Contents

Reports & Opinions

Dialogue

Features

Reviews

Copy Dates

C Vu 17.4: July 7th 2005

C Vu 17.5: September 1st 2005

Contact Information:

Editorial: Paul Johnson
77 Station Road, Haydock,
St Helens,
Merseyside, WA11 0JL
cvu@accu.org

Advertising: Thaddaeus Frogley
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): £25

Full (C Vu and Overload): £35

Corporate: £120

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

A great deal has happened since last I wrote an editorial. We've had our annual conference (more of which later), I've finally had my stomach operation, we have a new production editor, a new person taking care of the Student Code Competition and lots of progress on the ACCU website. Things really are moving along!

Other things haven't moved on very much. We've (in the UK) had an election, litigation in the IBM/SCO, Novell/SCO and RedHat/SCO case are going slightly faster than an asthmatic ant carrying some heavy shopping up a hill and books with fundamental mistakes are still being published despite such piffling little things like the 1998 C++ standard coming in.

Coding Styles in Books

Something is eating at me and I'm pretty sure that it's nibbled at you all at some point and that is the plague of fundamental mistakes made in modern texts. I've been recently reading a 2004 book for Maths and Physics in Computer and Video Games programming. Partly out of interest, but mostly as I teach it and despite the book being very well presented and written, it is let down with the likes of code like this:

```
/* purpose: to calculate the
distance between two points
// input:  P1 – an array of
two floats representing point
1
//         P2 – an array of
two floats representing point
2
// output: the distance
between the two points

float distance2D(float *P1,
float *P2)
{
  // calculate the distance
and return it
  return
(float)sqrt(pow(P2[0] –
P1[0]), 2) +
pow(P2[1] – P1[1], 2);
}
```

Okay, there doesn't really seem to be too much wrong there – in fact, there is nothing wrong with it. Not until you consider that the language in use is C++, so the casting method for the float is incorrect and I'm at a loss to see why float is being used over double.

Am I being too fussy over the cast of float or the use of floats over double? Well, no. Casting in C++ in very much different to C and for a modern compiler, there is no benefit of using a float over a double, infact, there are some very good arguments (albeit old ones) for using a double in preference to float.

This continues throughout the book. Sure, it's not really a big issue to those who know what they're talking about and do the mental changes so they are reading what should have been written rather than what has actually been written, but for the newbie, this sort of information is setting them up for quite a fall later on.

Slightly later on in the book, there is a small piece on converting from degrees to radians. It's well explained then the following is given to enable conversions:

```
#define RadToDeg 57.29577951f
#define DegToRad 0.017453293f
```

Had this been a C book, I'd have accepted it. However in C++ these should really be:

```
const double RadToDeg =
    57.29577951;
const double DegToRad =
    0.017453293;
```

These simple errors detract greatly from a very good book. Until you hit the CD.

Why do companies insist on putting CDs in books which detract from the book itself?

The worst example of this comes from a book I reviewed a while back called "Linux Games Programming" where it contained (in the words of the Hitch Hiker's Guide to the Galaxy) "much that was apocryphal if not wildly inaccurate" - material which just wouldn't compile, missing libraries from code examples and code which just didn't belong on the disc!

For this particular book, I was encouraged to see multiple references to OpenGL – which is great for everyone as it means all of the examples can be compiled on a multitude of platforms and I can then give live demonstrations. Um. No. It is OpenGL but the code examples all have lots of Win32 bits surrounding the code and if you have little or no experience of OpenGL (or Win32 code), the examples are about as much use as a chocolate teapot. Do I spend the time hacking things around to get the code to compile on any of my Linux boxes or do I write fresh code to do the same as the example code, become more proficient in OpenGL coding but at the same waste time I don't really have in order to use the demonstration code? Answers on a postcard.

I find it unsatisfactory (to say the least) that in 2005 we are still being subjected to books with this type of mistake and oversight in them. At least in the case of this particular book, the publishers are more than happy to pass on comments to the author for the next edition.

Desert Island Books

It seems that competitions are all the thing at the moment. In the last edition, we had the competition results in Francis' Scribbles for the Desert Island books. It was quite interesting to see what books people would have on their mythical desert island.

I'd like to run 2 similar competitions (with a similar prize).

The first would be simple. Any 5 books which you would send your teenager off to university with (they can be studying any programming course and is not restricted to conventional "computer studies" courses). They are supplied with a Mini Mac, flat screen monitor (solar powered) with gcc 4. That should make it far more interesting!

The second would be more interesting. Any 5 books which should be buried in quicklime.

Why would the second one be more interesting than the first? Interesting question.

We all know the good books (typically the ones our reviewers give Highly Recommended ratings to!) and would be happy to be seen with them. However, as we amass books, it is inevitable that we pick up a real turkey with more howlers than you'd find in a pack of wolves. These are the books that if you were not so embarrassed to have to admit to buying in the first place, you'd have taken it back and demanded not only a refund, but probably a couple of books to make up for the mental stress and anguish it placed you under. Everyone has them.

What was your crime for this to befall you? What was so awful that you are condemned to this? I don't know. It must have been something terrible. To make it worse, you've been supplied with a ZX Spectrum using HiSoft "C" and a crummy black and white TV (only usable on UHF 36) and a tape recorded with 1 C60 and which has to be run off batteries.

Best entries will win a prize. Only genuine books though...

On With The Show...

That's enough of my ramblings. On with the show!

Paul F Johnson

View From the Chair

Ewan Milne <chair@accu.org>

At the time of writing, a fortnight has passed since the conference: just enough time to take stock and reflect on another highly satisfying and enjoyable event. Thanks one last time to everyone who came along and contributed to the success. Speakers, sponsors and delegates, everyone plays a crucial role.

Next year's conference is scheduled for the 19th - 22nd April 2006, almost certainly with a pre-conference day on the 18th. It is also most likely that we will return to the Randolph Hotel.

There has been much debate about the future location of the conference. We recognize that the Randolph is not the perfect venue, but also that in the real world, compromises have to be made. And it must be said that the Randolph does actually score well in the evaluation form feedback. The conclusion is that while we are actively investigating other locations, due to the long lead times involved this task is mainly focused on future events: 2007 onwards. There is one candidate already

on the table which may be a realistic alternative for next year, with a final decision being made in the next month or two. I willreport back next issue.

Also look out for the Call for Participation for the 2006 conference: alarming as it may seem, preparations are getting underway, and this will be appearing in accu-general early in July.

Finally, I'd like to draw your attention to Pete Goodliffe's announcement of the 2004 ACCU article writing competition winners. Not least as this is much more timely this year, thanks to the hard work and dedication of the judges who bravely cloistered themselves in theMorse Bar until the final decisions were made. But also, let me echo Pete's comments: the quality was very high, the choice genuinely tough.

Thanks to all of our authors.

Ewan Milne

Membership Secretaries Report

David Hodge <membership@accu.org>

At the time of writing (Mid May) we have 999 members. The conference gave us our usual boost in numbers over the last two months.

By the time you get to read this the renewal season for most members will be starting. If you would like to save yourself five pounds by paying by standing order just request the details by email.

David Hodge

Standards Report

Lois Goldthwaite
<standards@accu.org>

Five members of the UK C++ panel attended the April meeting of WG21 in Lillehammer, Norway. Much of the discussion during the week centred around plans for the next version of the C++ standard, to be issued late this decade.

This has been the main topic for the last couple of years, but in Lillehammer, for the first time, I have seen a vision of what future C++ will be like. The many proposals on the table boil down to two basic themes: 'say what you mean' and 'programming in the large'.

'Say what you mean': in C++ 0x it will be much easier to use the language to express concepts directly. This will render obsolete a good deal of the template hackery and hard-to-understand non-template idioms which are now necessary to achieve common objectives. One example is `iterator_category`:

```
template<class Iterator>
struct iterator_traits{
  // ...
  typedef typename Iterator
        ::iterator_category
        iterator_category;
};

template<class T> struct
    iterator_traits<T*> {
  // ...
  typedef random_access_
  iterator_tag iterator_
  category;
};
// ... and lots more in
   similar vein
```

This cumbersome machinery, accompanied by a shedload of anastomosing nested typedefs and additional template specialisations in the standard library, exists for the purpose of ensuring that when the compiler chooses among overloaded versions of an algorithm in the standard library, it will select the implementation which offers the highest performance available for the data structures to which the algorithm is applied. This is a laudable objective, but the mechanism is complicated and comprehensible only to C++ experts.

Assuming that proposals to add 'concepts' for generic programming and/or Design by Contract mature in time to be included in C++ 0x, we'll be able to jettison the complex layers of partially specialised templates in favour of some simple (but yet-to-be-formalised) syntax that says, in effect, "this container has Random Access Iterators" and "with Random Access Iterators, use this really fast version of `sort()`".

Another aspect of 'say what you mean' is 'moving comments into code', with checking by the compiler that the code accurately reflects the intent. To continue the example above, if you accidentally pass some argument to a function which doesn't meet the requirements of that function, the compiler can give you a comprehensible error message instead of two pages of gobbledygook template expansions.

C++ is used in many extremely large software applications, such as forecasting the weather or operating the telephone network. Increasing the language's support for such very large projects is a second major theme for the evolution of C++. One approach to this is a scheme for 'modules' to make private symbols really private, so that their names aren't visible in scopes where they might cause unintentional name clashes. And I'm not talking about only class members with private access here -- the scheme could limit visibility to a programmer-specified list of symbols from any namespace, making it easier for an application to link with libraries from different vendors. Resolving conflicts between such independently-chosen names, even when the application itself uses neither version, can cause major headaches to current programmers.

It is hoped that one of the by products of 'moving comments into code' will lead to higher performance, because it frees the compiler from performing some defensive actions which are now thought necessary (one example is committing all variables to memory before a function call, because there's a chance the function might change something or throw an exception).

A project which fits under both themes, and is vitally important to the future of C++, is redefining the 'abstract machine', the black-box description in the standard of what happens when a program executes. The current definition implicitly assumes a single thread of execution. But now that multi-core and multi-processor machines are appearing on the desktop, some rigorous defining of exact semantics (such as when a write to a variable inside a loop must be visible in memory, instead of cached in a register) is needed to make programs running in concurrent environments reliable. A related project, if it can be completed in time, is to add support for multi-threading to the standard library, building on the semantics and underlying primitives defined by the memory model.

Bjarne Stroustrup would probably object that I should have given equal weight to a third theme: 'encouraging novice C++ programmers'. After all, if new programmers don't take up C++, the language will have no future. While I agree with him on this, I expect that being able to 'say what you mean' in C++ -- expressing abstractions and design concepts with clarity and having that supported by compile-time checking -- plus the traditional C++ support for high performance, low-level control, and multiple paradigms, will ultimately prove seductive to novices and experts

alike. (C++ is to programming as sex is to reproduction: other methods may have some technical superiority in this or that aspect, but they're not nearly as much fun.)

These are ambitious goals, and the C++ committee has laid down an ambitious schedule for achieving them. If C++ 0x is to become the next official ISO standard, after several rounds of international voting, a draft document must be prepared in the next 18 months or two years. Proposals for new features will be accepted by the evolution working group until this October, but with limited time and manpower some hard choices must be made on what gets left out. Indeed, there is no guarantee that the proposals outlined above will all attain maturity enough for inclusion.

If you would like to join the work of making this vision come true, please write to standards@accu.org for information on joining the UK C++ panel.

Lois Goldthwaite

Officer Without Portfolio

Allan Kelly <allan@allankelly.net>

Over the last few months I, well, really the ACCU "New Web Group" have made various promises about when our new website would be available. And the more observant of you might even notice that these dates have come and gone and the site has not changed.

So, you might well ask, what is going on?

There are two answers to this question, different but complementary. The short answer is: most of the new web group members (Ewan Milne, Alan Lenton and myself) also had conference commitments so much of March and April was taken up with conference preparation and consequently little time was left for the web.

The longer answer is different. Before we began this project there had been a debate in the ACCU, particularly with the ACCU committee, the question was: do we update the website technology or the website content first? Its chicken and egg, there is little point to putting old content on a new site, but then, content on the old site was difficult to update so no new content happened.

We broke this cycle by updating the technology. The content problem didn't go away it was still there. There is a new website, on new technology, but with the old content. Only now, there is no denying that content is the problem, we now need to update this content.

Our solution to this is to appoint a website Editor. This editor will have equal standing to the editor of Overload or C Vu, their remit will be to update the web content and keep the web site content up to date. They may also decide to make the website into a kind of journal itself.

I don't want to pre-empt the web Editor, this will be a new role – distinct from Electronic Communications officer. The new Editor will be able to define their role, how they want to work, what they want to do with the website. At the time of writing the New Web Group and the Publications officer have been looking for a suitable editor. We think we've found someone, hopefully we can reveal the name next issue.

On the technology front, we're going to proceed to the next stage. We will be asking Turtle Networks to quote for a second stage of work, this should see Mentored Developers, online Journals and the USA site brought into the new system. I'm also hopeful that we will be able to bring forward the revamp of the book reviews section before too long.

I'm sorry the new website isn't yet visible, please bear with us for a little while longer.

Allan Kelly

ACCU 2004 Article Writing Competition

by Pete Goodliffe <pete@cthree.org>

The nominations have been collected, the votes counted, and (since it all happened during this year's conference) the judges have sobered up and still agree with their decisions.

What am I talking about? The 2004 ACCU article writing competition, of course. Every year we provide a little gentle encouragement to write for the ACCU journals, with the lure of a free ACCU T-shirt and untold acclaim (well, acclaim within the ACCU at least). This year we have awarded three prizes: to the best *C Vu* article, the best *Overload* article, and the best article by a new author.

It's been quite a difficult task for the judges, since the quality of articles has been particularly high. Every awards ceremony makes this kind of glib claim, but it's true – there was plenty of material this year to set our judges an arduous task. There was more than a little discussion before the winners were decided.

Congratulations to the winners and our sincere thanks to all those who have taken time to write for the ACCU this year. These articles are the lifeblood of the organisation, so why don't you consider whether you could pick up a pen and submit something? It's nowhere near as frightening as it seems, and there is a group of nice friendly people who'll help get your article through to publication.

Without further ado, here are the results:

Best C Vu Article

Winner:

'An Introduction to Programming with GTK+ and Glade in ISO C and ISO C++' by Roger Leigh.

Honourable Mentions:

'Code in Comments' by Tomas Guest

'Professionalism in Programming' series by Pete Goodliffe.

Best Overload Article

Winner:

'A Mini-Project to Decode a Mini-Language' by Thomas Guest.

Honourable Mentions:

'Where Egos Dar'e by Allan Kelly

'Efficient Exceptions' by Roger Orr

Best Article by a New Author

Winner:

'An Experience Report on Implementing a Custom Agile Methodology' by Aspironi et al.

Honourable Mentions:

'Garbage Collection and Object Lifetime

'by Ric Parkin

Please consider giving some feedback to the authors you read in these pages. It's great to see your name in lights, but once the novelty wears off, it can be quite lonely to write something and get nothing in return. If you enjoy an article, let the author know!

Thanks to the judges for taking time to get this year's competition sown up in a timely fashion. Good luck for next year's competition!

Pete Goodliffe

Dialogue

Comments

In the last issue, we published a piece called "Forgetting the ABC by Orjan Westin". It seems that it has sparked something of a conversation on email between Orjan, Herb Sutter and Andrei Alexandrescu.

I've set it out chronologically with to and from set at the start of each email.

From Orjan to Herb

Hi,

I know you are both very busy, so I don't really expect a reply to this (although I might try to corner you at the ACCU conference, Herb - what's your favourite beverage?) but I thought you might be interested in the following short article, which was published in the latest edition of CVu, journal of the ACCU.

I just wonder if you have ever thought of what I call the ABC principle?

From Herb to Orjan

Hi Orjan, and cc'ing CVu's editor since you said your critique is already published and so he can print this response if he likes.

Thanks for the kind words about C++ Coding Standards. You also wrote:

> What did surprise me was an omission. [...] I just wish they hadn't missed what I have always thought of as the ABC principle - Always Be Conventional. To be fair, it is covered, albeit in a couple of specific instances. Items 26 (Preserve natural semantics for overloaded operators), 27 and 28 (Prefer the canonical form[s of +, +=,++, and the minus equivalents]) and 55 (Prefer the canonical form of assignment) are all expressions of ABC. However, while it is important to follow the principle when dealing with operators, I was surprised that there was no generalisation of it, since it is a common problem anywhere an established convention exists. To be fair, I think you've just overlooked the very first Item 0. :-) Also Item 6. ABC is basic wisdom and common sense, just like KISS, and it bears repeating. While Sutter and Alexandrescu devote four out of one hundred and one items to special cases, they fail to point out the underlying general principle. I do not know whether this was through oversight or because they considered it too obvious, but personally, I feel it is an important principle that is neglected too often.

It is important; that's why we deliberately wrote the opening Item 0 with the repeated mantra of being "consistent" (FWIW I think in this context that's a better word than "conventional"). Here are some specific examples, including seven quotes from that Item where the word is used explicitly:

- general formatting (*"Do use consistent formatting"*, *"be consistent with the style already in use"*)
- indenting (*"Use any number of spaces you like to indent, but be consistent"*)
- naming variables, as in your first example (*"do use a consistent naming convention"* and *"do use consistent and meaningful names and follow a file's or module's convention"*)
- brace placement (*"be consistent: Don't just place braces randomly or in a way that obscures scope nesting, and try to follow the style already in use in each file"*)
- spaces/tabs (*"be consistent: If you do allow tabs, ensure it is never at the cost of code clarity and readability and readability as team members maintain each other's code (see Item 6)"*)

That's all in the first two pages. Note also that this overlaps with the important theme of Item 6, which covers your other examples.

Your article's summary said:

> It's just a matter of following conventions where they exist, fulfilling expectations and avoiding putting in surprises for other developers (or oneself, in six months time).

That sure sounds a lot like our Item 6's opening Discussion paragraph:

> "Your code's maintainer will thank you for making it understandable — and often that will be your future self, trying to remember what you were thinking six months ago."

Hmm. :-)

If you look again, I think you'll find we're in violent agreement that consistency is important. Thanks for writing, and best wishes,

From Andrei to Herb

I concur with Herb, and I'd like to add a paradox that I find funny: Orjan's statement is not congruent with itself.

That is, throughout his discussion he implies "Always Be Conventional" aka ABC as a universal, valid, known, beaten-to-death principle ("ABC is basic wisdom and common sense, just like KISS, and it bears repeating").

Yet, a google search for the phrase "Always be Conventional" yields 68 results, zero of which have anything to do with programming. A search for "Always be Conventional" and "ABC" yields two results, none of which are related to programming, and none of which even mentions ABC as an abbreviation of "Always be Conventional" (or "Always be Consistent" for that matter).

So why is the statement not congruent with itself? Because if the statement were conventional, it would obey to whatever established conventions are out there, and "Always be Conventional"/ABC is simply not out there.

As such, some statements in the short article are simply untrue representing author's thinking and not generally accepted views, as the writing however states quite explicitly. Nothing personal, but I suggest CVu changes the article, pulls it off, or publishes a correction in a future issue.

From Orjan to Andrei

> I concur with Herb, and I'd like to add a paradox that I find funny: Orjan's statement is not congruent with itself.

I, too, am sometimes amused by displays of incongruency. For instance, here you agree with Herb who said, if I may paraphrase, "*It is important and you are saying the same thing as we do*", and then you ask for the article to be pulled. I chose to be amused, rather than offended, since you do say it's not meant personally.

> That is, throughout his discussion he implies "Always Be Conventional" aka ABC as a universal, valid, known, beaten-to-death principle ("ABC is basic wisdom and common sense, just like KISS, and it bears repeating").

There is an expression that goes "the map is not the world". It may be worth thinking about this.

> Yet, a google search for the phrase "Always be Conventional" yields 68 results, zero of which have anything to do with programming. A search for "Always be Conventional" and "ABC" yields two results, none of which are related to programming, and none of which even mentions ABC as an abbreviation of "Always be Conventional" (or "Always be Consistent" for that matter).

I am not surprised. When presenting what I was going to write about, I used the phrase "*what I have always thought of as the ABC principle - Always Be Conventional*."

I say at the beginning of the article that this is MY name for it. Now, had I been an influential writer who had been hammering this phrase for years, it might possibly have become a common name, but I am not, I haven't and and it isn't.

It's my name for a principle, because as far as I am aware it has no established name. One specific instance of it has been summarised by Scott Meyers as "When in doubt, do what ints do", but that is a specialization and I wanted to talk about it in a wider scope.

Having introduced my name for it, I then went on to illustrate what I meant with some examples. Obviously I have failed to delineate precisely what the seemingly offending phrase meant to me, but I'll come back to that.

> So why is the statement not congruent with itself? Because if the tatement were conventional, it would obey to whatever established conventions are out there, and "Always be Conventional"/ABC is simply not out there.

