowing others to eavesdrop. It then explains how different encryption styles are implemented, the standards that are out there, including some real world examples of things going right and how thing can sometimes go wrong.

There are chapters on encryption protocols. Lastly there are a few chapters on the politics, people and flops in the cryptography world.

It also gives a list of external references for further reading.

The book contains a lot of simple diagrams that I found helpful with some of the descriptions, though most were very well written descriptions.

If you are into cryptography it seems to be a good book for helping to understand how things are done and the pitfalls that can get into you if you do not know about them.

Technology Acquisition, Buying the Future of Your Business by Allen Eskelin (0-201-73804-X) Addison-Wesley, 256pp @ $29.99 (no UK)
reviewed by Huw Lloyd

This short book describes the author's process for acquiring IT systems in the price range of $500,000 to $10 million. It is portrayed as a linear process; the roles and activities are described in time sequence. The target audience are project managers tasked with acquiring a system, and all other stakeholders.

This text does not explore the strategic ramifications for buying-in technology. There is a brief look at the trade offs between internal development and buying an external solution, but the majority of the book focuses on the key players and activities in the acquisition process. So as a study of major things that could go wrong – such as changing business needs – I think it is inadequate. But as a primer, for a PM wishing to get a good feel for all the activities entailed, I think it performs well.

It is an easy read, without any big surprises or knotty details– although it is occasionally repetitive. Where close attention is required the author has provided some good samples (the 30 page Request For Proposal is good) and templates for key documents necessary for such an undertaking. The author's experience becomes evident in his suggestions to pay particular attention to certain project details. Checklists and tips accompany each chapter.

References are very sparse in the book. I would have appreciated some pointers for more in-depth material that this guide necessarily glosses over.

The Data Warehouse Toolkit 2ed by Ralph Kimball & Margy Ross (0471200247) Wiley, 464pp @ £36-95 (1.49)
reviewed by David Ross

My day job revolves around the building of Data Warehouses and Data Mining environments and I regularly recommend Ralph Kimball's numerous books on the topic to colleagues and clients. This particular book, an update of the 1996 edition, is no exception. This time around there is less emphasis on selling the concept of dimensional modelling (now the accepted way to build warehouses) and more on how to do it.

The books takes an interesting approach: after first discussing the basics of why and how create a dimensional model, it then uses cases studies from various industry sectors to explore the intricacies of the techniques and issues which arise during implementation. The temptation to focus on your own speciality should be avoided since each provides new insights. For instance, the handling of multiple time zones is covered in the section on transportation, while fact-less fact tables are covered in the education chapter. Throughout there is an emphasis on addressing real business issues, with numerous example schemas to support the text.

There are a few chapters relating to concept and benefits of data warehousing, how to implement a warehouse and future directions. While these provide useful context (the first chapter being an excellent introduction to the field), this is principally a book about dimensional modelling and as such I can strongly recommend it.