[concluded at foot of next page]

Student Code Critique Competition 34

Set and collated by Roger Orr
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, as are possible samples.

Before We Start

Remember that you can get the current problem set in the ACCU website (`http://www.accu.org/journals/`). This is aimed to people living overseas who get the magazine much later than members in the UK and Europe.

Student Code Critique 33 Entries

Special thanks to **Richard Corden** for providing us with a snippet he came across.

I'm having a problem whose cause I'm not able to detect. I sometimes end up in the true block of the if statmement where iter->first is not '5'. Could you explain me what is going wrong?

```
#include <map>
#include <algorithm>
typedef std :: multimap <int, int> MyMapType;
//
// Filter on values between 5 and 10
struct InRange
{
  bool operator ()(MyMapType::value_type const
      & value) const
  {
    return (value.second > 5) && (
        value.second < 10);
  }
};

//
// Not really important how this happens.
void initMap (MyMapType & map);

int main ()
{
  MyMapType myMap;

  // initialise the map...
  initMap (myMap);
  MyMapType::iterator lower =
      myMap.lower_bound ( 5 );
  MyMapType::iterator iter = std :: find_if (
      lower, myMap.upper_bound ( 5 ),
      InRange () );

  //
  // Did we find this special criterial?
  if (iter != myMap.end ())
  {
    //
    // Yup...we have a value meeting our
       criteria
  }
  else
  {
  }
}
```

[continued from previous page]

I humbly ask that you consider the notion that the principle of which I wrote may be well established, general, basic, well known, and important. I doubt that you have any objections to that notion, since you say you concur with Herb that it is important, and you did spend four items focusing on specializations of it in your book.

It is possible that the phrase I used was unclear - English is not my first language and there might be nuances I am not aware of that makes it say something different from what I wanted to express - and I would be happy to write a clarification if asked to do so.

I imagine that for clarity's sake I should have said that "*This principle is basic wisdom...*" rather than use MY name for it, but since I had both established that this name was something that I personally use as a shorthand for the principle, and (to the best of my ability) illustrated what I meant with "ABC", I thought it would be clear what I spoke about. That it was not is now obvious, and I apologise for my failure. However...

> As such, some statements in the short article are simply untrue representing author's thinking and not generally accepted views, as the writing however states quite explicitly.

What statements are untrue? You imply, it seems to me, that I have wilfully lied and I can't say I am happy about that. I am astonished that you have spent a lot of effort disproving something I have never said, and that you think this warrants pulling the article.

You may, if you wish, contend that the principle is false, unknown or not a generally accepted view. That, at least, would be in reply to something I said. If so, I would happily argue my case, because I believe it is true whatever label is used to refer to it. Not knowing any other, I used my own. That I am the only one using that label has no impact on the validity of the principle to which it refers.

In honesty, though, I have always thought of the principle using a phrase in Swedish, which can be literally translated as "Follow Established Conventions" - something I told the editor in my cover notes. This is clearer and less ambigious, I admit, but given that I think it is a basic principle, and that it could be translated to give the acronym "ABC", I gave in to the temptation to get a title that would, I hoped, express the basic nature of the subject.

As for your Items 0 and 6, I have read them, but I do not think they cover what I was looking for. They are (especially Item 0) at a higher level of generalisation, while the other items I refer to are at a lower one.

In the examples I gave, I was concerned with interfaces, and what established conventions have led programmers assume and expect. I expect things to work in a way reminiscent of similar things. In a container with a STL-style interface, I expect the function "empty" to return a bool and not mess with the data. One could argue that it should do the same thing as "clear" and one could win that argument on all points except one: the convention established by the STL is that is should tell whether the container is empty or not.

These are the kind of conventions I argue one should be aware of and adhere to, and while Item 6 touches on it, I, personally, would have liked an explicit mention of the importance of doing the expected and assumed. It would certainly have been helpful to me thirteen years ago, when I first began working with C++. That is just my personal opinion, though, and I will continue to recommend the book to all C++ programmers I meet.

From Andrei to Orjan

Sorry, I choose to stand by my words. Sorry about the stir that my previous reply has caused. I do think that "Always be conventional" is applicable because it's overly general, but so is NDSSATYWNBAOWYD, which is the self-evident advice that my mom jokingly gives me: "*Never Do Something Shameful, And Then You Will Never Be Ashamed Of What You Did*". Does it apply to programming? Sure. Could I find examples that back it up? Definitely. Could I say "NDSSATYWNBAOWYD is basic wisdom and common sense, just like KISS, and it bears repeating"? No way.

That's where my axe to grind is: comparing ABC with KISS implies that googling for "Keep it Simple, Stupid" would reveal comparable ubiquity with "Always be conventional", and "repeating" means it has been said many times by many people before, which it simply hasn't.

And that's where the discourse ended. So much said over such a small article.

As always, if you have anything to say about the content, please feel free to get in touch.

From Seyed H. Haeri <shhaeri@math.sharif.edu>

First of all, I think I've got to 'fess up that this is the first time – out of three (!) – that I feel satisfied after solving the problem. In the previous times, the problem was more or less trivial, and all we'd got to do was to suggest the programmer how to improve his/her code. This time, we've got a real problem; something which is likely to happen in real programming, and, unfortunately, is not that much uncommon in the commercial world. OK, and now the code, line-by-line:

The first two lines, i.e.

```
#include <map>
#include <algorithm>
```

Seem OK. Next we come across

```
typedef std::multimap<int, int> MyMapType;
```

(I'm wondering whether the lack of appropriate vertical spacing here is due to lack of enough space in the journal, or is that adjusted so by the programmer. If the latter is the case, add this to the list of drawback of the code.)

This line by itself is OK as well. The programmer has correctly observed the necessity of using **typedef**'s, yet he/she has missed making the job complete. That is, he/she is better to add the following as well:

```
typedef MyMapType::const_iterator
        const_iterator;
```

The following two points about the above line worth mentioning:

First, I've chosen to use **MyMapType::const_iterator** over **MyMapType::iterator** as **const**-correctness implies that according to our usage. This will become clearer as we proceed. (Note that Item#26 of Effective STL does not apply here.)

Second, I've chosen **typedef**ed **MyMapType::const_iterator** as **const_iterator** rather than **iterator** to clarify it for the (potential) code reader that I'm using constant iterators.

I'd like to add the following **typedef**s according to similar reasons:

```
typedef MyMapType::value_type value_type;
typedef MyMapType::key_type key_type;
```

Next:

```
struct InRange
{
  bool operator () (const MyMapType::
      value_type& value) const
  {
    return (value.second > 5) &&
           (value.second < 10);
  }
};
```

Although there is no "problem" on retaining this predicate as a function object, it is better to be transformed to its pure function counterpart. (Consult Item#39 for more on the reason.)

Furthermore, those two magic numbers 5 and 10 are asking us to turn them into constants. (See Item#2 of C++ Gotchas for more on magic numbers.) As they apply only to the predicate itself, I'd prefer to make them template parameters. Therefore, here my refined version:

```
template <int lBound, int uBound>
bool inRange(const value_type& value)
{
  return (value > lBound) && (value < uBound);
}
```

The comment of

```
void initMap(MyMapType& map);
```

Urges me leaving it off, and I'll obey that! Afterwards, there is no special point about the following lines.

```
int main()
{
  MyMapType myMap;

  initMap(myMap);
```

until we reach

```
  MyMapType::iterator lower =
        myMap.lower_bound(5);
```

Here is the first place where we should use **const_iterator** instead of **MyMapType::iterator**. I say that because we've nowhere tried to manipulate the result of dereferencing of **lower**.

Furthermore, supposing that this code is going to have enough functionality, 5 turns out to become another magic number. Thus, this is how I refine the above line:

```
const key_type rangeVal = 5;
const_iterator lower =
myMap.lower_bound(rangeVal);
```

The next line is where the programme goes logically wrong. Hereafter, the programmer has wrongly assumed that he/she has found "an exact" occurrence of 5 unless he's reached to the end of **myMap**. And that's wrong. That's exactly what has caused him to get astonished by the result of programme execution. I guess the following snippet from the Standard should make it easier to explain what's going on here (row 9 of Table 69):

> "**a.lower_bound(k)** … returns an iterator pointing to the first element with key not less than k."

That is, it does not guarantee that it will return an iterator to **an element with the exact key k**. Consider the following multi-map for example

{<1, …>, <1, …>, <2, …>, <3, …>, <6, …>, …}

Where **lower** would return <6, …> – in which the key is not 5. Yet it is **the first key not less than 5**. Accordingly, I change the next line to the following:

```
  if(lower->first == rangeVal)//if exact match
       found
  {
    const int lBound(5), uBound(10);

    const_iterator iter =
      std::find(lower,
                myMap.upper_bound(rangeVal),
                inRange<lBound, uBound>);

    if(iter != myMap.end())
    {
      //…
    }
    else
    {}
  }
}
```

A supplementary point about the above programme is that as Scott Meyers explains in Item#45 of Effective STL, if you need to know where the **range** (potentially) containing what you need is located, you should use **std::equal_range()**. Considering that, the programme will become:

```
std::pair<const_iterator, const_iterator> p =
  myMap.equal_range(rangeVal);

//See Item#45 of Effective STL for why this
      works

if(p.first != p.second)//if there is an exact
     match at all
{
```

```
  const_iterator iter =
    std::find(p.first, p.second,
        inRange<lBound, uBound>);

  if (iter != p.second)//we've got a value
      meeting our criteria
  {
    //…
  }
  else
  {}
//…
```

This programmer seems to know enough of the basics of STL programming. Yet, he/she lacks deep-enough knowledge of STL. I suggest reading the following books by order:

- *Generic Programming and the STL: Using and Extending the C++ Standard Library* by Mathew H. Austern
- *Effective STL: 50 Specific Ways to Improve Your Use of Standard Template Library* by Scott Meyers.

From Nick Buller <nickbuller@hotmailcom>

Program Structure

As has been mentioned many times the program entry point `int main()` is defined as returning an `int` value but no return statement is present in the main code block. The general program structure should be as follows:

```
int main()
{
  // Program code omitted for brevity
  return 0;
}
```

As mentioned in Francis' commentary for SCC28 "... the default form of the definition of main should encapsulate its code in a `try` block".

I also add the correct include and the using block as I prefer the code not to be littered with `std::`, so the main function becomes:

```
#include <iostream>

using std::endl;
using std::cerr;

int main()
{
  try
  {
    // Program code omitted for brevity
  }
  catch(...)
  {
    cerr << "An unexpected exception"
            " occurred." << endl;
    return 1;
  }

  return 0;
}
```

Constants Not Magic Numbers

As with reams of code that I have tried to decode the code contains magic numbers. I believe that the code I write will be read more often by other people then by myself (lets take it as a given they are looking for a bug). So we should try and make life simple rather than expect they automagically understand the magic number left provide them with a good starting point.

Rather than writing

```
return (value.second > 5) && (value.second <
       10);
```

Write the following

```
static const int someMeaningFullName = 5;
static const int someOtherMeaningFullName
    = 10;

...

return (value.second > someMeaningFullName)
  && (value.second<someOtherMeaningFullName);
```

Comment What Are You Trying To Achieve

Now that you have code that can be understood a comment would not hurt.

Variable Names

Forgive me but `MyMapType`, or should it be `OurMapType`, Ummm. The name of the `mapType` should indicate its use. Unfortunately as no comments, well-defined constants or comments exist no recommendations can be made.

On a Personal Note

I dislike the following code for 3 reasons: -

```
MyMapType::iterator lower =
      myMap.lower_bound(5);
MyMapType::iterator iter = std :: find_if
      (lower, myMap.upper_bound(5),
      InRange());
```

1. A local variable is used to store the lower bound iterator but the second is called directly and passed to the function.
2. `myMap.upper_bound(5)` returns an iterator by value so there is no reason not to store it in a local variable as is used for the lower_bound return or at least be consistent.
3. The call to `std::find_if` is split on 2 lines and the second line is not indented.

Code like this can be a nightmare to read.

```
MyMapType::iterator lower =
      myMap.lower_bound(5);
MyMapType::iterator upper =
      myMap.upper_bound(5);

MyMapType::iterator iter = std :: find_if(
      lower, upper, InRange());
```

A Solution

A single line change to the code would provide a fix to the problem: -

```
if (iter != myMap.end())
```

would be replaced by

```
if (iter != upper)
// or if (iter != myMap.upper_bound(5))
```

The Problem

To understand what is happening we need to look at the definition of the multimap and its members:

```
template <class Key, class T, class Compare =
   less<Key>, class Allocator = allocator<T> >
```

The definition of `myMap` is only parameterised on `<int, int>` so the default `Compare` and `Allocator` will be used, in this case we are interest in `less<Key>`. If `f` is an object of class `less<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns **true** if x < y and **false** otherwise.

```
const_iterator lower_bound(const key_type& x)
      const;
const_iterator upper_bound(const key_type& x)
      const;
```

- `myMap.lower_bound(5)` will find the first `myMap` element whose key is not less than 5, if no value is found then `myMap.end()` is returned.
- `myMap.upper_bound(5)` will find the first `myMap` element whose key is greater than or equal to 5, if no value is found then `myMap.end()` is returned.

If the map contains the values (5, x), `lower_bound` will return an iterator to the one and only entry and upper bound.

Commentary

I suspect that many readers were scared off by this SCC, which reflects people's lack of familiarity with `std::multimap`. If you are one such reader then I encourage you to explore the range of associative containers in the standard library – in my experience most people always use `std::map` even where another container might be better.

The fundamental problem with the code shown is the test

```
if (iter != myMap.end ())
```

to detect failure to find an element in the map. What is wrong with this test? The value of `iter` on failure is the end of the range not the end of the *container*. The range ends with `upper_bound(5)`, so this is the value against which `iter` must be tested to see if a match was found.

In this example the use of `multimap` was really a bit of a red herring and simply having a good understanding of the use of standard library iterators would have been enough to spot the bug.

A subsidiary problem is the lack of detail in report of the problem. When a failure occurs it would be more helpful to know what the value of the key is (and ideally the complete contents of the container). This could be a good place to start talking with the student about error handling, fault reporting, and designing for testability.

The first solution presented above does fix the bug and moreover covers some other problems with the code. There is even a quotation of the relevant section from the standard to explain what is going wrong. Unfortunately the standard is designed for precision and the technical vocabulary used is not always clear to casual readers!

I do however have a (minor) criticism with the above solution in that I think the first book listed may be too advanced for a student at this stage. I would suggest getting started with the collection classes via a good basic tutorial to using the STL, such as "The C++ Standard Library" by Nicolai Josuttis and progressing to Austern's book later on.

The second solution presented makes it easy for the student – a one line solution to the bug is given. The first remark – that main requires a "return 0" - is one of those things that is true in practice but not in theory… The standard states that main is a special function and that a return statement is not required. One popular compiler does not support that exclusion – so for maximum portability it is probably a good idea to add an explicit return statement.

The Winner of SCC 33

The editor's choice is:

Sayed H. Haeri

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

Student Code Critique 34

(Submissions to scc@accu.org by July 10th)

Here is a problem with a 'C' program. Please try to both solve the student's current problem and also help them learn from this mistake.

I send an unsigned char called tipus which is meant to be some hex number from 0x00 to 0xff and which should decide the string to be returned...
If the unsigned int called valor I send is above 0xA000 I use the struct "decidetest" to send a string back, else, I send a string selected from the struct "decide".
If no substitution is made, then the string returned is always a space in html..." "
For some reason I always get the "static char escape" string returned... any idea? (the function compiles all right...)

```
char* Detect_type(unsigned char tipus,
      unsigned int valor)
{
int i;
static char escape[16] = "         ";
static struct decide {
unsigned num;
char *string;
} cadena [] = {
        {0x00, "Sobre V PK      "},
        {0x02, "Sobre V RMS     "},
        {0x0E, "——Power OFF—-"},
        {0x10, "——Power ON——"},
        {0xff, "                "},
};

static struct decidetest {
unsigned numero;
unsigned char num;
char *string;
} cadenatest [] = {
        {0x00, "Sobre V PK  Test"},
        {0x02, "Sobre V RMS Test"},
        {0x0E, "——Power OFF—-"},
        {0x10, "——Power ON——"},
        {0xff, "                "},
};

if (valor >= 0xA000)
  {
    for(i = 0; i < sizeof(cadenatest) /
        sizeof(cadenatest[0]); i++)
    if(cadenatest[i].num == tipus)
      return cadenatest[i].string;
  }
else
  {
    for(i = 0; i < sizeof(cadena) /
        sizeof(cadena[0]); i++)
    if(cadena[i].num == tipus)
      return cadena[i].string;
  }

return (escape);
}
```

Francis' Scribbles

by Francis Glassborow

Something Deeply Unpleasant

A couple of people (with regret, one of them a long-term ACCU member) having been gossiping about a confidential Standard's issue that concerns Lois Goldthwaite and myself. I do not intend to add more fuel to an already blazing pyre by going into details here.

However, '*do not repeat rumours that can prejudice another professional*' is a guideline that should be automatic among professionals. To that, I would add that anyone who ignores such a guideline is behaving unprofessionally. Of course, we all have a tendency to gossip but we should make an effort to avoid doing so when people's reputations are at stake. We should also have the professionalism to remind a gossiper firmly that rumours can have serious consequences.

For the record, I was not a member of the UK delegations to the recent meetings of WG14 and WG21 at Lillehammer. I actually attended both those meetings representing Plum Hall (Tom Plum's Hawaiian based company) and as a reciprocal liaison between WG14 and WG21. However, you should not make any deductions from that, though some seem to wish to do just that. You should also note that Lois Goldthwaite attended the WG21 meeting as HoD for the UK. She is a well-respected member of the

International Standard's community and the editor of the Performance TR. Any suggestion that she would not have been part of the UK delegation to WG21 is clearly ludicrous

The Evolution of C++

At the recent meeting of WG21 in Lillehammer (Norway) someone commented that evolution took place by random mutation rather than by intelligent design. So perhaps Bjarne Stroustrup should be given the title 'Random Mutator of C++.'

Joking aside, there are many proposals for change and enhancement of C++ on the table. These are not exactly random but they do depend upon what matters enough to individuals for them to take time writing a paper making a proposal. There are probably many other good ideas that just have not been put forward.

If you have something that you feel you want to be considered then you have very little time left. We decided in Lillehammer that we definitely would not consider new proposals for change to the core of the language made after the pre-meeting mailing for the October meeting. It would also be fair to warn you that we are making no guarantee to consider new proposals made between now and then, just that we will not reject them out of hand. At this stage, writing a paper and expecting someone else to present it for you will be a waste of time. If you do not care enough to be at the meetings to promote your proposal, it will simply die (unless someone else feels that it has a great deal of merit)

The current number of proposals is extensive. The vary from very simple additions and changes to keep C and C++ in line with each other (for example, adding `long long` – yes you may hate it but continued resistance would simply use energy that is better spent elsewhere) to extensive proposals such as those for concepts. We have come a long way since the early meetings of the evolution group three years ago. We are beginning to see what it is that we want to achieve. I have little doubt that some of the newer participants will feel hurt when we do not select their proposals. Several of these people have specifically joined WG21 so that they could promote an idea that they believe to be valuable. These people are probably right, but that does not mean that the value will be great enough to incorporate them into the next version of C++.

Now we have all the mutations, sorry proposals, on the table we enter into the phase of natural selection. There is no reasonable way that all the excellent proposals will make it into the next version of C++.

I often suspect that most people have little idea of how much work goes into adding even the simplest feature. No one doubts the good intentions and hard work of those making proposals but they have to realise that they are asking other people to do a great deal of work as well. Simple assurances that something will work as described are not enough.

If you have the slightest doubt about this, consider ADL (argument dependant look-up). The original proposal for this came from a highly respected source and it was only after several years of intense experience that the holes began to show. We do not want to make the same kind of mistake again. C++ is not some instant language that can be fixed on the fly or by the next release from its owners. C++ is a language for serious programming and needs stability if it is to meet the needs of its users. Changes to a widely used language are expensive if the language is used for anything other than instant programming for instant disposal.

Now, even if you have not made any proposals for changes and additions to C++ you can do your bit to help the selection process. Have a look at the proposals (see `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/`) and decide which ones look particularly attractive to you. When you have done that, offer to help those making the proposal. If you want something, be willing to give it a push to try to get it further up the ladder.

Get involved, pick a proposal you like and write a short (or long if want to) piece explaining why you would like to see it in C++. Send it to me and I will both see it is published here and that those concerned with WG21 see it. We cannot do design by democratic vote but that does not mean that we do not listen to the opinions of users.

Nomeclature

I find terminology is one of the irritating problems with programming in general and language standards in particular. I am in the midst of writing my second book and want to be careful to use the correct terms so that readers will not be confused by colloquial usage that is at variance with correct usage. However, I find this to be difficult to do in practice.

I am happy with the distinction between declarations (about names) and definitions that provide a meaning for a name. Perhaps it is unfortunate that:

```
int foo();
```

is a declaration, `whilst`:

```
int i;
```

is usually a definition (as well as a declaration). The insistence on calling:

```
class x;
```

a forward declaration further clouds the issues. Surely, that is just a declaration. What is making it even harder to write about C++ is describing what the sort of thing we are declaring in each of the following (in other words what kind of name are we introducing):

```
int i;
int& i_ref (i);
int bar(int parm, int & parm_ref);
```

I think most of us would agree that `i` is a variable. In most, but not all contexts, `i` designates specific storage for an integer value. However, what kind of thing is `i_ref`? I am sure that most people are happy to call that a variable as well. It behaves like a variable; it looks like a variable and so on. However, according to the strict letter of the C++ Standard it is not a variable but a reference. References may look like variables but they are something else. What makes it much worse is that the Standard wants to talk about `i_ref` as a reference that refers to `i`. That is not true either; `i_ref` is a name that designates the same object that `i` designates.

Does all this precision help us talk and think about our code? From where I am sitting, C++ has objects and names. Names designate objects (they can also designate other entities such as types and functions). It is possible to have an object that is not named (we often call these temporaries, but that is not strictly correct because dynamically created objects are also nameless).

References always refer to objects; variables always refer to objects. The difference between them is that a reference is bound to an existing object at the point of definition but a variable is bound to a newly created object at the point of definition. Confused? Well I am. I have to be extraordinarily pedantic if I am to avoid misspeaking.

What is wrong with referring to a name that designates an object as a variable? In almost every circumstance, a variable and a reference are interchangeable. If it looks like a variable and behaves like a variable let us call it a variable. Let us use the term ‘variable’ for any name that designates an object. That should include `parm` and `parm_ref` in the function declaration above.

Why should we care? We should try to avoid such messes as those created by the many authors who confuse ‘declaration’ and ‘definition’. That lead to the silly term 'forward reference' because so many writers used the term 'class declaration' when they meant 'class definition'.

Reference Type

Is there such a thing as a reference type? We have pointer types which are quite distinct from the type to which they point. We also have `const` and `volatile` types. Those are also different to the type they qualify. But my question is whether there is any such thing as a reference type, or is the term just a lazy way of talking about a reference to a type.

For example:

```
int i(0);
int * i_ptr(&i);
int & i_ref(i);
```

`i` is of type int; `i_ptr` is of type `int*` (pointer to `int`). However, what is the type of `i_ref`? Surely, it is of type `int`. Or is it? Is there any way that

we can distinguish between a reference and a variable? Yes, this connects back to my problem with distinguishing between names that are variables and names that are references.

Both `i` and `i_ref` identify an object of type `int`. Once we get into user defined types a reference has the property that it has two types, the static type (the type it is declared as) and its dynamic type (the type of the object it references). These two types do not have to be the same. However, notice that we write about the static and dynamic types of a reference but we do not say that either of these is a reference type. So I ask again, is there any such thing as a reference type in C++.

I am coming to the conclusion that there is not but I would very much like to hear your opinion before I finally commit myself in my new book. I have no problem with writing that something is a reference type in an informal situation but when I am trying to explain the C++ type system I do want to be correct.

I think that as long as we carefully explain terminology that using a short cut is perfectly acceptable. As long as we have stated the difference between a name being a variable and a name being a reference we should be allowed to shorten the phrase 'variable, reference, parameter or reference parameter' to 'variable'. Just as writing definition is a legitimate abbreviation for 'declaration and definition.'

PC Notes Taker

From time to time, I come across pieces of hardware that seem to show some promise. One of my on going irritants is that manufacturers often fail to understand their product and so fail to make full use of it.

The hardware for this product is very simple. It consists of a special pen that signals its location when the nib is pressed for writing (with a cartridge of ordinary biro type ink) and a clip similar to that you have on a clipboard. All the special electronics is hidden in the clip and receives the pens signal. It transmits the information about the pen location to the PC through a USB connection.

It comes with some special software to handle the data supplied via the USB port. This allows you to draw on an ordinary sheet of A4 paper held in the clip, an application program captures what you draw on the paper. Out of the box, that is it. There are a couple of extra programs available on the web site and if you have Microsoft's handwriting-to-text engine one of these allows you to write on the paper and have it converted to text. That is quite impressive, but the Microsoft software is doing most of the work.

The first thing to note is that this product is MS Windows specific. The provided software is of no use if you have an Apple machine or a Linux based PC. That is silly; it is not that hard for a reasonable programmer to supply software for the other common operating systems.

[It is not that simple, mainly due to the way that different operating systems access hardware and to my knowledge, there is not any platform independent way to do this – PFJ]

The next problem is that the software interferes with some other uses. I have to remember to exit from the support software if I want to play most modern computer games, because the supplied software puts the game into sleep mode. Not a big problem, but one that could and should have been avoided. Moreover, I wonder what other software will have similar problems of co-existing with the PC Notes Taker software.

Now the big issue is why they think that they should be selling such a limited set of applications for their hardware. Given the way the product works, it clearly has the potential to be a mouse substitute. Even better, we could use the product to create a form, print out multiple copies and use those for capturing data. The handwriting to text would convert from the handwritten paper copy to an electronic version.

I am guessing, but I think there are a good number of people who would welcome such a tool.

If the manufacturer published the interface information for the hardware all the above problems would be solve. There are dozens of competent programmers who would jump at the chance to add value to the hardware. As it is, the product will have a minuscule market, remain relatively expensive and will die as things like tablet PCs become popular and cheaper.

When will the industry learn that trying to do everything results in an uncompetitive product? Let others enhance your product and you end up with lots of sales without too much investment. Just look at what happened to the PC once IBM gave up trying to control it.

Even better, letting others write extra software promotes the product because the software will help make the hardware better known. If I see a piece of software that solves a problem for me if I buy a relatively cheap peripheral, I am quite likely to make the purchase. Software sells hardware, the reverse is very rarely the case.

If you have experience of this product or products like it, I would be happy to hear from you and publish your comments and experiences for the benefit of others.

Problem 20

Here is a miniscule program. Now I think it is clear what the programmer intended but what do you think a conforming compiler will do with it and why? How should the author have written the definition of `p`?

```
struct data {
     int i;
     int j;
};
int main() {
     data** p = new (data*)[5];
}
```

Commentary on Problem 19

What is the portability problem with the following small C++ program?

```
#include <iostream>

int main(){
   std::cout << "Hello World" << std::endl;
}
```

At first sight, there seems to be nothing wrong with that program and I guess you will find it (with minor variations) in almost every introductory text on C++. However whether a compiler will compile it depends on the implementation.

The problem, which comes as a surprise to a great number of C++ experts is that there is no requirement in the standard for the `iostream` header to include either `istream` or `ostream`. Without the `istream` header the compiler will not find a declaration of *appropriate* versions of `operator<<` nor of `std::endl`.

Now we can make a good case for such omissions because, for example, a programmer might not want to suck in the whole of `istream` just to use `std::cout`. However, the result is to create a surprise for programmers the first time they meet an implementation of `iostream` that does not include `istream` and `ostream`.

Cryptic Clues for Numbers

Here is last issue's clue:

A first course on C++ at the University of Rome [3 digits]

I guess that many of you found that a tough clue. The two parts of the clue are interwoven. First clue consists of 'first course' and 'University'. The second part is 'C++' and 'Rome'. Both give 101 as the answer (i.e. 100 + 1).

Twice lucky? Thrice lucky? About when China ruled the seas.

Note that the second part of this clue will not be general knowledge for most people. But if you can get the first part, Google will help confirm it.

Francis Glassborow

Francis Glassborow (francis@robinton.demon.co.uk) is a freelance computer consultant and long-term member of BSI language panels for C, C++ and more recently Java and C#. He is a regular member of the UK's delegations to WG14 and WG21. He is also the author of 'You Can Do It!' and introduction to programming for novices.

Features

ACCU Conference 2005

A Retrospective

by Pete Goodliffe <pete@cthree.org>

April 2005, and hundreds of techies – from Goths to mild-mannered janitors, hardcore C++ programmers to eager students – converged in Oxford for the annual ACCU conference, a feast of technical learning, peer-to-peer discussion, and plenty of late-night drinking. This short article is an attempt to describe the event to those who missed it, to act as nostalgia for those who were there (and sober), or to inform those who were there (but drunk) what they missed.

As last year, the conference was held in the impressive setting of the Randolf hotel. This met with mixed reactions – many prefer such a central Oxford location, although for others it's far too difficult to reach. Many dislike the prices at the bar, others hardly noticed. The Randolf certainly added a grand "atmosphere" to the event. Airconditioning was a happy improvement over last year's oven-like conditions, although the lunchtime fare wasn't vastly superior (and seemed to be an exercise in covering your clothes with mayonnaise: don't ask).

The line-up of speakers was excellent, and the coverage of subject matter was broad. There were tracks devoted to languages (C++, Java, Python, with .NET material too) and also to process and security – something for everyone. Speakers included industry luminaries like Bjarne Stroustrup, Jim Coplien, and David Abrahams, as well as numerous people from the ACCU ranks. The four day timetable[1] was packed with talks, seminars, panel discussions, vendor presentations, evening events, and more.

A large part of the ACCU conference is the chance to meet up and socialise with fellow programmers. This was carried out with aplomb after the day's festivities (and often continued into the early hours of the next day). The conference is a great opportunity for ACCU members to put names to faces, to insult one another (who was 'chav'?), and to exchange war stories. The now-regular events: Blackwell's evening reception and the speaker's banquet, provided a great chance to mingle and meet different people.

To give you a flavour of what went on this year, a few people have provided their views on the event. First up is Timothy Wright – an ACCU conference virgin. These are his thoughts on the event:

"This was the most inclusive and interactive conference that I have been to. There were nightly group outings to dinner and the pub and everyone was welcome. Sometimes we overwhelmed the restaurant. I met many people who post on accu-general and it was good to put names with faces.

The conference had two main benefits for me. The quality of the talks and the presenters, and the ability to mingle and talk to others. Some of the key points I took from the talks are:

- That bugs and security issues in software can be classified by mean time to bug – how long it takes to find a particular issue. The security talk on this subject brought me back to college since the professor used handwritten viewgraphs which were hard to see from the back of the room.
- There are significant differences between the implementations of generic programming in Java, C# and C++, and they have to do with specific design goals.
- Concurrency is hard; even the experts get it wrong sometimes.
- There is some cool stuff coming in the C++ 0x standard including concepts, move semantics, class extensions etc. Many of these new features are designed to help the beginner or part-time programmer and to remove the necessity to become an expert and learn the tricks. Unfortunately, we will have to wait for 2009 for the standard and 2011 for a compliant compiler.
- You too can extend the iostreams with a few simple rules.
- That symmetry is good except for when it is bad.
- That software development is learning. However it does matter how you define knowledge and learning especially if Coplien attends the talk.
- If you want to sit down in a talk, you must arrive early.
- That software is really about solving business problems not about applying object-oriented techniques. It is about writing specific software not general software."

Mick Brookes, another conference newbie, offered these thoughts on some of the presentations he went to. The first, Kevlin's Consolidated C++, was from the one-day pre-conference schedule:

Kevlin Henney – Consolidated C++

"Kevlin first explained about the session name – apparently it's difficult to convince programmers (and their managers) that an 'Intermediate' C++ course is going to be useful, but he couldn't bring himself to contribute to a name inflation problem and go with 'Advanced'. He then lead us through a number of topics: Value-based programming, an Intro to Templates, Encapsulation and Cohesion, Generic Programming, and Class Hierarchies. Along the way, he was quick to show us just where things could be improved in the C, C++ and CORBA standards, and taught us never to feel embarrassed about pulling a door marked push.

The audience had varying levels of experience with C++, but I'm sure that everybody learned something new, and had to re-examine some of what they thought they understood. Down in the shallow end, I was thrown enough familiar material to feel safe about being stretched in the areas I had less experience with. 7 hours clearly wasn't long enough, with a whole section on exception handling having to be cut, but I still think it'll take me at least 6 months to absorb everything raised on the day and in the notes."

Paul Grenyer – Aeryn: C++ Testing Framework

"Paul gave us a bottom-up introduction to Aeryn, his new lightweight testing framework, starting with test conditions and building up to named test sets. Not only did he explain the features, he took the time to explain why they'd turned out that way - going so far as to challenge us with, "Aeryn uses macros, anyone got a problem with that?" Thinking about it now, I'm sorry nobody spoke up (I think the accompanying glare put us off...) - Paul must've been ready to put us all straight, and I'd definitely have learned something. He showed us how customizing the result reporting format only requires plugging into a simple interface, and then explained how Mock Objects can help break dependencies in test code and speed up the feedback loop.

Aeryn has a feature that blew me away: if an object being tested is streamable, the test output streams it; otherwise we get a sensible default. Paul and Richard Harris helped show me how this magic worked after the session - giving me a first taste of SFINAE."

[concluded at foot of next page]

1 Well, actually it was a five day program since this year saw the introduction of a "preconference" – a one day event covering introductory material.

Bjarne Stoustrup's Fiver

by Jez Higgins <jez@jezuk.co.uk>

Back in the mists, when I first started programming it quickly became apparent that the bulk of the tools I used everyday were invented or written about by Brian Kernighan. I was making my living on the back of the work of many people, but Brian's contribution to my take-home pay seemed the most significant. I developed the idea of Brian Kernighan's Fiver. If I ever met Brian Kernighan, I'd give him five pounds as a symbol of the living he'd given me.

As time went by Brian's everyday significance declined, and his place in my little pantheon of thanks was taken by Bjarne Stroustrup. I decided that should I ever meet him, I'd give him a fiver too.

Last Thursday, that opportunity finally arose. There I was in the bar. There he was. I had a fiver in my wallet. A normal fiver, not a special one I'd been keeping or anything like that. Just an everyday five pound note.

Bjarne was talking to Cope, among others, and frankly Cope rather frightens me so I was loathe to interrupt. After 20 minutes or so, he got up to leave. He was going to walk right past me. Fate was lifting her skirts and taunting me. I had to do it.

"Dr Stroustrup? ..." Quickly, I explained about the fiver, and offered it to him. He declined. "I'm going to have dinner with my wife," he continued, "perhaps we could use this buy a drink later on."

A drink with Bjarne Stroustrup? Blimey. But, hang on. We're in the bar of the Randolph Hotel in Oxford, seemingly one of the most expensive bars on the planet. A fiver here doesn't go far. Then it hit me - a fiver wasn't enough. Bjarne's trying to tap me for a tenner!

The disappointing end of the story is that I didn't get to buy that drink, because our bar trajectories didn't cross again. I used the fiver on car parking. Bjarne no doubt thinks I'm a nutter.

A stingy nutter at that.

Jez Higgins

[continued from previous page]

Allan Kelly – Viewing Software Development as Learning

> "To write software, we must learn the problem domain, and lots of technical skills, and lots of process skills. The act of writing software enables ourselves and our customer to learn more about the problem, and often leads to changes in requirements. Allan used this session to present his ideas on learning, and on how acknowledging the big role it plays allows us to improve the development process. He suggested that books and presentations etc. are mere information until they're acted upon, and that the action allows learning to take place, turning information into knowledge. This means that learning and change are closely connected, and that separating them can be unproductive. Allan's presentation resulted in a lively discussion with the audience (see www.allankelly.net), and revealed that lots of people seem to be worried about learning stuff that's just wrong."

Mark Easterbrook, a conference regular, had these thoughts on some of the specific presentations:

Herb Sutter – Genericity in .NET, Java and C++ (2 sessions)

> "An excellent presentation by Herb comparing generic programming implementations comparing and contrasting Java, C#/.NET, C++ generics in .NET and C++ templates. This was in typical Sutter style switching between presentation slides, a code editor and the command line, and between Microsoft and Linux. This session provided two of the humorous moments of the conference: there was no obvious way that Herb was changing slides in the presentation so someone asked how – the answer was that he had a remote mouse in his trouser pocket, to which someone had to ask what he was operating it with. Later, after one of Herb's jibes at Microsoft, a mobile phone rang in the audience, and someone was quick to call out "It's Bill"."

Prince/Schneider – 90 minutes from the end

> "The sub-title for this partly interactive presentation was "Patterns for the end game". Although the presenters' intent was to cover the problems that occur at the end of projects and how to overcome them, much discussion ensued as to whether they were really end-of-project problems and solutions. If a problem occurs at the end due to mistakes earlier on, it is not really an end-game problem, this is just where is manifests itself. Likewise, problems that can occur anywhere, but are much more serious or obvious towards the end, are not end-games problems, just more visible then. The prepared conclusions backed up the discussions in that end-game planning is just part of the overall project management."

Matthews – Concurrency Requirements

> "Hubert started by explaining that concurrency problems are rarely found in the real world – you never see two skeletons locked in a deadly embrace – so they are a machine issue for which we have no natural ability to understand. Also, concurrency is rarely an explicit user requirement – it is an issue in solution space rather than the problem domain. Traditional and contemporary methods of writing correct software and proving it correct such as unit testing, invariants, pre- and post-conditions, and regression testing don't work for concurrency because the problem is temporal, rather than sequential, and a change in the timing can radically change the outcome. Hubert covered a number of real world examples to demonstrate there is no obvious solution, and maybe no solution at all. There was considerable discussion during and after the presentation indicating considerable interest and concern about the subject. This was a thought provoking presentation about a subject we are likely to hear much more about in the future."

Vollmann – Threads considered Harmful

> "Detlef aimed to dispel any remaining thoughts that concurrency is anything but very difficult. First he recalled the number of experts who thought they had solved concurrency and then were proved wrong – often after publishing a "definitive" book! Then he showed how the atomic operations required by concurrency (e.g. popping off a stack) are opposite to the requirements of exception safety provided by the STL (top(), void pop()) – you cannot have both thread safely and exception safety in the same operation. He peppered his talk with practical demonstrations of a Mandlebrot graphics GUI window with concurrency "solutions" and either their failure to solve the initial problem or how the new problems they introduced were worse than the original problem. Discussion and questions at the end overran the time allocated suggesting that this is a subject of great interest and concern to a lot of developers."

Henney – It's all geek to me.

> "Kevlin hosted an interesting and lively finale to the conference in the form of a lively panel quiz with considerable audience participation. The last slot on Saturday is a difficult one to fill – many delegates need to leave early to catch transport links and those left are often flagging from long days and even longer evenings! Both an endnote and running normal slots have been tried in the past and some alternative was sorely needed. This was the right ending, and the right person to present it – Kevlin took no prisoners, even the conference chair was not spared as he tried to keep score to many decimal places as whole, fractional and sometimes decimal points were awarded, and sometimes taken away. Some of the subjects covered were literature (C++ standard to the Hitch Hikers' guide to the Galaxy), people (Dennis Ritchie's middle name), and guess how the audience will vote on a vague question. Fortunately the four panellists walked away with some decent prizes that Kevlin had managed to extract from the sponsors, so their humiliation was not totally in vain."

This gives a flavour of the range and quality of material presented this year. With over 70 sessions there was a lot to take in.

All the attendees owe a debt of gratitude to the conference committee, the event organisers, to the speakers, and to all who made this year's conference what it is. If you haven't been before, then I hope that this article enthuses you – I'll see you there next year!

Pete Goodliffe

Out with the old...

Pippa Hennessy <pip@oldbat.co.uk>

So, the time has finally come for me to bow out of the ACCU Production Editor role. I'd intended to carry on doing it for a few more years but an unexpected promotion has brought a vastly increased workload in my day job, and this combined with greater child-care responsibilities as my sons' father is working away from Nottingham has meant I couldn't devote the time necessary to producing the journals to the high standard I've tried to achieve.

I'm sitting here scratching my head, trying to remember how long I've been doing the job. It feels like forever... but at the same time it seems like only yesterday I was down in Oxford learning the tricks of the trade from Francis Glassborow. I owe a huge thank you to Francis for all his help in the early days, his advice and support was invaluable, and without him at least one pair of journals wouldn't have arrived at the printers on time (it's a long story involving the premature demise of a hard disk and almost but not totally matching font sets).

I have to say I've thoroughly enjoyed learning the skills required to produce a professional quality magazine. I'd edited smaller-scale magazines in the past, for example I produced the Notts County Bridge Bulletin for a couple of years, using Microsoft Word for the layout and photocopying and stapling the copy myself. However I'd never used a desktop publishing package, and never had to manage a large-scale production process like this before, so it was a bit of a shock to the system. I've been extremely impressed by the dedication of the committee and the editorial team, all of whom are volunteers, yet all of whom have worked above and beyond the call of duty to provide you with invariably high-quality journals every two months without fail. I'd especially like to thank the editors past and present (James Dennett, John Merrells, Alan Griffiths, Paul Johnson), various committee members (David Hodge, Pete Goodliffe, Chris Lowe, Thaddaeus Frogley, Ewan Milne), Alan Lenton for the fabulous cover images, all the contributing authors and columnists, and others too numerous to list. Thanks also to Ian and all the team at Parchment, and Simon and John and all at Able Types. You've all been an absolute pleasure to work with.

Enough now, this is starting to sound a bit like an Oscar acceptance speech. The tears are flowing freely down my long silk gown even as I type... It only remains to say that you haven't heard the last of me - now I've got all this spare time (!) I'm hoping to contribute a few articles, and I'm planning to get involved in the editorial team as a reader/reviewer. I'll certainly keep reading the journals, although perhaps not in quite as much detail!

All the best of luck to your new Production Editor - Alison Peck - although I'm sure with the extensive experience and skills she has, and such a professional and committed editorial team, she won't need any luck.

Pippa Hennessy

And in with the new...

Alison Peck <ACCU@clearly-stated.co.uk>

I've been working as a technical author for over 9 years now, but not always with that job title. I left permanent employment at the end of October last year to have a go at freelancing - basically, if it involves words or training, I'm interested. I've mainly (but not exclusively) documented software in the past, covering the full range of users from novice to developer and I've also been involved in training all sorts of stuff, software-related and anything else I've been asked to do. Technical authoring is a second career for me - in a previous life, I was a nurse.

I live in Nottingham and I've been here for over 20 years -my family has nearly all left home now, with the youngest off to university in the autumn. I just need to find some way to stop them coming back!.

Hobbies? I haven't got time at the moment - the dissertation for MA is due in July so that and trying to establish some freelance stuff means I'm kept fairly busy. I've started to keep a few tropical fish, but nothing too intense.

Alison Peck

Silas's Corner

Cross-Compiling Python Scripts into Windows Applications

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

I recently wanted to distribute my Python program to a few other people who were running Windows computers and who were not technically minded. Asking them to go through the process of installing a Python interpreter on their systems was not an option.

py2exe (from `py2exe.sourceforge.net`) is a fairly simple approach to converting Python into standalone Windows programs. Py2exe installation will modify an existing Windows Python installation, but both Python and py2exe let you install without administrator privileges, so if you have user access to a Windows system then you can install both in a temporary directory.

Cross-Compiling from Another Operating System

Py2exe only runs on Windows, so if your usual operating system is not Windows then you will have to move to Windows every time you want to release a new version of your program, which is not ideal. I wanted to find a way of making a Windows standalone executable without using Windows to do it.

The Windows emulator WINE is not yet up to the task. However, if you can get to a Windows system to run py2exe once, you can do it in such a way that incrementally maintaining the result is possible.

By default, py2exe creates a `.exe` file that contains your main module, but all other Python modules (whether supplied by you or part of the library) are put into a file called `library.zip`. So what you need to do is get py2exe to compile a small "wrapper" module that imports your main program from another module; that way, your main module will also be stored in `library.zip` where you can maintain it yourself without having to touch the `.exe` file.

Put all the modules of your script into a directory on the Windows system, and add a simple wrapper such as this:

```
# This is wrapper.py
import my_main_module
my_main_module.main()
```

Then add a suitable py2exe-compatible distutils setup script, such as this:

```
# This is setup.py
from distutils.core import setup
import py2exe
setup(console=["wrapper.py"])
```

Now type `python setup.py py2exe` and the result will be put into the `dist` subdirectory. Everything in that subdirectory goes onto your distribution disc or whatever.

If you produce an updated version of **`my_main_module`**, compile it to Python bytecode (`my_main_module.pyc`) and store it into `library.zip`, replacing the previous version (all other files in `library.zip` stay the same). If your updated version requires extra library modules, ensure that these (and any that they depend on) are also present in `library.zip`, again in Python bytecode format. It does not do any harm to put too many modules in `library.zip`, except that it will be larger than necessary. If additional binary modules (e.g. C or C++ modules) are required then this is more complicated; in that case it is better if you had imported them in the initial code when you ran py2exe on Windows.

During my tests, Py2exe stored the files in `library.zip` with no compression. I'm not sure if it will always work so well if you use compression. Most zip utilities can be told to avoid compression; for example, Info-Zip, distributed with many Linux systems, will do this if you add **`-0`** to the command line.

Silas Brown

Welcome to the Wonderful World of Porting!

Paul F. Johnson <editor@accu.org>

Since learning C++ back in 2000, I've been one of the programmers on the open source desktop publishing package called Scribus. This application looks, feels and works well and has been favourably compared to commercial offerings from Quark and Adobe. It is already used around the world for commercial applications – from newspapers and magazines to catalogues, the application gets around (especially as it is being championed by Novell's SuSE Linux distribution).

The application is written in standard C++ and uses the Qt widget set primarily with libart, littlecms and libxml2 also in the mix. There are a few quirks, but nothing much. Due to this and now Apple are using (effectively) BSD, there is already a native Mac OS version of the software. Same cvs tree, but with a couple of small alterations to accommodate the different hardware.

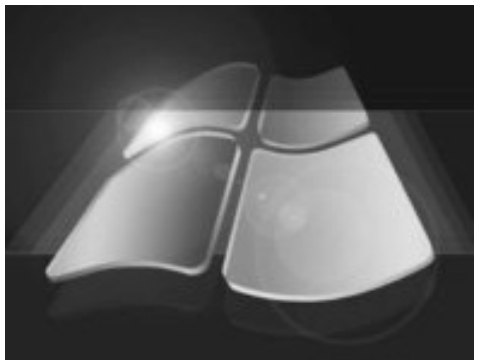
Time rolls on and the programming team have been under a bit of pressure to produce a native Win32 version. There is already a cygwin version available, but it has been a long term aim for there to be a truly native version without the baggage found on the cygwin port.

As I have experience of programming under Win32, I was elected to do the port. Oh well, that should be fun...

Take 1 : VisualStudio.NET

I initially approached the porting process by using Visual Studio .NET under WinXP and with an education licensed copy of Qt.

Um. That was fun. The compile of Qt failed and I had to do that by hand. Not a pleasant experience. I then had to install TortoiseCVS to enable me to obtain the source code. Another pain in the backside process. Finally, they were installed and I could begin.

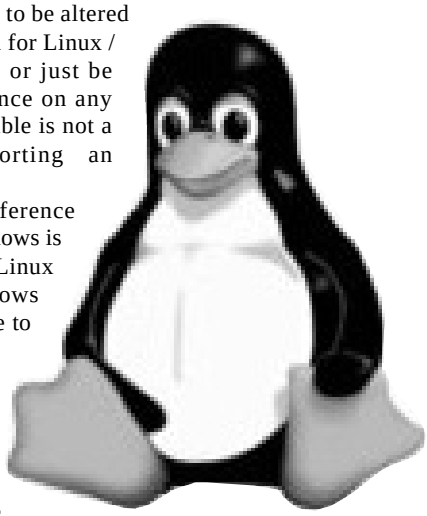


The first problem with any port is to resolve the dependencies. LittleCMS already has a prebuilt version for Windows as do most of the other parts required for the build (including fontconfig, which can be found on the gimp website).

After the dependencies had been resolved (and installed correctly), the next stage was to find the aspects which differ greatly between how the different operating systems work. There are two as far as Scribus is concerned: printing and files. File systems vary greatly from platform to platform, Linux and Windows are no different in this respect.

While to any Linux application developer having self-built applications in `/usr/local` seems fine, to a Win32 user who has only ever used Win32, the logic of having parts of the application in the Windows directory with parts in the Program Files directory would seem fine. Unfortunately, when you've exclusively been based on one platform, this logic vanishes (or at least becomes hazy). Another example, files under Linux are (typically) saved in `$HOME`, under Windows they can be just about anywhere. `$HOME` also doesn't exist under Windows (well, not that I could use!).

The difference in file systems also has another drawback as far as porting is concerned. The Makefile cannot be "just" used. The paths have to be altered to reflect the fact that `/usr/local` would be of little use under Win32. Also anywhere in the code which relied on `$HOME` would also have to be altered with either a conditional for Linux / Mac OS and Windows or just be removed totally. Reliance on any hard-wired system variable is not a good idea when porting an application.



The next major difference between Linux and Windows is in the printing system. Linux uses CUPS while Windows uses its own system. Due to a decision made early on in the development cycle of the original versions of Scribus, the Qt qprinter class was never used for the printing, instead interfacing with CUPS was opted for. At the time, there wasn't a plan for a Win32 version and Mac OS has a port of CUPS already, so the main alternate platform was already covered in those terms.

My idea was to omit the printing system completely until I had time to understand how printing works under Windows.

Right, that's all the parts in place, any environment variables gone and dependencies met. Now to import the source into Visual Studio.

My last experience with the Microsoft Visual series was Visual C++ version 5. Quite a lot has changed, but not for the better from what I could see. The importer was awful. I wasted more time in trying to import a cvs repository than any other aspect of the port so far. In the end, I gave up, wrote a script and restarted Visual Studio. Good. That's the code in. Let's build.

Let's not. The compile failed at the very first file as I'd not remembered to switch off that very annoying bit that includes the non-standard **`<stdafx.h>`** header. Why on earth that is switched on by default is anyone's guess. I find it more of a hindrance than a help.

The actual build went quite nicely until the printing and font handling files came to being compiled. Here, my system gave up with memory fault errors (which is strange as the system has over a gig inside). On closer examination, it wasn't the computer at fault, but the compiler giving some very strange errors. It seems that VS.NET is still having problems with templates, especially with the way I'd been optimising the code. Either there is a leak in the version of the compiler I was using or the compiler was plain daft.

At this point I was asked not to continue porting Scribus. I won't go into the politics, but I was thankful. Not least as it meant that I didn't have to carry on fighting the build system in order to just compile some pretty straightforward source code.

All went quiet at this point for about a year until March 2005. During this time, I was approached to start the port again. To ensure that the same pressure was not bought down on me as before, I attempted to do the port under emulation on my Linux machines.

Emulation is good fun and can be very useful (there is nothing like firing up a ZX-Spectrum emulator and playing Match Day for hours on end on a typically wet summer afternoon or a ARM emulator to ensure some code will actually run on a target platform), but it really does have its limits.

I attempted to get Borland C++ vsn 6 running using CrossOver Office (a more professional version of Wine). The install went fine, but it plain refused to run.

Next was to try MinGW (a Win32 version of GCC). This worked, but was quirky and on the whole, using CrossOver wasn't really proving to be too good. Win4Lin Pro fared little better. While Win4Lin requires a full install of Windows to be installed, it was incredibly slow.

Next time : Onward to a Win32 box and porting.

Paul F. Johnson

Introduction to Tcl/Tk: Part 2

by R.D. Findlay, iCanProgram Inc. <bob@icanprogram.com>

Introduction

In our first article in this series we introduced the Tcl/Tk language and the accompanying programming cycle. We wrote some trivial programs and explored some basic concepts.

In this article we will go further and show that Tcl/Tk can be used to write some very capable applications.

The Menu Bar

Most of you will have used computer software with pull down menus. The familiar File Edit etc. menu bar is across the top of most program windows including the browser you are using to view this page right now. We are going to learn how to create menus in Tcl/Tk .

Before we begin we must discuss terminology.

- a menu bar is the area at the top of a window on which we will hang our menus.
- menu labels are the words in the menu bar where we click our mouse
- the menus are the things which pop down when we click our mouse on a menu label in the menu bar.
- the menu item is the thing we click on once the menu has popped down

Let us illustrate by way of an example.

Using the now familiar text editor create a new file and enter the following partial program:

```
#=======================================
# main - entry point
#=======================================
destroy .menu bar
menu .menu bar

. config -menu .menu bar
```

Note the **.** in front of config is on its own. That is there is a space before the word config.

When you run this program with wish it won't be all that impressive. In fact for you Windows users it will look exactly like nothing happened. The problem here is that different computers (PC Windows, Mac, Linux) handle windows slightly differently.

This is as good a point as any to introduce a couple of constructs which will help make our Tk windows behave better.

Adding a Frame

Using your text editor once again add the following statements just before the menu .menu bar statement above.

```
wm geometry . 200x250+10+10
wm title . my program
```

Don't forget to leave a space before and after the dot in these statements. The purpose of this dot will be come apparent shortly. In the meantime if you run your little program once again you should see a slightly bigger window with a proper my program displayed as the title. Unfortunately we still don't have an obvious menu bar showing up.

To help with this we need to introduce a new widget called a frame. Frames can be thought of as their picture frame counterparts. A picture frame holds a painting or a photograph. Tk frames are kind of like rectangular containers into which we place widgets. At this point we are going to simply use the frame to colour a section of our window thus leaving the menu bar obvious.

You will need to add the following statements after the wm title statement above.

```
destroy .myArea
set f [frame .myArea -borderwidth 5 -
      background blue]
pack $f -side top -expand true -fill both
```

Presto!

When we run our little program now at least Windows users will see a blank area under the title ... our menu bar! You may have to look quite hard to notice it. It's just above the blue area.

Let's examine the statements in more detail starting with the wm and frame statements first.

We notice a couple of new statements beginning with the command wm. Those of you running on a Linux platform might recognize that wm is short for window manager. Tcl/Tk was first developed on a UNIX platform. On the likes of Mac, Windows and Linux, the windowing interface is built in. The term window manager loses its meaning in that case. The designers of Tcl/Tk elected to keep with the name in any event.

For our purposes the wm command allows us to control the wish window. We can specify its size. We can specify where we want it to appear on the desktop. We can specify which title is to appear at the top of the window.

The first of the `wm` commands concerns the location and size of our window:

```
wm geometry . 200x250+10+10
```

Recall that in the first Tk article I mentioned that the window in Tk is known simply as **.** (dot). The explanation lies once again in Tk's Unix roots. Suffice to say that the . immediately after the word geometry in the command is the name of the Tk window. The next argument consists of a set of 4 numbers joined by a x and a couple of +. The first pair of numbers represents the width x height of the window in units of pixels. Our window is to be 200 pixels wide and 250 pixels high. The next pair of numbers represents the position of the window as measured in pixels from the left of the desktop screen and the top of the desktop screen. Our window will be 10 pixels in from the left and 10 pixels down from the top of the desktop screen.

The second `wm` statement is somewhat obvious:

```
wm title . my program
```

It will add the title my program to the top of our wish window when the program starts running.

We have introduced our first example of a frame construct.

```
set f [frame .myArea -borderwidth 5 -
                                background blue]
```

Notice the compound statement consisting of an inner and an outer part. The inner part (inside the [ ]) will be evaluated first and the result will be what the variable f is set to. The inner part sets up a frame widget called myArea. This frame is attached to the top level window called '**.**'. This frame widget has 2 attributes set on it: one for the width of the border and the other for the background colour. This statement is exactly analogous to the button **.hello** statement which we had in previous examples in our first article.

We are then using a pack command to position the outer (parent) frame inside the window.

```
pack $f -side top -expand true -fill both
```

The **-expand true** arguments tell the packer to allow the packing space containing the frame **f** to expand to as needed into the available window cavity. The **-fill** both arguments tell the packer to let the frame expand in both (x and y) directions. Don't worry I find the distinction between these two very difficult to understand ... fortunately I've almost always seen them come together like this. Suffice to say that if you add **-expand true -fill** both to your pack command things will grow to fill whatever space you have for them in your window.

Finally back to discussing menu bars!

The first thing we need to recognize is that the Tcl/Tk menu widget designers are a cruel lot. They decided that menu bars and menus are really just the menu widget with different attributes turned on. I for one find this confusing ... but bear with me it is really not all that bad.

The first statement creates menu bar instance of a menu widget called ... you guessed it ... menu bar.

```
menu .menu bar
```

This is exactly similar to the

```
button .hello
```

statement that we used in the previous article to create an instance of a button widget.

The command menu creates a menu widget. The argument .menubar says call it menubar and attach it in the main window (called .). As with our previous example of the button, nothing will happen until we force the widget to render itself visible. In the previous button example this was accomplished with the pack command.Menu widgets are special and we don't use a pack command to render them.

This is because menus cannot exist unless they are attached directly to other widgets. In our little example What other widgets do we have up? you might ask. The answer is we always have the main window up which is itself a widget. Because menus can only exist when part of a window, the designers of Tcl/Tk decided to manipulate the menu widget as a special window attribute.

Stay with me here ... it does get easier once we are through this part. This does explain the next statement:

```
. config -menu .menubar
```

A little tricky ... but not overly so. The leading dot is the representation for the main (or parent) window widget. The first argument configure (or **`config`** in the shortened form) behaves exactly the same as the configure argument we used to change an attribute of our button widget in the previous example. In this case we are changing the attributes of the main window itself. What we are doing is attaching our menu widget as the menu bar in the window. Just to thoroughly confuse everything the Tcl/Tk designers decided to use **`-menu`** rather than **`-menu bar`** to do this attachment. So much for concise terminology.

There is a saying amongst programmers which goes like this:

If you don't understand a set of statements thoroughly, go ahead and use them anyway as long as they work.

It is probably good advice for the menubar.

The Menu Label

Let's enhance the program with some menu labels now. For this, add these two statements to the bottom of the program above:

```
set File [menu .menu bar.mFile]
.menu bar add cascade -label File -menu .menu
      bar.mFile
```

When you run the program now you will see that it will have added a label called File to our menu bar. If you press this label with your mouse you will see that it tries to pop up a menu but all it succeeds in doing is showing a small dashed line.

Here's where the Tcl/Tk menu widget designers tax our understanding a little more.

The first statement is an example of a compound statement not unlike the format statements we have already used. Remember that the stuff inside the [ ] gets worked out first. The result of that is substituted and the whole statement is then resolved. The [ ] above creates another menu widget we have called **`.menu bar.mFile`**. We are then calling this new widget **`File`** using the **`set`** command. Recall from our discussion above how a menu widget cannot exist on its own ... it must be attached to another widget. This other widget could just as well be another already existing menu widget ... our menu bar for example.

The statement :

```
.menubar add cascade -label File -menu .menu
bar.mFile
```

does the attachment. The **`-label`** attribute is what gives the **`File`** menu label.

It now might make sense why we chose the awful name for the newest menu widget:

```
.menu bar.mFile
```

Think of it as a convenient way of showing a hierarchy. top level window called . -> menu widget called menu bar -> menu widget called **`mFile`**

If you don't understand this ... don't worry just use it and your program will work.

Add the Menu Item

Recall that by using the **`$`** in front of a variable means use the value stored at. In the section above we used to set command to create a variable we called **`File`**.

We can now reference our pull down menu widget as **`$File`**.

Add the following statement to the end of the program.

```
$File add command -label Quit
```

When you run the program it will look much more like a graphical menu. The menu item **`Quit`** should appear. However, when you click your mouse on the item nothing happens ... yet.

If you recall the button example we had the same challenge. We resolved that with a **`-command`** attribute. Menus are no different. The only thing we need to add now is an action command when the particular menu item is selected.

Add the Action Command

The menu widget supports the -command attribute much like the button widget did before. Let's enhance our last statement to look like:

```
$File add command -label Quit -command exit
```

Congratulations! You now have created your very first menu driven Tcl/Tk program. If you click on the Quit menu item now your little program should exit cleanly.

The Text Widget

Another useful Tk widget is the text widget. We will be making extensive use of the text widget as we expand our exploration to ever more complex examples. Besides the fact that we will be using them in our examples, the text widget is also one of the most feature rich and useful of the Tcl/Tk widgets.

In case you are wondering exactly what is a text widget anyway ... think of your text editor. A properly configured text widget is fully capable of being used as a text editor.

The first enhancement we are going to make to our little program is to add the text widget into our frame area.

Using your editor scroll down inside your source file until you reach the **`set f [ frame`** statement. You are going to insert the text widget lines between this statement and the **`pack $f`** statement as illustrated with the code snip below.

```
set f [frame .myArea -borderwidth 5 -
background blue]

#=======================================
# add the text widget
#=======================================
text $f.t -bd 2 -bg white -height 5
pack $f.t

pack $f -side top -expand true -fill both
```

When you run this program with wish a small white box should appear near the top of the blue frame area.

That wasn't that hard was it!

You now added a text widget to your program. Try typing something into the little white box that appears. You should be able to do many of the things you are familiar with in your text editor ... things like inserting, overwriting, highlighting, and deleting text.

It would be a subject for a more advanced discussion to look at all the things you can do with text inside a text widget. Suffice to say we will only be using a tiny fraction of that capability in this exercise.

But first let's not get too far ahead of ourselves and let's look at the new statements that we have introduced.

```
text $f.t -bd 2 -bg white -height 5
```

This statement begins with the widget invocation command text. This creates a text widget. This is very similar to the command you used previously to create a button, menu or a frame.

- The first argument is a little strange. It says we are going to name our new text widget as **`$f.t`**. Recall that the **`$`** has special meaning in the Tcl/Tk language. It means substitute the value of the variable in my place. In this case that variable contains the frame widget itself.
  What **`$f.t`** really means to us is simply that the text widget is going to be added as a child widget to the frame. Remember the hierarchy ... parent->child -> grandchild etc. Simply put, the text widget is inside the frame.
- The next argument is short for border. It simply says leave 2 pixels of space all around the text widget as a border.
- The next argument is short for background. It simply says that we want the background of our text widget to be white.
- The final argument says make this widget 5 lines high. In other words it will be big enough to store 5 lines of text.

There are many more arguments that can be used to set attributes of this text widget but the few we have chosen are sufficient for our purposes.

The other new statement:

```
pack $f.t
```

we have encountered before. For our purposes it says render our widget named **`$f.t`** visible please.

Adding Some Text

Our text widget would not be of much use if we could not add lines of text to it from within our little program.

Using your text editor once again add the following statements just after the **`pack $f`** statement above.

```
#==========================
# add a line of text
#==========================
$f.t insert end abc tag0
```

When you run your newest version of our little program now the letters `abc` appear in the first line of the text box.

Let's examine what the new statement actually does.
This statement is a lot like the configure statement we used in the past with the menu bar. In English it would read something like:

> "to the end of the visible text in the text box called **`$f.t`** insert a new string of text containing **`abc`** and mark that string with an invisible label called **`tag0`**."

To help illustrate how this insert statement works try adding another insert line immediately after this one like.

```
$f.t insert end def tag1
```

Notice how the string is now `abcdef`. The insert simply appended the new text to the end of what was already there.

How do we put `def` on the line below? The easiest way is to add a special character to the end of the first line. This is the same character that would be inserted in your text editor when you hit return . In deference to the days when this key actually did mechanical things on a typewriter it is called the carriage return character. In word processors and in text editors it is invisible on the screen but actually exists in the file. That is why if you hit backspace at the beginning of a line it suddenly gets merged to the end of the previous line. Reason: you just deleted that invisible carriage return character that forced the line separation. In Tcl/Tk this carriage return is represented by **`\n`**.

So if we were to modify our abc line to look like

```
$f.t insert end abc\n tag0
```

we would force the `def` to go to the next line.

Now try positioning your cursor at the beginning of the def line in the text box and hit backspace on you keyboard. Now press return. You should be able to move `def` to end of the `abc` line or back to the second line.

Invisible characters are a huge part of every word processor program you have ever used. They control everything from line spacing to font type and point size.

Adding an Open Menu Item

A program with only a `Quit` menu item is not very useful. Traditionally under the `File` menu there is an `Open` menu item. Let's add the `Open` menu item by editing our source file and inserted the lines required immediately above those for the `Quit` item as per our code snip below:

```
#=======================================
# add the Open menu item
#=======================================
$File add command -label Open

#=======================================
# add the menu item
#=======================================
$File add command -label Quit -command exit
```

Recall in the previous article introduced the very powerful concept of procedures. Procedures are used to add new statements to the Tcl/Tk language. In fact computer languages would be next to useless without that ability.

We are going to add a procedure which will be executed when we select the `Open` button in the menu.

At the top of your source file add the following few statements

```
#==========================
# openFile - entry point
#==========================
proc openFile { } {
set fn openFile

puts stdout [format %s: ding $fn]
} ;# end openFile
```

When you run your program now ... absolutely nothing should change.

"OK ... where is he going with this now?" you ask.

What happens if we enhance our `Open` menu item line to look like

```
$File add command -label Open -command
openFile
```

Now what happens when you run your program and select the `Open` menu item under the `File` menu.

For those of you sceptics ... take a look into your wish console window. You should see the result of that **`puts stdout`** message on that window:

```
openFile: ding
```

Let's move on and introduce our first dialog.

Although you may not know it you have already used dialogs lots of times in your computer already. Simply put, a dialog is the name GUI designers gave to those small windows that pop up when you do things like opening a file, saving a file etc. A typical dialog asks you to supply input and won't let you do anything else with your computer until you have done so ... or hit the `Cancel` button to dismiss the window.

The Open Dialog

The Tcl/Tk language comes with a rich set of predesigned dialogs for doing a whole range of common things. A dialog serves exactly the purpose you might expect ... it provides information and solicits user input to select from amongst that information. One of the more common dialogs is that which is presented to users every time they wish to open a file.

[concluded at foot of next page]

AgileNorth Conference

http://www.agilenorth.org.uk

Call for Papers

The 1st AgileNorth Conference is being held on Tuesday 13th September 2005 at the Harris Park Conference Centre in Preston, Lancashire.
You are invited to submit outlines for either:

- Workshops - lasting for 1.5 hours that encourage audience participation
- Presentations – lasting for 45 minutes

Organisation

For local technical and business staff who wish to learn and share their experiences of becoming and being agile, the AgileNorth Conference is a one day community conference that provides a forum for learning, sharing, improving and building local support networks. Unlike other conferences our conference will encourage participation of those who are new to agile.

Submission

Topics

We are looking for submissions that encourage participation from those who are new to Agile Methods. Topics of interest include, but are not limited to:

Agile outside in - agile development from the perspective of team leaders, managers, customers

- Creating an agile business, starting with agile software development
- Continuous process improvement, transitioning to agile using small steps
- The economics of agile software development, e.g. ROI, short-term vs. long term, bookkeeping methods
- Selling agile software development to customers
- Working with customers, planning and defining releases

Agile inside out - agile development from the perspective of developers, team leaders, project managers

- Methods: XP, Scrum, DSDM, Crystal etc
  - Estimating
  - Testing
- Stepping Stones: Key practices on the road to an Agile Process
- Techniques: Estimating, Refactoring, Test-Driven Development…
- Technology: OOP, J2EE, .NET, databases...
- The developer role: pair programming, collective code ownership, working with testers etc.
- Team Skills: Coaching, Facilitation, Change Process, Group Dynamics
- Introducing Agile Methods openly: Confessing to your manager you're doing XP
- Introducing Agile Methods by Stealth: "Guerilla XP"

Agile lessons – case studies, war stories, share your eXperiences.

- What has worked?
- What didn't work?

Submission Details

For a session proposal, the following information is required (you may provide more information if you feel that's appropriate):

- Title
- Organiser(s), including a short biography and contact information (e-mail address and phone number) for the principal organiser.
- Session duration (45 or 90 minutes)
- Session type: simulation; game; workshop; interactive presentation; case study; interactive demonstration; goldfish bowl; panel discussion; other .
- Session description: objectives, contents, process, timetable, benefits of attending, the way in which the audience is involved
- Intended audience (e.g. developers, managers, testers) and their experience level (e.g. apprentice, journeyman, master)
- What does the organiser expect to learn from running his session?

Format for Submission

The submissions should be in the form of a MS PowerPoint or OpenOffice presentation or as an MS Word or OpenOffice document outlining your topic and its target audience. You may wish to include areas such as whether it is reusable by others, any limitations and future development ideas.

[continued from previous page]

Inside the procedure **`openFile`** add the following statement (just after the **`puts stdout ding`** line is as good as any place).

```
set myFile [tk_getOpenFile]
```

Now run the program.

This is really starting to look like a real program. In fact all the real programs you have been using on your computer all along are nothing more than more complicated versions of this little program you have written.

To use the writing analogy: the real programs are full novels ... you have learned how to write a paragraph. All that is required to produce a full novel is to connect lots of paragraphs together into a cohesive story.

The Save Dialog

The next enhancement we are going to make to our little program is to add the call to the save dialog, which mirrors our call to the open dialog we introduced above. Using your editor modify your source file to add the **`saveFile`** procedure as shown below.

```
proc saveFile { } {
set fn saveFile
set myFile [tk_getSaveFile]
puts stdout [format %s:myFile=<%s> $fn
$myFile]
} ;#end saveFile
```

You'll also want to add the Save option under the File menu as indicated below.

```
#=======================================
# add the Open menu item
#=======================================
$File add command -label Open -command
      openFile
#=======================================
# add the Save menu item
#=======================================
$File add command -label Save -command
      saveFile
#=======================================
# add the menu item
#=======================================
$File add command -label Quit -command exit
```

When you run this program with wish and select the Save menu option you should see a dialog appear asking you to enter the file name you wish to save things as. Try typing some imaginary file name and clicking the save button.

For those of you who were paying attention to what lines you modified above you would have expected that you should see a line of output appear on your wish console window. That line should look something like:

```
saveFile:myFile=<c:/myprog/bob.tcl>
```

The string of words in the **<>** above represent what is known in programming circles as a full file name. Notice that this full file name tells you exactly where to find your file. Each operating system (Windows, Mac or Linux) will have slightly different full file names.

You have used a hierarchy in your computer travels often without really knowing it exists. The graphical representation of the file system as folders and files shields this from you. But underneath that graphical veneer almost every computer file system behaves in much the same way. You can almost always access a given file by simply specifying the full path to that file.

Next time, we'll see if we can use our new found knowledge to actually access a file on our hard drive from within our Tcl/Tk program.

R. D. Findlay

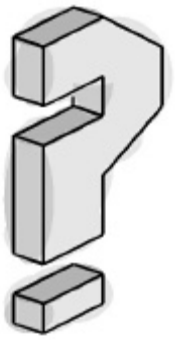
Professionalism in Programming #32

An Exercise for the Reader

by Pete Goodliffe <pete@cthree.org>

> Imagination grows by exercise, and contrary to common belief, is more powerful in the mature than in the young.
>
> W. Somerset Maugham

It's time to put on our walking shoes again, and trudge back through the annals of history to revisit some of the themes we've encountered on our journey through the professionalism series. We've investigated an eclectic series of themes over the years, all to do with the day-to-day programming experience. Throughout it all I've been trying to draw out how 'professional' programmers react, behave, and work. This is the second such nostalgia-inducing column I've written and, as before, this is an opportunity to asses exactly how 'professional' you are.



In this article I present a series of thought provoking questions based on some topics that I've covered in past issues[1]. Mull them over and see what you think about each question; make the effort to come up with an answer. The motivation is simple: to improve your skill you have to get involved – passively soaking up information doesn't do you much good. Instead, you've got to invest some effort. I make no apologies for writing such a difficult article this issue – *it's for your own good!*

The plan is simple: I pose the questions first, give you some time to think about them (go fetch a coffee, sit down peacefully, and think it all through). Then I provide some discussion on the questions. This discussion is not a definitive set of 'answers' (most of these questions have no single simple answer), they're more my immediate thoughts and responses. Compare your answers with mine.

Questions

First, here are the questions. We'll look at two different topics here. Spend a while considering your answer to each one before you move on to the following section.

The Need for Speed (C Vu 16.2, February 2004)

This article series focused on software optimisation, explained the reasons not to optimise and also the reasons you should. We looked at general classes of optimisation, and at some quite specific code-level optimisations.

1. Why is optimisation an issue? Why *don't* we write efficient code? What stops us from using high performance algorithms in the first place?
2. A `List` data type is implemented using an array. What is the worst case algorithmic complexity of each of the following `List` methods?
   - The constructor.
   - `append` – places a new item on the end of the list.
   - `insert` – slides a new item in between two existing list items, at a given position.
   - `isEmpty` – returns `true` if the list contains no items.
   - `contains` – returns `true` if the list contains a specified item.
   - `get` – returns the item with a given index.
3. How important (honestly) is code performance in your current project? What is the motivator for this performance requirement?
4. In your last optimisation attempt:
   - Did you use a profiler?
     - If *yes*: how much improvement did you measure
     - If *no*: how did you know whether you made any kind of improvement?
   - Did you test the code still worked after optimising
     - If *yes*: How thoroughly did you test?
     - If *no*: Why not? How could you be sure the code still worked properly for all cases?
5. How well specified are your program's performance requirements? Do you have a concrete plan to test that you meet these criteria

Software Testing (C Vu 13.4, October 2001)

This article looked at the thorny topic of testing the software that we write. We looked at when, how and why we test software.

1. Write a test harness for the following piece of code: a function to calculate the greatest common divisor of two integers. Make it as exhaustive as you can. How many individual test cases have you included?

   - How many of these passed?
   - How many failed?
   - Using these tests, identify any faults and repair the code.

```
int greatest_common_divisor(int low, int high)
  {
    if (low  high)
    {
      int tmp = high;
      high    = low;
      low     = tmp;
    }

    int gcd = 0;
    for (int div = low; div  0; –div)
    {
      if ((low % div == 0)
          (high % div == 0))
        if (gcd  div)
          gcd = div;
    }
  return gcd;
  }
```

2. Should you test the all of the *test code* that you write?
3. How does a programmer's testing differ from a QA department member's testing?
4. For what percentage of your code do you write tests? Are you happy with this? What sort of testing do you give the remaining code? Is this adequate? What will you do about it?
5. What's your usual response to finding an error in your code?

Discussion and Answers

Lazy readers will have jumped here already. Please do spend some time considering your answer to each question first. It will be interesting to compare your response with mine. Do you disagree with anything? Do you agree? Let me know.

The Need for Speed

> 1. Why is optimisation an issue? Why *don't* we write efficient code? What stops us from using high performance algorithms in the first place?

There are many perfectly valid reasons for not writing 'optimised' code on the first attempt:

- You don't know the final pattern of usage. With no Real World test data, how can you choose the code design that will work best?
- It's hard enough to get the programming *working*, let alone *fast*. To prove it's feasible we choose designs that are easy to implement so prototypes get finished quickly.
- 'High performance' algorithms can be more complex and daunting to implement. Programmers naturally shy away from them, since it's an area where faults can be easily introduced.

An obvious mistake is to think that the taken to run something is proportional to the amount of effort spent writing it[2]. You might have written some file parsing code in hours; it will always takes ages to execute, because disks are slow. The complex code you spent half a week getting right may only consume a few hundred processor cycles. The efficiency of a piece of code, or the amount of time you need to spend optimising it, bears no relation to the amount of time you spent writing it.

1 You might want to go back and revisit the specific articles before ploughing into these questions.

2 That looks stupid when you see it written down, but it's a very easy trap to fall into at the codeface.

2. A **List** data type is implemented using an array. What is the worst case algorithmic complexity of each of the following **List** methods?

- The constructor is O(1) since it only needs to create an array; the list is initially empty. However, it's worth considering that the size of this array will affect the complexity of the constructor — most languages create arrays fully populated with objects, even if you don't plan to use them yet. If the constructors for these objects are non-trivial, then the **List** constructor will take some time to execute.
- The array size might not be fixed — the constructor could take a parameter to determine this size (effectively setting the maximum list size possible). The method then becomes O(n).
- The **append** operation is O(1). It simply has to write an array entry and update the list size.
- **insert** is O(n). In the worst case you will be asked to insert an element at the very beginning of the list. This requires all the elements in the array to be shuffled up one place before writing the first element. The more items in the **List**, the longer this will take.
- Unless you have a ridiculously bad implementation, **isEmpty** is O(1). The list size will be known, so the return value is a single calculation based on this number.
- **contains** is O(n), presuming the list contents are unordered. In the worst case you will be asked to look for an item that doesn't exist, and will have to traverse every single list item.
- **get** is O(1) thanks to the array implementation. Indexing an array is a constant time operation. If **List** had been implemented as a *linked list*, then this would have been an O(n) operation.

3. How important (honestly) is code performance in your current project? What is the motivator for this performance requirement?

The performance requirements should not be arbitrarily chosen. They should be justified; not just a time limit pulled out of thin air. Every performance requirement is important — there are no specifications that don't matter. How much concern a particular requirement generates depends on how hard it is to meet. Whether it's hard or not, you still have to come up with a design that satisfies it.

4. In your last optimisation attempt:

- Did you use a profiler?
  - If *yes*: how much improvement did you measure
  - If *no*: how did you know whether you made any kind of improvement?
- Did you test the code still worked after optimising
  - If *yes*: How thoroughly did you test?
  - If *no*: Why not? How could you be sure the code still worked properly for all cases?

Only the most dramatic performance improvements can be detected without a profiler, or some other good timing tests. Human perception is easily fooled — when you've just slaved to speed up the program, it will always *appear* faster to you.

Test performance improvements carefully, and discard those that are not worthwhile. It's better to have clear code than a miniscule speed increase and unmaintainable logic.

5. How well specified are your program's performance requirements? Do you have a concrete plan to test that you meet these criteria?

Without a clear specification, no one can really complain that your program isn't fast enough!

Software Testing

1. Write a test harness for **greatest_common_divisor**. Make it as exhaustive as you can. How many individual test cases have you included?

- How many of these passed?
- How many failed?
- Using these tests, identify any faults and repair the code.

There are a large number of tests you should run, even though there are very few 'invalid' input combinations. Thinking of invalid inputs first: test for *zero*. It may or may not be an invalid value (we've seen no spec, so we can't tell), but you'd expect the code to cope reasonably with it.

Next, write tests considering combinations of 'usual' inputs (say of 1, 10, and 100 in all orders). Then try numbers with no common multiple like 733 and 449. Test for some very large numbers, and also for some negative numbers.

How do you write these test cases? Use a good test framework, or put them all in a single test function. For each test, don't programmatically calculate what the correct output value should be[3], just check against a known constant value. Keep your test code as simple as possible:

```
assert(greatest_common_divisor(10,  100) == 10);
assert(greatest_common_divisor(100, 10)  == 10);
assert(greatest_common_divisor(733, 449) == 0);
... more tests ...
```

There are a surprisingly large number of tests for this simple function. You could argue that for such a small piece of code it's easier to inspect, review, and prove correctness rather than laboriously create a set of tests. This seems like a valid argument. But what if later on someone makes modifications? Without the tests you'd have to carefully re-inspect and re-validate the code, an easy task to overlook.

Did you find the mistake in **greatest_common_divisor**? There's a clue coming up. If you don't want the puzzle spoiled then look away now. Try feeding it *a negative argument*. This is a more robust (and more efficient) version written in C++:

```
int greatest_common_divisor(int a, int b)
{
  a = std::abs(a);
  b = std::abs(b);
  for (int div = std::min(a,b); div  0; –div)
    {
      if ((a % div == 0)  (b % div == 0))
          return div;
  }
  return 0;
}
```

2. Should you test the all of the *test code* that you write?

If you think about this for long enough it will give you a headache. You can't keep testing test code how can you be sure the test code for your test code's test code is correct? The trick is to keep tests *as simple as possible*. This way, the most likely testing errors will be missing important test cases, not problems with the actual lines of test code.

3. How does a programmer's testing differ from a QA department member's testing?

Testers are more concerned with the black box style of testing, and usually only perform product testing. It's rare to have testers working at the code level, because most products are executable software; there are comparatively few code library vendors.Programmers are more concerned with white-box tests, making sure their masterful creations work as they planned.

The secret aim of any programmer writing tests is to prove that their code works, not to find cases where it doesn't! I can easily write a load of tests to show how perfect my code is by deliberately avoiding all the bits I know are dodgy. This is a good argument for getting someone other than the original programmer to create test harnesses.

4. For what percentage of your code do you write tests? Are you happy with this? What sort of testing do you give the remaining code? Is this adequate? What will you do about it?

Don't feel obliged to a write test harness for every scrap of code. Use your head. Inspections are fine for small functions. But you must be sure that you are performing the adequate and appropriate testing for which you are responsible, not just skipping an unpleasant task.

[concluded at foot of next page]

3 This would open the door to more coding errors - image the pain of bugs in the test code!

Are Certificates Worth the Paper they are Written On?

Alan Lenton <alan@ibgames.com>

I often get asked about certification for programmers, and whether it makes any difference in the job market. The answer, predictably, is both yes and no! It really depends on what sort of job you are looking for, and how the recruitment procedure of the firms you are interested in applying to works.

Let me start by asking a slightly different question. Does a programming certification prove you can program?

Not necessarily. It certainly proves that you can pass programming tests. Programming itself, is, however, a different matter. The problem is that programming is something you need to constantly practise. Let me give you a couple of analogies.

My first analogy is that learning to program is a bit like learning to drive a car. You spend some time with an instructor, and he or she teaches you the formal elements you need. But I don't think I've ever known anyone who passed their driving test with what an instructor taught them. To really learn to drive you then have to get a friend to sit with you while you practise until the driving becomes second nature. When you reach that level you go back to the instructor who teaches you how to pass the test.

In a way computing is very similar. Your teacher can teach you the basics, but to go beyond the basics you have to practise. The problem from the point of view of a potential employer is that people usually get the certification at the end of the teaching period - they know enough to pass the test, but are not fluent.

There is also one way in which programming is very different from driving a car. Even if you stop driving, you never really forget how to do it, and you can get back up to speed pretty rapidly (the same goes for riding a bicycle or roller skating). This is not the case with programming. Programming skills are 'use it or lose it' skills par excellence.

That's why the second analogy I would make for programming is with music making. It's not enough merely to learn to play a musical instrument, you have to constantly practise to keep your skills.

Programming is much the same, if you don't keep programming you will lose your programming skill and fluency. Even worse, you will fall behind in what is a rapidly changing skill set. It's not just that you get up to speed, it's that you are learning new techniques as you go along, and the longer your experience, the better you are as a programmer.

Why does experience make so much difference? It has to do with the nature of the problem solving needed for programming. Virtually any problem that requires professional programming has thousands, if not millions, of potential solutions. Not only that, but all the solutions that will work have different trade-offs in development time, ease of use, resources required, maintenance and a hundred other things. The difference between an experienced programmer and a novice is that the experienced programmer can almost immediately partition that solution space into a small number of feasible solutions and a large number of impractical solutions. Even a well trained novice cannot do that, until they have experience to inform their judgements. To a novice all the solutions are more or less equal.

This is why certification doesn't really tell you, as an employer, much about how good a potential employee is going to be. If you get interviewed by a professional, your certification won't cut much ice - they will be looking for answers to their question that indicate a depth of knowledge which is entirely different from the skill that the certificates test. (Incidentally, I suspect that most of the really good programmers I know would probably fail the tests!) Your interviewer is also going to be looking for something else - an ability to fit in with a team. You can be the most brilliant programmer in the world, but if you can't work with other people, you're sunk. Social skills matter just as much.

But I did originally say that part of the answer to the question I started with - are certificates worth the paper that they are written on - was yes.

The reason for this has little or nothing to do with your programming ability. It has everything to do with the structure of modern business. The days when the people responsible for the work hired and fired the people who did the work are long since gone in companies that employ more than a handful of people. The last half century has seen the rise of the Human Resources (HR) department, and that has changed the rules.

HR departments evolved from the original personnel departments, which had a fairly minor role. As the volume of legislation relating to the workplace and employees grew, they assumed more and more functions and eventually took control of hiring and firing to ensure that such activities complied with the relevant legislation. This meant that you were no longer hired by the person you would work for - the person who had the technical knowledge of your work. Instead you were be hired by people whose profession was understanding employment law.

This had an immediate consequence - the HR department needed some sort of validation to prove that you were technically qualified, since they had no means of judging for themselves. This set the stage for demands first of all for academic qualifications (Computer Science degrees in this case), and then certificates to say that you were qualified to do the job.

Of course, it's relatively unusual to see a 'pure' model of this process. Usually the department has some input into the job advert - although I did once see an advert for a programmer with five years' experience of C++, just three years after the first C++ compiler became available! The department also usually handles the interviews.

The important point, though, is that the interviewees are selected by HR, and HR will automatically exclude all those who are not qualified. And to an HR department, if you don't have a certification, you aren't qualified, even if you've been programming for years.

Finally, an additional push to the whole business of certification is that training, publishing and testing for these certificates has become a very lucrative business, which, taken with the need felt by HR departments for the certificates, has generated a whole new industry.

So there you have it.

The answer is that you probably do need certification, although it tells your prospective employer nothing about your skills - but what it does tell him or her is that you understand the rules of the game!

Have fun on the net!

Alan Lenton

[continued from previous page]

Ask yourself this: how many errors have bitten you recently which could have been prevented with a good set of tests? Make sure you do something about it.

5. What's your usual response to finding an error in your code?

There are several possible reactions:

- disgust and disappointment,
- the urge to blame someone else,
- happiness, if not downright *excitement*, and even
- pretending you didn't find it, ignoring it, and hoping it will go away (as if that's likely)

Some of those are so plainly wrong that I'll assume you can rise above them.

Does it seem a little daft to suggest you might be *happy* to find a fault? Surely that's the reasonable reaction for a quality-conscious engineer, as it's far better to find faults during development than for a user to find them in the field.

Your level of excitement will depend on where in the development lifecycle the fault is found. Discovering a show-stopping bug the day before release won't make anyone smile.

Pete Goodliffe

Patterns in C - Part 3: Strategy

By Adam Petersen <adampetersen75@yahoo.se>

Identifying and exploiting commonality is fundamental to software design. By encapsulating and re-using common functionality, the quality of the design rises above code duplication and dreaded anti-patterns like copy-paste. This part of the series will investigate a design pattern that adds flexibility to common software entities by letting clients customize and extend them without modifying existing code.

Control Coupled Customers

To attract and keep customers, many companies offer some kind of bonus system. In the example used in this article, a customer is placed in one of three categories:

- *Bronze customers*: Get 2 % reduction on every item bought.
- *Silver customers*: Get 5 % reduction on every item bought.
- *Gold customers*: Get 10 % off on all items and free shipping.

A simple and straightforward way to implement a price calculation based upon this bonus system is through conditional logic.

Listing 1: Solution Using Conditional Logic

```
typedef enum
{
   bronzeCustomer,
   silverCustomer,
   goldCustomer
} CustomerCategory;

double calculatePrice(
    CustomerCategory category,
    double totalAmount,
    double shipping)
{
   double price = 0;
    /* Calculate the total price depending on
       customer category. */
    switch(category) {
     case bronzeCustomer:
          price = totalAmount * 0.98 +
              shipping;
          break;
     case silverCustomer:
          price = totalAmount * 0.95 +
              shipping;
          break;
     case goldCustomer:
          /* Free shipping for gold
             customers.*/
          price = totalAmount * 0.90;
          break;
     default: onError("Unsupported category");
     break;
    }
    return price;
}
```

Before further inspection of the design, I would like to point out that representing currency as `double` may lead to marginally inaccurate results. Carelessly handled, they may turn out to be fatal to, for example, a banking system.

Further, security issues like a possible overflow should never be ignored in business code. However, such issues have been deliberately left out of the example in order not to lose the focus on the problem in the scope of this article.

Returning to the code, even in this simplified example, it is possible to identify three serious design problems with the approach, that wouldn't stand up well in the real-world:

1. *Conditional logic*. The solution above basically switches on a type code, leading to the risk of introducing a maintenance challenge. For example, the above mentioned security issues have to be addressed on more than one branch leading to potentially complicated, nested conditional logic.

2. *Coupling*. The client of the function above passes a flag (category) used to control the inner logic of the function. Page-Jones defines this kind of coupling as "control coupling" and he concludes that control coupling itself isn't the problem, but that it "*often indicates the presence of other, more gruesome, design ills*" [5]. One such design ill is the loss of encapsulation; the client of the function above knows about its internals. That is, the knowledge of the different customer categories is spread among several modules.

3. *It doesn't scale*. As a consequence of the design ills identified above, the solution has a scalability problem. In case the customer categories are extended, the inflexibility of the design forces modification to the function above. Modifying existing code is often an error-prone activity.

The potential problems listed above, may be derived from the failure of adhering to one, important design principle.

The Open-Closed Principle

Although mainly seen in the context of object oriented literature, the open-closed principle defines properties attractive in the context of C too. The principle is summarized as "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" [1].

According to the open-closed principle, extending the behaviour of an ideal module is achieved by adding code instead of changing the existing source. Following this principle not only minimizes the risk of introducing bugs in existing, tested code but also typically raises the quality of the design by introducing loose coupling. Unfortunately, it is virtually impossible to design a module in such a way that it is closed against all kinds of changes. Even trying to design software in such a way would overcomplicate the design far beyond suitability. Identifying the modules to close, and the changes to close them against, requires experience and a good understanding of the problem domain.

In the example used in this article, it would be suitable to close the customer module against changes to the customer categories. Identifying a pattern that lets us redesign the code above in this respect seems like an attractive idea.

STRATEGY

The design pattern STRATEGY provides a way to follow and reap the benefits of the open-closed principle. *Design Patterns* [2] defines the intent of STRATEGY as "*Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it*". Related to the discussion above, the price calculations in the different customer categories are that "*family of algorithms*". By applying the STRATEGY pattern, each one of them gets fully encapsulated rendering the structure in Figure 1.

The context, in this example the `Customer`, does not have any knowledge about the concrete categories anymore; the concrete strategy is typically assigned to the context by the application. All price calculations are delegated to the assigned strategy.

Using this pattern, the interface for calculating price adheres to the open closed principle; it is possible to add or remove categories without modifying the existing calculation algorithms or the `Customer` itself.

Implementation Mechanism

When implementing the STRATEGY pattern in C, without language support for polymorphism and inheritance, an alternative to the object oriented features has to be found for the abstraction.

This problem is almost identical to the one faced when implementing the STATE pattern [4], indicating that the same mechanism may be used, namely pointers to functions. The possibility to specify pointers to functions proves to be an efficient technique for implementing dynamic behaviour.

In a C-implementation, the basic structure in the diagram remains, but the interface is expressed as a declaration of a pointer to function.

Listing 2: Strategy interface in `CustomerStrategy.h`

```
typedef double (*CustomerPriceStrategy)(
    double amount, double shipping);
```

The different strategies are realized as functions following the signature specified by the interface. The privileges of each customer category, the concept that varies, are now encapsulated within their concrete strategy. Depending on the complexity and the cohesion of the concrete strategies, the source code may be organized as either one strategy per compilation unit or by gathering all supported strategies in a single compilation unit. For the simple strategies used in this example, the latter approach has been chosen.

Listing 3: Interface to the Concrete Customer Categories in `CustomerCategories.h`

```
double bronzePriceStrategy(double amount,
    double shipping);

double silverPriceStrategy(double amount,
    double shipping);

double goldPriceStrategy(double amount,
    double shipping);
```

Listing 4: Implementation of the Concrete Customer Strategies in `CustomerCategories.c`

```
/* In production code, all input should be
   validated and the calculations secured upon
   entry of each function.
*/
double bronzePriceStrategy(double amount,
    double shipping)
{
   return amount * 0.98 + shipping;
}

double silverPriceStrategy(double amount,
    double shipping)
{
   return amount * 0.95 + shipping;
}

double goldPriceStrategy(double amount,
    double shipping)
{
   /* Free shipping for gold customers. */
   return amount * 0.90;
}
```

Using the strategies, the context code now delegates the calculation to the strategy associated with the customer.

Listing 5: `Customer.c`

```
#include "CustomerStrategy.h"
/* Other include files omitted... */
struct Customer
{
   const char* name;
   Address address;
   List orders;
   /* Bind the strategy to Customer. */
   CustomerPriceStrategy priceStrategy;
};

void placeOrder(struct Customer* customer,
    const Order* order)
{
   double totalAmount = customer-
>priceStrategy
(order->amount, order->shipping);
   /* More code to process the order… */
}
```

The code above solves the detected design ills. As the customer now only depends upon the strategy interface, categories can be added or removed without changing the code of the customer and without the risk of introducing bugs into existing strategies. The remaining open issue with the implementation is to specify how a certain strategy gets bound to the customer.

Binding the Strategy

The strategy may be supplied by the client of the customer and bound to the customer during its creation, or an initial strategy may be chosen by the customer itself. The former alternative is clearly more flexible as it avoids the need for the customer to depend upon a concrete strategy. The code below illustrates this approach, using a customer implemented as a FIRST-CLASS ADT [3].

Listing 6: Binding Strategy Upon Creation, `Customer.c`

```
CustomerPtr createCustomer(const char* name,
    const Address* address,

CustomerPriceStrategy priceStrategy)
{
   CustomerPtr customer = malloc(sizeof *
       customer);

   if(NULL != customer){
      /* Bind the initial strategy supplied by
         the client. */
      customer->priceStrategy = priceStrategy;

      /* Initialize the other attributes of
         the customer here. */
   }

   return customer;
}
```

Listing 7: Client Code Specifying the Binding

```
#include "Customer.h"
#include "CustomerCategories.h"

static CustomerPtr createBronzeCustomer(const
    char* name, const Address* address)
{
  return createCustomer(name, address,
      bronzePriceStrategy);
}
```

Depending on the problem at hand, a context may be re-bound to another strategy. For example, as a customer gets promoted to the silver category, that customer should get associated with the `silverPriceStrategy`. Using the technique of pointers to functions, a run-time change of strategy simply implies pointing to another function.

Listing 8: Rebinding a Strategy, `Customer.c`

```
void changePriceCategory(CustomerPtr customer,
    CustomerPriceStrategy newPriceStrategy)
{
   assert(NULL != customer);
   customer->priceStrategy = newPriceStrategy;
}
```

Yet another alternative is to avoid the binding altogether and let the client pass the different strategies to the context in each function invocation. This alternative may be suitable in case the context doesn't have any state memory. However, for our example, which uses first-class objects, the opposite is true and the natural abstraction is to associate the customer with a price-strategy as illustrated above.

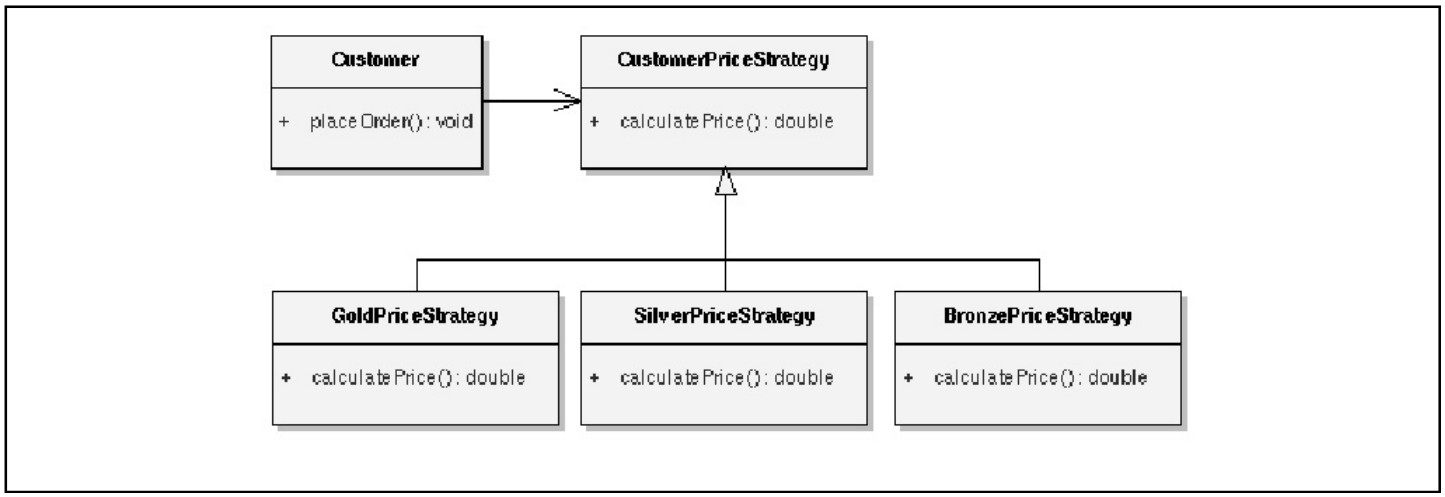


Figure 1: STRATEGY Pattern Structure

Comparison of STRATEGY and STATE

When discussing the STRATEGY pattern, its relationship with the pattern preceding it in the *Design Patterns* [2] book deserves special mention. The design patterns STATE [2] [4] and STRATEGY are closely related. Robert C. Martin puts it this way: "*all instances of the State pattern are also instances of the Strategy pattern, but not all instances of Strategy are State*" [1].

This observation leads us to a recommendation by John Vlissides, co-author of *Design Patterns*, stating that "*Let the intents of the patterns be your guide to their differences and not the class structures that implement them*" [6]. And indeed, even though STATE and STRATEGY have a similar structure and use similar mechanisms, they differ in their intent.

The STATE pattern focuses upon managing well-defined transitions between discrete states, whereas the primary purpose with STRATEGY is to vary the implementation of an algorithm.

Related to the example used in this article, had the categories been based on the sum of money spent by a customer, STATE would have been a better choice; the categories would basically illustrate the lifecycle of a customer, as the transitions between categories depend upon the history of the customer object itself. The STATE pattern may be used to implement this behaviour as a finite state machine.

On the other hand, in case the placement in a certain category is the result of a membership fee, STRATEGY is a better abstraction. It is still possible for a customer to wander between different categories, but a transition to a new category doesn't depend upon the history of that customer object.

Although not to be taken as a universal truth, a further observation relates to the usage of these two patterns and their relationships towards the client. STATE tends to be an internal concern of the context and the existence of STATE is usually encapsulated from the client. Contrasting this with STRATEGY, the client usually binds a concrete strategy to its context.

Consequences

The main consequences of applying the STRATEGY pattern are:

1. The benefits of the open-closed principle.
   The design pattern STRATEGY offers great flexibility in that it allows clients to change and control the behavior of an existing module by implementing their own, concrete strategies. Thus, new behavior is supported by adding new code instead of modifying existing code.

2. Reduces complex, conditional logic.
   Complex interactions may lead to monolithic modules littered with conditional logic, potentially in the form of control coupling. Such code tends to be hard to maintain and extend. By encapsulating each branch of the conditionals in a strategy, the STRATEGY pattern eliminates conditional logic.

3. Allows different versions of the same algorithm.
   The non-functional requirements on a module may typically vary depending on its usage. The typical trade-off between an algorithm optimized for speed versus one optimized with respect to memory usage is classical. Using the Strategy pattern, the choice of trade-offs may be delegated to the user ("Strategies can provide different implementations of the same behavior" [2]).

4. An extra level of indirection.
   The main issue with this consequence arises as data from the context has to be obtained in a strategy. As all functions used as strategies have to follow the same signature, simply adding potentially unrelated parameters lowers the cohesion. Implementing the context as a FIRST-CLASS ADT [3] may solve this problem as it reduces the parameters to a single handle.

   With the FIRST-CLASS ADT approach, the knowledge about the data of interest is encapsulated within each strategy and obtained through the handle to the FIRST-CLASS ADT. A careful design should strive to keep the module implemented as a FIRST-CLASS ADT highly cohesive and avoid having it decay into a repository of unrelated data needed by different strategies (such a design ill typically indicates that the wrong abstraction has been chosen).

   Similarly, in case the flexibility of the STRATEGY pattern isn't needed and the problem may be solved by conditional logic that is easy to follow, the latter is probably a better choice.

Example of Use

STRATEGY may prove useful for specifying policies in framework design. Using STRATEGY, clients may parameterize the implementation.

For example, error-handling is typically delegated to the application. Such a design may be realized by letting the client provide a strategy to be invoked upon an error. By those means, the error may be handled in an application specific manner, which may stretch between simply ignoring the error to logging it or even reboot the system.

Next Time

The next article will look further into dependency management and continue to build on the open-closed principle by introducing a C implementation of the OBSERVER [2] pattern.

Adam Petersen

References

1. Robert C. Martin, "*Agile Software Development*", Prentice Hall
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J, "*Design Patterns*", Addison-Wesley
3. Adam Petersen, "Patterns in C, part 1", C Vu 17.1
4. Adam Petersen, "Patterns in C, part 2: STATE", C Vu 17.2
5. Meilir Page-Jones, "*The Practical Guide to Structured Systems Design*", Prentice Hall
6. John Vlissides, "*Pattern Hatching*", Addison-Wesley

Acknowledgements

Many thanks to Magnus Adamsson, Tord Andersson, Drago Krznaric and André Saitzkoff for their feedback.

New Container Classes in Qt 4

Jasmin Blanchette <jasmin@trolltech.com>

The next major version of Qt, version 4.0, is expected to be released in June. Qt 4.0 introduces many new features and many APIs that have existed ever since version 1.0 have been redesigned.

In this article, we will review Qt 4's new set of template container classes.

A Bit of History

Qt 1.0 was released in 1996 with its own container classes. These classes were used internally by Qt and were part of Qt's public API as an offer to Qt application developers. They existed in two versions: A macro-based implementation and a template-based implementation.

- Internally, Qt used the macro-based implementation internally, because many compilers at the time didn't support templates.
- The template-based implementation was available to application writers who were using recent enough compilers.

Having been developed before STL became part of the C++ Draft Standard, the Qt 1 container classes had their own original design. First, they were pointer-based, meaning that they stored pointers to objects instead of values. For example, a `QList<int>` was conceptually equivalent to a `std::list<int *>`.

In a GUI toolkit, most classes are not copiable (for example, `QObject`, `QWidget` and their subclasses), so it made sense to store pointer to these. When storing "value" types such as `int` and `QString`, the containers had an "auto-delete" option that you could turn on to give ownership to your objects to the container.

With Qt 2, the need for value-based collections was addressed by a new set of container classes inspired by the STL containers. Qt 2 introduced `QValueList<T>`, `QValueStack<T>` and `QMap<K, T>`, with an API that resembled STL a bit, with some adaptations to Qt's naming conventions (for example, `push_back()` is called `append()` in Qt). The iterators met the STL axioms for iterators, ensuring that you could use them in STL algorithms. Qt 2 also provided a few algorithms of its own, such as `qHeapSort()`, for the benefit of application developers who couldn't rely on the presence of STL.

With Qt 3, we added `QValueVector<T>` and we renamed the old-style, Qt 1 container classes to include "Ptr" in their name. `QList<T>` became `QPtrList<T>`, `QVector<T>` became `QPtrVector<T>`, `QQueue<T>` became `QPtrQueue<T>`, and `QStack<T>` became `QPtrStack<T>`. This step was taken to reduce confusion and to keep beginners away from these classes.

Implicit Data Sharing

Unlike existing STL implementations, both sets of Qt containers (and in fact most Qt tool classes) use an optimization called implicit data sharing to make copying containers faster.

Implicit sharing takes place behind the scenes in Qt. When you take a copy of an object, only a pointer to the data is copied; the actual data is shared by the two objects (the original and the copy). When either object is modified, the object first takes a copy of the data, to ensure that the modification will apply only to this object, not to other objects that shared the data. This is why this optimization is sometimes called "copy on write".

Since Qt's container classes are all implicitly shared, you can pass them around to functions as you will. Because of this, implicit sharing encourages a programming style where containers (and other Qt classes such as `QString` and `QImage`) are returned by value:

```
QMap<QString, int> createCityMap()
{
  QMap<QString, int> map;
  map["Tokyo"] = 28025000;
  map["Mexico City"] = 18131000;
  ...
  return map;
}
```

The call to the function looks like this:

```
QMap<QString, int> map = createCityMap();
```

Without implicit sharing, you would be tempted to pass the map as a non-const reference to avoid the copy that takes place when the return value is stored in a variable:

```
void createCityMap(std::map<std::string,
int> &map)
{
  map.clear();
  map["Tokyo"] = 28025000;
  map["Mexico City"] = 18131000;
  ...
}
```

The call then becomes:

```
std::map<std::string, int> map;
createCityMap(map);
```

Programming like this can rapidly become tedious. It's also error-prone, because the implementor must remember to call clear() at the beginning of the function.

The main drawback with implicit sharing is that some care must be taken when copying containers across threads. Qt 3 includes a class called `QDeepCopy` that ensures that no two threads reference the same data simultaneously, but it is the application developer's job to remember to use this class when appropriate.

The Qt 4 Way

When we started working on Qt 4, the first issue we had to address was that of the container classes and the other tool classes (notably `QString`). Two of the main goals with Qt 4 were to provide a nicer API that can compete advantageously with newer toolkits and to make Qt more efficient, a requirement from Qt's embedded market.

One option was to deprecate Qt's container classes and to tell our users to use STL. This had many advantages, but also brought its share of issues:

- Existing Qt users are familiar with Qt's API and many of them prefer their container classes to have a Qt-like API.
- STL is not available on some embedded platforms (needed for Qt/Embedded).
- Implementations of STL vary quite a bit, meaning that an application developed on a certain platform might not compile on another because of some advanced STL construct.
- STL is usually implemented with raw speed in mind, at the expense of memory usage and code expansion (the "code bloat" problem).
- STL containers aren't implicitly shared.

For those reasons, we decided to write our own set of containers for Qt 4, which would replace the previous Qt containers and offer a solid alternative to STL. The new containers have the following advantages:

- The containers provide new iterators with a nicer, less error-prone syntax than STL, inspired by Java's iterators. (The STL-style iterators are still available as a lightweight, STL-compatible alternative.)
- The containers have been optimized for minimal code expansion.
- An empty container performs no memory allocation, and only requires the same space as a pointer.
- Even though they are implicitly shared, they can safely be copied across different threads without formality. There's no need to use `QDeepCopy`. This is possible by using atomic reference counting, implemented in assembly language.

The Container Classes

Qt 4.0 provides the container classes shown in Table 1, overleaf.

Algorithmic Complexity

Table 2 summarizes the algorithmic complexity of index-based lookups, insertions in the middle, prepending and appending for Qt 4's sequential containers:

From the table above, it might look like insertions in the middle are much faster using `QLinkedList<T>` than using `QList<T>`. However, in practice, for lists of about a hundred items, both offer more or less the same speed for insertion in the middle. For lists smaller than that, `QList<T>` is faster.

Table 1: QT4.0 Container Classes

Class	Summary
`QList<T>`	This is by far the most commonly used container class. It stores a list of values of a given type (`T`) that can be accessed by index. Internally, the `QList` is implemented using an array, ensuring that index-based access is very fast.
`QLinkedList<T>`	This is similar to `QList`, except that it uses iterators rather than integer indexes to access items. It also provides better performance than `QList` when inserting in the middle of a huge list, and it has nicer iterator semantics.
`QVector<T>`	This stores an array of values of a given type at adjacent positions in memory. Inserting at the front or in the middle of a vector can be quite slow, because it can lead to large numbers of items having to be moved by one position in memory.
`QStack<T>`	This is a convenience subclass of `QVector` that provides "last in, first out" (LIFO) semantics. It adds the following functions to those already present in `QVector`: `push()`, `pop()` and `top()`.
`QQueue<T>`	This is a convenience subclass of `QList` that provides "first in, first out" (FIFO) semantics. It adds the following functions to those already present in `QList`: `enqueue()`, `dequeue()` and `head()`.
`QSet<T>`	This provides a single-valued mathematical set with fast lookups.
`QMap<Key, T>`	This provides a dictionary (associative array) that maps keys of type Key to values of type `T`. Normally each key is associated with a single value. `QMap` stores its data in Key order; if order doesn't matter `QHash` is a faster alternative.
`QMultiMap<Key, T>`	This is a convenience subclass of `QMap` that provides a nice interface for multi-valued maps, i.e. maps where one key can be associated with multiple values.
`QMap<Key, T>`	This has almost the same API as `QMap`, but provides significantly faster lookups. `QHash` stores its data in an arbitrary order.
`QMultiHash<Key, T>`	This is a convenience subclass of `QHash` that provides a nice interface for multi-valued hashes.

Table 2: Algorithmic Complexity of QT4.0 Container Classes

	Lookup	Insert	Prepend	Append
`QLinkedList<T>`	O(n)	O(1)	O(1)	O(1)
`QList<T>`	O(1)	O(n)	amortized O(1)	amortized O(1)
`QVector<T>`	O(1)	O(n)	O(n)	amortized O(1)

Table 3: Associative Containers

	Lookup	Insert
`QMap<Key, T>`	O(log n)	O(log n)
`QHash<Key, T>`	O(1) on average	amortized O(log n) on average
`QSet<T>`	O(1) on average	amortized O(log n) on average

The final table, Table 3, covers Qt 4's associative containers.

It is possible to get amortized O(1) behavior on average for insertions into a `QHash<Key, T>` or a `QSet<T>` by setting the number of buckets in the internal hash table to a suitably large integer before performing the insertions.

Pros and Cons of STL Iterators

The main advantage of STL iterators over any other type of iterators is that you can use them together with STL generic algorithms (defined in `<algorithm>`).

For example, if you want to sort all the items in a `QVector<int>`, you can call `sort()` as follows:

```
QVector<int> vector;
vector << 3 << 1 << 4 << 1 << 5 << 9 << 2;

sort(vector.begin(), vector.end());
// vector: [1, 1, 2, 3, 4, 5, 9]
```

Qt 4 also provides a set of generic algorithms in `<QtAlgorithms>`. This is useful if you build your software on platforms that don't provide an STL implementation (e.g., on Qt/Embedded).

Each Qt 4 container QXxx has two STL-style iterator classes: `QXxx::iterator` and `QXxx::const_iterator`:

- the non-`const` iterator can be used to modify the container while iterating;
- the `const` iterator should be used for read-only access.

The three main advantages of Qt's STL-style iterators are that they are compatible with STL's generic algorithms, that they are implemented very efficiently (an STL iterator is typically just syntactic sugar for a pointer), and that most C++/Qt programmers are already familiar with them. But they have some disadvantages as well.

1. Modifying a container using a non-const STL iterator is notoriously error-prone. For example, the following code might skip one item, or skip the "end" item and crash:

```
// WRONG
for (i = list.begin(); i != list.end(); ++i) {
  if (*i > threshold)
    i = list.erase(i);
}
```

It must be written as follows:

```
i = list.begin();
while (i != list.end()) {
  if (*i > threshold)
    i = list.erase(i);
  else
    ++i;
}
```

2. Iterating forward and backward are not symmetric.

When iterating backward, you must decrement the iterator before you access the item. For example:

Forward

```
i = list.begin();
while (i != list.end()) {
 bamboozle(*i);
 ++i;
}
```

Backward

```
i = list.end();
while (i != list.begin()) {
  –i;
  bamboozle(*i);
}
```

The STL addresses this problem by providing a `reverse_iterator<T>` iterator adaptor that wraps an iterator type to make it iterate backward, as well as `rbegin()` and `rend()` functions in its containers. This enables you to write

```
i = list.rbegin();
while (i != list.rend()) {
  bamboozle(*i);
  ++i;
}
```

However, this solution requires two additional iterator types, `reverse_iterator` and `const_reverse_iterator`.

3. Many users find the operator-based syntax cumbersome.

The operator-based syntax is especially cumbersome when the value type is a pointer, because you then need to dereference the iterator twice -once to get the pointer and once to get the object.

For example:

```
QList<QWidget *> list;
...
QList<QWidget *>::iterator i = list.begin();
while (i != list.end()) {
  (*i).show();    // won't compile
  i->show();      // won't compile
  (**i).show();   // OK
  (*i)->show();   // OK
  ++i;
}
```

The requirement that you must call both `begin()` and `end()` on the container object is also a common source of confusion.

For example, the following code typically results in a crash:

```
// WRONG
i = splitter->sizes().begin();
while (i != splitter->sizes().end()) {
  humbug(*i);
  ++i;
}
```

This is because the object on which `begin()` is called isn't the same as the object on which `end()` is called; `QSplitter::size()` returns a container by value, not by reference.

The Solution: Java-Style Iterators

Qt 4 introduces Java-style iterators as an alternative to STL-style iterators. As their name suggests, their syntax is inspired from the Java iterator classes. They attempt to solve the main issues with STL-style iterators.

Like STL-style iterators, Java-style iterators come in two variants: a `const` and a non-`const` iterator class.

For `QList<T>`, the Java-style iterator classes are `QListIterator<T>` and `QListMutableIterator<T>`. Notice that the shorter name is given to the iterator type that is most frequently used (the `const` iterator).

One nice feature of the non-`const` iterators (the "mutable" iterators) is that they automatically take a shallow copy of the container. If you accidentally modify the container while an iterator is active, the iterator will take a deep copy and continue iterating over the original data, instead of giving wrong results or crashing. This makes Java-style iterators less error-prone to use than STL-style iterators.

The main disadvantage of Java-style iterators is that they expand to more code in your executables.

Java-style iterators don't point directly at items; instead, they are located either before or after an item. This eliminates the need for a "one past the last" item (STL's `end()`) and makes iterating backward symmetric with iterating forward.

Let's see how this works in practice. Here's a loop that computes the sum of all items in a `QList<int>`:

```
QList<int> list;
...
 QListIterator<int> i(list);
    while (i.hasNext())
      sum += i.next();
```

The list is passed to the iterator's constructor. The iterator is then initialized to the position *before* the first item. Then we call `hasNext()` to determine whether there is an item to the right of the iterator, and if so, we call `next()` to obtain that item and advance the iterator beyond the item. We repeat this operation until the iterator is located *after* the last item. At that point, `hasNext()` returns false.

Iterating backward is similar, except that we must call `toBack()` first to move the iterator to after the last item:

```
QList<int> list;
 ...
 QListIterator<int> i(list);
 i.toBack();
 while (i.hasPrevious())
   sum += i.previous();
```

The `hasPrevious()` function returns true if there is an item to the left of the current iterator position; `previous()` returns that item and moves the iterator back by one position. Both `next()` and `previous()` return the item that was skipped.

Sometimes we need the same item multiple times. We cannot call `next()` or `previous()` multiple times, because it moves the iterator in addition to returning the item. The obvious solution is to store the return value in a variable:

```
while (i.hasNext()) {
  int value = i.next();
  sumSquares += value * value;
}
```

For convenience, the iterator classes offer a `peekNext()` function that returns the item after the current iterator position, without side effects. This allows us to rewrite the code snippet as follows:

```
while (i.hasNext()) {
  sumSquares += i.peekNext() * i.peekNext();
  i.next();
}
```

You can also use `peekPrevious()` to obtain the item before the current iterator position. This gives us yet another way of writing the "sum squares" code snippet:

```
while (i.hasNext()) {
  i.next();
  sumSquares += i.peekPrevious() *
      i.peekPrevious();
}
```

So far, we've only seen how to use the `const` iterator types (e.g., `QListIterator<T>`). Non-`const`, or mutable, iterators provide the same navigation functions, but in addition they offer functions to insert, modify, and remove items from the container.

Let's start with a code snippet that replaces all negative values in a `QList<int>` with zero:

```
QList<int> list;
...
QListMutableIterator<int> i(list);
while (i.hasNext()) {
  if (i.next() < 0)
    i.setValue(0);
}
```

The `setValue()` function changes the value of the last item that was skipped. If we are iterating forward, this means the item to the left of the new current position.

When iterating backward, `setValue()` correctly modifies the item to the right of the new current position.

For example, the following code snippets replaces all negative values with zero starting from the end:

```
QListMutableIterator<int> i(list);
i.toBack();
while (i.hasPrevious()) {
  if (i.previous() < 0)
    i.setValue(0);
}
```

First we move the iterator to the back of the list. Then, for every item, we skip backward past the item and, if its value is negative, we call `setValue()` to set its value to `0`.

Let's now suppose that we want to *remove* all negative items from the list. The algorithm is similar, except that this time we call `remove()` instead of `setValue()`:

```
QListMutableIterator<int> i(list);
while (i.hasNext()) {
  if (i.next() < 0)
    i.remove();
}
```

One strength of Java-style iterators is that after the call to `remove()` we can keep iterating as if nothing had happened. We can also use `remove()` when iterating backward:

```
QListMutableIterator<int> i(list);
i.toBack();
while (i.hasPrevious()) {
  if (i.previous() < 0)
    i.remove();
}
```

This time, `remove()` affects the item *after* the current iterator position, as we would expect. If you need to find all occurrences of a certain value in a sequential container (a `QList`, `QLinkedList`, or `QVector`), you can use `findNext()` or `findPrevious()`.

For example, the following loop removes all occurrences of 0 in a list:

```
while (i.findNext(0))
  i.remove();
```

Qt 4 provides Java-style iterators both for its sequential containers (`QList`, `QLinkedList`, `QVector`) and for its associative containers (`QMap`, `QHash`). The associative container iterators work a bit differently to their sequential counterparts, giving access to both the key and the value.

In summary, Java-style iterators solve the main issues with STL-style iterators:

- Iterating forward and backward are symmetric operations.
- Modifying a container using a Java-style iterator is easy and not error-prone.
- The Java iterator syntax is highly readable and consistent with the rest of the Qt API.

Foreach Loops

If you just want to iterate over all the items in a container in order, you can use Qt's foreach keyword. The keyword is a Qt-specific addition to the C++ language, and is implemented using the preprocessor. Its syntax is:

```
foreach (variable, container)
  statement
```

For example, here's how to use `foreach` to iterate over a `QLinkedList<QString>`:

```
QLinkedList<QString> list;
...
  QString str;
  foreach (str, list)
    bamboozle(str);
```

Just like C++'s `for` loop, the variable used for iteration can be defined within the `foreach` statement. And like any other C++ loop construct, you can use braces around the body of a `foreach` loop, and you can use break to leave the loop.

Qt automatically takes a copy of the container when it enters a `foreach` loop. If you modify the container as you are iterating, that won't affect the loop. (If you don't modify the container, the copy still takes place, but thanks to implicit sharing copying a container is very fast.)

Jasmin Blanchette

Reviews

Bookcase

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

Francis Writes

Concerning Writing Style

As I worked at applying the C Vu style sheet to the reviews I became increasingly irritated by some of the reviews. My irritation was not so much with the content but with the writing style. Please do not take the following as criticism of individuals; I do not mean it that way but as guidance.

When writing for publication we need to decide what kind of writing style we will adopt. Few of us have had the luxury of having been taught such things at school, and most writers for C Vu have not had a heavy-handed professional editor critiquing their work. It is important to understand the house style of a magazine or publisher. In the case of C Vu we have a long history of a relaxed style that addresses the reader directly. To do this the writer needs to consider a number of points. Among the most important are:

Prefer the first person to the third; take the reader into your world by addressing them directly. Yes, there are times when the second or third person is more appropriate but you should choose consciously and deliberately.

Avoid using the passive voice. Academics tend to write in the third person and in the passive voice. That may meet their needs but it makes their writing much less approachable. Sometimes the passive voice is fine but try to become conscious of when you choose to use it.

I try to avoid abbreviations such as "wasn't" (though that was not always the case) as such usage is probably too relaxed for a professional publication.

Finally, try to write firmly without being tempted to provide false emphasis. One of my favourite examples is the difference between "I love you." and "I love you very much." The former is unconditional and unlimited; nothing can be a stronger statement. The latter is intended to be stronger but is weaker. Think about it and start trimming your use of 'very', 'quite', 'fairly' etc.

One way to improve your reviewing style is to read book reviews. What is it about some that help? What is it about some that irritate? Try to review your reviews before you send them in.

Francis

The following bookshops actively support ACCU (offering 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 us 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

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.

New Books

I sometimes notice books that I feel deserve bringing to your attention more quickly than via a review that will take time to produce. This is even more the case now that I do much less reviewing myself.

There is a new edition of the C++ Primer out. Stan Lippman wrote the first and second editions by himself. Josée Lajoie joined him as co-author of the third edition. Her intimate knowledge of the core of C++ went a long way to ensuring technical correctness in terms of what C++ had become since the first edition.

Since then a great deal has happened to the style of C++ programming. It is often easier for a newcomer to see how to incorporate changes to a text. I have no idea how Stan managed to persuade Barbara Moo (Andy Koenig's wife) to join the team but it was inspired. Barbara has completely rewritten the book, considerably shortening it in the process. She has added her unique fluency with the English language to an already excellent book. At the same time, I can sense the presence of Andy Koenig looking over her shoulder giving encouragement and adding extra detail.

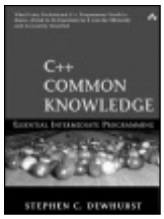
I will say no more because someone else should be doing the review.

I was in Blackwell's recently talking to one of their employees about the books they should be stocking for the ACCU Conference. The Pearson Education rep came in shortly after me and was listening to the conversation. While we were going through the Addison-Wesley lists, my eyes fell on a curious couple of entries: 'The Art of Computer Programming' Volume 1, Fascicle 1 and 'The Art of Computer Programming' Volume 4, Fascicle 2. I know that 'fascicle' means a part work; I also know that TAoCP vol. 4 has never been published. It seemed that there should have been a Vol. 4 Fascicle 1. I asked the Pearson Educational rep if she could cast any light on this. The next day Blackwells delivered advanced copies of vol. 1 Fascicle 1 and vol. 4 Fascicle 2 to my house by hand.

From those I have learnt that Donald Knuth is aiming to publish a new release of TAoCP at the rate of two fascicles a year. Each fascicle will be about 128 pages. They will be something akin to a late beta release of text that will eventually form the definitive version. Fascicle's for a specific volume will not necessarily be published in sequence.

Fascicle 1, Volume 1, replaces the original MIX assembler language with a new MMIX version which is a better match for the current hardware. Fascicle 2, Volume 4 is part of chapter 7 (Combinatorial searching) and covers 7.2.1. I suspect that this is the first time that a chapter of a book has been published in parts and out of sequence. Note that Knuth intends that volume 4 will be published in three parts 4A, 4B and 4C and that most of this will be chapter 7, though the final part of 4C will be chapter 8 (Recursion). So now we have part of a part of a volume. I still suspect that publication any time in the range 2007 – 10 is optimistic for volume 4 so I think these fascicles (or their earlier versions on the Web) are about as good as you are going to get for some time.

C & C++



C++ Common Knowledge: Essential Intermediate Programming by Stephen Dewhurst (0-321-32192-8), Addison-Wesley, 248pp @ £29-99 (1.00)
reviewed by Francis Glassborow

This book is a collection of 63 items with coverage that varies from a single page to half a dozen pages. The author is trying to give brief coverage to those C++ items that he considers are most vital for the intermediate programmer.

Of course, any such selection of topics will have a certain subjectivity to it and I have no doubt that if we surveyed a dozen different C++ experts they would each identify different items as missing and different items as superfluous. In my case, I am not convinced that understanding of template template parameters is necessary common C++ knowledge.

On the other hand, there are places where I think the author should be giving some warnings about the nasty corners of C++ that can surface. A good example is that of ADL (Argument Dependent Lookup) which was once greeted as a brilliant idea. These days many experts treat it with circumspection and I think it important that we tell programmers how to suppress it.

Having said that, I think this is an excellent and relatively brief book. However, its brevity means that topics are necessarily dealt with in a way that assumes a good programming background from the reader, probably rather more than most post-novices will have.

The book's major shortcoming, in my opinion, is a lack of references on a topic-by-

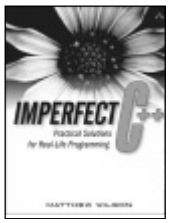
topic basis. There is a reasonable bibliography at the back of the book but that only leaves the reader with a nice list of about a dozen Addison-Wesley books to buy. Direct references at the end of each topic would be far more useful to the reader who finds they need to study a topic in greater depth.

My other big grouse is with the publisher who continues to capitalise keywords when they occur in headings. For example, 'Const' has no meaning, the word can only be spelt 'const'.

This book would be an excellent starting point for discussions that would help develop a C++ programmer's skills. I am less certain about its utility to those transferring to C++ from other languages. For example, Java programmers have a very specific set of problems when starting with C++ (unspecified order of evaluation, lack of forwarding constructors, minimum rather than absolute ranges for built in types etc.)

I think this book would be a worthy addition to the budding C++ programmer's library but it should constitute the start of their studies rather than the totality. They would probably be wise to take their time over studying the latter topics.

Note that the UK price of this book is a disgrace. Bookshops in the UK should be complaining vociferously. The £/$ exchange rate hovers at around 1.90, so a charge of £29.99 for a book selling for $29.99 in the US is completely unacceptable. Continuation of this kind of pricing will just drive ever more customers into the hands of Amazon.



Imperfect C++ by Matthew Wilson (0-321-22877-4), Addison-Wesley, 588pp+CD @ £34-99 (1.29)
reviewed by Francis Glassborow

I find this a difficult book to review, at least in part because I have a lot of sympathy for much that the author writes. The sub-title is 'Practical Solutions for Real-Life Programming' and declares what it is that the author believes he has provided.

However, he often deals with things that are not strictly part of C++, and often things that are dubious extensions provided by some vendor. Indeed, he devotes the whole of the last and longest chapter to the ideas of properties.

The first impression I have of this book is that I want to keep it away from young, impulsive programmers who are always looking for the 'brilliant' hack. Though it is clearly not the author's intent, too many of his ideas and mechanisms invite hackery by the immature.

On the other hand, I would like to put this book in the hands of those who want to develop a thoughtful understanding of how they can use C++ to achieve their objectives. The problem here is that already relatively skilled C++ programmers will want to argue with the author. That is a good thing because their thinking is being challenged, but it too often results in abandonment of a book because the places where you disagree are raising your blood pressure to too great an extent.

The book is littered with good advice (well advice that I agree with and therefore deem to be good) as well as being irritating by giving advice that I think is inappropriate (in other words advice with which I disagree).

If you are past your C++ novitiate and want to study (note that word, reading is not enough) the ideas of an experienced and highly skilled C++ practitioner this is an excellent book. It will be well worth the time you might take studying it. However note that if you are reading this review post 2008, I might say something very different because much of the contents are heavily dependant on the current C++ Standard. A future Standard (due for final release in 2009) will almost certainly remove the need for some of the methods described and provide alternatives – better ones – for others).

I think readers would get best value from this book by using it as a basis for discussion. If you have the good fortune to work in a company whose culture encourages development of skills, this book would be useful for a weekly study group.

If you are looking for instant solutions/gratification leave this book alone because it will only make you even more dangerous as a programmer. If you want to spend time thinking and developing your C++ skills and insights take the time to study this book.

Hachish C++ by Michael Flenov (1-931769-38-9), alist, 325pp+CD @ £23-99 (1.46)
reviewed by Francis Glassborow

I am not going to waste much of your time reading a review of this book because enough time has been wasted by me reading it.

The author's mindset is far away from what I consider to be professional. He seems to think that writing a program that would, for example, hide the user's cursor is both funny and OK. He seems to have no idea that such a program results in uncountable hours of wasted time and frustration.

The use of C++ in the title is possibly misleading because as far as I can see he is writing about programming for MS Windows.

Do I need to tell you to NOT buy this book? No, because no professional would want to encourage such writing.

C# & Java

Hibernate A J2EE Developer's Guide by Will Iverson (0-321-26819-9), Addison-Wesley, 351pp @ £30-99 (1.29)
reviewed by Peter A. Pilgrim

Note that the following review was wrongly attributed in C Vu 17.2 to 'Hibernate: A Developer's Notebook' by James Elliot.

Hibernate: A Developer's Notebook is one of the first titles in a new series of O'Reilly books. The notebooks are supposed to contain just enough information to let you quickly learn about a new API or project; but only just what you need to "make it work". The book is divided into nine chapters, three appendices and has 178 pages.

The first chapter is short and describes installation and set up of Hibernate. The examples require the Ant, a build tool, in order to generate the executables. The examples are based on the Java based embeddable database HSQLDB. The author describes where to download Hibernate and install it within a project hierarchy.

After a short introduction into the world of object relational mapping O/RM, the second chapter starts working with Hibernate properly. We are already at page 13! The first topic is the mapping document. Hibernate frameworks require a XML mapping file to define meta data of plain old Java objects in terms of database persistence. The author demonstrates that mapping documents can be written quite practically with a text editor. Contrast this "mining at the coalface" approach in this notebook to the reference style of Bauer and King Hibernate in Action, and we are only at page 18 of the O'Reilly book. By the way, the business domain of the book examples is a MP3 album / track information database, which works with Java based embedded database, HSQLDB.

The third chapter explains how to create persistent objects in the database using the Track POJO class, which was generated in the previous chapter. In the following sections of the chapter, the author carefully explains the life cycle of mapped objects, and how they move from transitive to persistent states. Moving in the reverse direction of state is about finding persistent objects. If you want a fuller understanding of Hibernate state management, then the Manning book is the perfect reference. Elliot describes the process of finding saved Track objects, the retrieval make use of the infamous HQL. In contrast to the Manning book, HQL is introduced early on to demonstrate how similar the query dialect is to ANSI SQL. Of course, HQL is semantically different from SQL. The last sections of the chapter tell how to delete a persistent object, and describe the named parameters of the HQL query engine.

A database that only could persist simple POJOS would be next to useless. Typically, any data structure in an application required a type of collection to manipulate individual elements or to describe relationships between two or more different classes of objects. Chapter five is about mapping collection and declaring associations and it introduces a new class Artist.

The only criticism of the this chapter is that it completes missing out or fails to explain that Hibernate can map complex Java collection data types such as bags, lists, and ordered types.

The fifth chapter expands on the previous one with richer associations, starting with lazy associations. The author justifies the reason for lazy loading of associations especially for linkages between big tables. There is a section dedicated to ordered collections. The

associations between tables (Java classes) can be augmented as much you would expect in a business schema.

Chapter six describes how to persist enumerated types as Hibernate data types. In order to add persistence for enumerated data, then your POJO class must implement the PersistentEnum class.

Chapter seven is about custom value types, which are types that the developer defines and thus extends the default Hibernate data types. The author explains that complex and highly nested object structures can be persisted to a database using the framework.

Chapter eight discusses criteria queries with the major example to find MP3 tracks shorter than a specified length. Programming with the Criteria API is much like building functional objects and constructing them as an expression.

The subject of the final chapter is the Hibernate Query Language, which appear several times earlier in the book.

In conclusion, HADN is a great book to have on the coffee table, if you want are in hurry to get a project up and running with Hibernate. There are enough working examples for the developer to build functionally software without taking shortcuts. The only criticism is that such a book will not teach you advanced tips and tricks, so you may grow out of it pretty quickly if you suddenly become a seasoned Hibernate Expert. This latter important point will be sorely important if you need to optimise Hibernate for an enterprise quality application. Nonetheless on average, you will be sort of person who just wants to see best practice and actually view some code that just works as it says on the tin. This book is recommended.

Java A Complete Course by Stuart F. Lewis (1-8264-5927-7), Continuum, 342pp @ £19-99 (1.58)

reviewed by David Sullivan

What better way to learn Java than to read and review a complete course on the subject? This book was selected on the basis of the title alone. There is a trend amongst publishers of selecting a catchy title that creates an impression of what the book is about. It is a course granted, but it is far from complete and in some areas rather poor.

The book does strive to encourage good practice. In the introductory sections, documentation, analysis and design are featured.

The code examples take the form of a number of small projects. It is a pragmatic style of teaching material, which demonstrates concepts mainly through the example code, combined with a dry rather dull narrative.

There is no emphasis on important concepts and explanations do not illuminate the concepts well. For this reason alone the book cannot be recommended to readers learning their first programming language.

The programming examples do demonstrate concepts but are consistently not well chosen and do not enthuse. The best mini-project example in the book is a Mastermind game. Potentially a good idea but the GUI version contains text boxes that indicate the results of each move. A version with real graphics would have been more exciting.

Generally the book lacks attention to important aspects which would make the book successful. An example of this is the inclusion of package statements (e.g. package chap03) in most of the code examples. The explanation that these statements should be omitted and are just for the author's benefit is embedded somewhere in the text - as if an afterthought nowhere near the first occurrence of a package statement in the code.

Seeing the use of package in almost all the code, one might wish to find more information. Little exists. There is nothing in the index. A brief description that a package is "a group of Java classes" does exist in the glossary.

The book is an introductory course. Generally it is uninteresting and uninspired. There are much better books available. Not recommended.

Core Java 2 vol 1 7ed by Gary Cornell & Cay Horstmann (0-13-148202-5), Prentice Hall*, 762pp @ £39-99 (1.25)

reviewed by Alan Barclay

This is the latest revision of one of the best selling and most respected texts on the Java programming language.

Completely updated for J2SE 5.0 it will provide an excellent and concise insight into the language as well as the changes and new features offered by the latest release of the Sun JDK (now with over 3,000 API classes).

It is directly aimed at experienced programmers rather than complete novices.

For me Core Java has always represented the definitive Java language tutorial because it was the first Java book that I read and this edition continues the good work.

In addition to informative coverage of Java topics, this book is just littered with real world code examples (no toy examples), high quality illustrations and screen shots, Notes, Tips, Cautions and C++ remarks (although knowledge of C++ is not essential).

It is a book which I found that I could always open at a random page and find something interesting and informative to read, such is the excellent style and content.

Although covering just the fundamental parts of Java this is still a relatively large text and in this latest edition would be a very long cover-to-cover read.

The book itself would have been even bigger had the publisher not chosen to shrink the font size considerably from that used in previous editions. Although still clear this denser format is almost a concern but is my only negative comment.

Available on the authors related website are sample chapters, source code download and a not insignificant printing errata (for all editions in the series).

A great effort and worthy addition to the bookshelf of any would be serious Java programmer.

Highly Recommended.

Effective C# by Bill Wagner (0-321-24566-0), Addison-Wesley*, 307pp @ £28-99 (1.38)

reviewed by Pete Goodliffe

When I agreed to review this book I was worried that it would be a pale rip-off of Meyers' Effective C++ books. The book is actually part of a new series edited by Meyers himself – and it shows. Bill Wagner is a good writer, and you can clearly see Scott Meyers' hand here too: the book is well written, well balanced, the choice of topics is well made, and the advice is excellent.

Like the Effective C++ series, this book aims to teach programmers with reasonable prior experience about the correct idioms and common pitfalls of the language. It will be particularly useful for programmers transitioning to C# from C++ or Java. This book will not teach you C#.

The text is engaging and well written; Wagner explains lucidly and at an appropriate level of detail. It is clearly laid out and well structured: there are 50 specific items with memorable sound bite titles and deeper discussion.

The only thing that lets Effective C# down is the plethora of typos. The book is riddled with layout mistakes that make it look like it missed a final review. This is shame, but not a good reason to avoid purchase.

Effective C# is highly recommended, and if "More Effective C#" ever arrives I'll be eager to see it.

Microsoft Visual C# .NET 2003 Unleashed by Kevin Hoffman & Lonny Kruger (0-672-32676-0), SAMS*, 976pp @ £42-99 (1.40)

reviewed by Richard Putman

This book attempts to cover the vast range of the .NET framework using C# as well as the C# language itself. It has nine sections totalling 916 pages plus a comprehensive index and starts with a walk around Visual Studio. This is very difficult to do in words and it may have been better to point to one of the downloadable guided tours on the MSDN website.

The syntax of C# described in the next 300 pages takes you from basic control structures through to COM and code reflection. The introduction to C# chapter is bizarre – the third program shows a SQL connection class implementing `IDisposable` and we do not see "hello world" until the end of the chapter! Using undefined terms such as "metadata" will only confuse a beginner and there is no glossary of terms. Yet the beginner is clearly the target audience for the book: chapter 3 has a two-page program to demonstrate the use of logical AND and OR with a score of `Console.Write` statements – surely the most confusing way to demonstrate such a simple concept.

The remaining parts cover Windows Forms, Web Applications, Data Access, Web Services, Secure Applications, Enterprise and Connected Applications and Debugging and Testing. It is odd, given the scope of the book, that it omits any instructions on setting up IIS needed run many of the examples and instructs the reader to look elsewhere.

For some reason the chapters start on the left hand page with a useless "What you need" section containing entries such as "Recommended Software – Visual Studio.NET" and "Skills Required – C# coding". Most of the code layout is poor with messy indentation, multiple blank lines and regions of highlighted code that make reading the programs painful. The authors have often not even been bothered enough to remove the auto generated comments for class descriptions etc.

It seems in a desperate scramble to reach the thousand-page count for the book, the authors wasted space wherever possible in the early chapters. For instance, in the chapter on Windows Forms the same picture of their demo window appears no less than nine times.

The later chapters are considerably better and provide a useful introduction for the more advanced features of .NET that often require a separate book to cover in detail.

However, taking the book as a whole, it fails as a beginner's guide with often trivial and uninspired examples and no glossary. Too many worthless descriptions such as "The Call Stack window displays the functions on the call stack" combined with sloppy formatting of code make the book feel carelessly put together. Not recommended.

Python

Dive Into Python by Mark Pilgrim (1-59059-356-1), Apress, 413pp @ £25-00 (1.60)
reviewed by Lars Hartmann

The book starts out with a section on how to install python on various systems. I decided to use my laptop for the experiments with python. My laptop is an apple Powerbook G4. This posed no problems, the book comes with detailed instructions both for Debian and for OS X. I had no problems following the books step-by-step guide. The book also contains information about installation on different varieties MS Windows, Red Hat Linux and Mac OS 9. I have no reason to believe these descriptions will differ in accuracy.

The book is sprinkled with references to web pages. The references provide extended information about the inner workings of python. Some provide guides detailing how to write python code in a consistent way. All this information is a nice to know, not a need to know.

The book is nicely organized. It will carefully lead the reader through all the major areas of the language. The later chapters even touch upon advanced areas of software development, like Unit testing, refactoring and performance tuning.

The book is quite easy to read. All example code is given in the book. This makes it really nice for reading on a bus or train. All the code is carefully explained, and some of it is even in production on Mark's own web site.

Python comes equipped with a very powerful regular expression engine. Mark does a good job of explaining the basic ideas in regular expressions. After the chapter on regular expressions they are used in more and more advanced ways. Mark is very careful to explain when and why these advanced techniques are used.

The books index is quite good, which makes it usable as a reference. Once you start writing python you will use the python.org web site more than the book though.

Embedded and Real Time

Embedded C Programming with the Microchip PIC by Richard Barnett, Larry O'Cull, Sarah Cox (1-4018-3748-4), Thomson, 500pp+CD @ £46-00 (1.63)
reviewed by Derek Jones

This is an all-in-one book aimed at the introductory programming student market.

Is it worth buying this one book, or should more money be spent buying separate books dealing with learning C, programming the Microchip PIC processor (a very similar book by the same authors deals with the Atmel AVR), an extended programming project using this processor, a library reference, plus a CD containing an IDE and C compiler? I would suggest buying two books, a book on learning C and this book (or its companion if your lecturer targets the Atmel AVR).

The C tutorial in this book reads like it has been poorly cut and pasted from a set of more substantial notes by somebody who does not know the language very well. No attempt is made to distinguish between Standard C and the extensions commonly found in embedded compilers. A listing of the C syntax would be useful to the intended readership.

The chapters dealing with programming the Microchip PIC and the embedded programming project (which is a different one from that given in the Atmel AVR book) give lots of details. While they would be out of place in another book, the extensive coding examples are probably of use to the intended readership.

Although I did not work through the examples in detail, I got the feeling they were pitched at about the right level for students taking their first programming course. The index also appeared to be extensive enough to be useful.

If you are a student on a course based on this book then you will not be wasting your money buying it. If you already know C and are looking to learn about the practical details of writing embedded applications, then this book is one to look at. The material is not of sufficient quality at the price it is pitched, for me give it a value for money recommendation.

Database Programming

Designing Effective Database Systems by Rebecca M. Riordan (0-321-29093-3), Addison-Wesley*, 353pp @ £35-99 (1.39)
reviewed by Thomas Padron-McCarthy

There are tens of introductory database books that cover the database field from side to side. Most of them are a thousand pages or more. This book is different in that it is much shorter and also more practically oriented. (And more enjoyable to read!) Instead of "covering the database field", the idea is to help you Succeed In Developing A Database-based application. Because of this, it covers more than just database theory, and there are no chapters with details about data structures, locking algorithms and the other internal parts of a database management system.

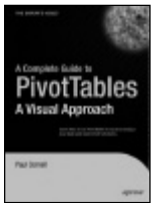
The first third of the book covers the same database theory that can be found in any introductory database book: Entity/Relationship diagrams, relational databases, SQL, normal forms. It is well written and easy to understand, with good examples and many practical tips. It also contains an unusually large section on data mining, and how to design databases for data mining.

The second third of the book is about how to run a software project, and the third part is about how to design the user interface for a database-centred application. The focus is on database applications, but much is general information that is useful for anyone who works in a software project or builds a user interface.

The book is entirely focused on Microsoft products, but even if you do not have much use for detailed information about the SQL dialects of Microsoft Access and Microsoft SQL Server, the more general knowledge and techniques will still make this book worth your time.

If you design and develop database applications, and have less than twenty years or so of experience, do read this book.

Recommended.



A Complete Guide to Pivot Tables.-A Visual Approach by Paul Cornell (1-59059-432-0), Apress, 338pp @ £22-00 (1.59)
reviewed by Richard Knight

From the outset, some familiarity with Excel is assumed, although the reader needs only minimal experience. No CD ROM is supplied with the book and the reader is directed to the publisher's website to download data and code samples. It seems odd that some but not all the relevant examples are available for

download.

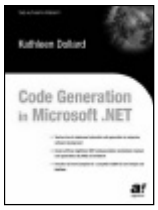
The book is typical of many "beginner" types: the font is large and the text is kept simple. The author frequently repeats most of his instructions. With each Chapter the author effectively repeats what he has previously taught but introduces new techniques. Each Chapter generally concludes with a "Try It" section. There are seven chapters in total and by Chapter 5 it is assumed the reader understands Pivot tables. At this point he introduces Pivot Charts and clearly explains when and why to use Charts rather than tables.

His penultimate Chapter discusses dealing with large collections of data, OLAP and multi-dimensional datasets. This could be a rather large leap for many readers. The final Chapter deals with programming Pivots using VBA. This Chapter is not a VBA tutorial and the reader will need to be familiar with VBA up to intermediate level. His example code is clean and well structured.

This book covers Pivots thoroughly. The author even points out what could be construed as some of the failings of Pivot tables, such as the problem with formatting being lost when changes are applied. Once the reader has appreciated how the book is structured (that is wait until told before trying) the book is extremely easy to read and follow.

Recommended.

General Programming



Code Generation in Microsoft .NET by Kathleen Dollard (1-59059-137-2), Apress, 730pp @ £44-00 (1.36)
reviewed by Frank Antonsen

My over all impression of this is quite favourable, so if you think about using code generation you could this book a try.

In .NET there is a rudimentary code-generation tool built in, called the CodeDOM. With this you construct the parse tree in a generic fashion, attach a formatter for a particular .NET language (in theory at least – only VB.NET and C# are currently available), which produces the actual code in that language. This is do-able, but not very flexible. Instead, Kathleen suggests using XSLT.

She has developed a "code generation harness", which can generate code in any language from metadata expressed in XML. She gives detailed descriptions of how to extract such metadata from a database, modelling the business objects, and how to use the technique to generate VB.NET code, ASP.NET mark-up and SQL Server stored procedures. She even shows how to generate Visual Studio project files. I would like to have seen C# output as well, but at the Apress website a C#-version can be found, so this is a minor issue.

These descriptions are fairly detailed, but I would have liked to see more emphasis put on how to actually write a code generation tool. This should not detract from the overall impression of her harness, though. This really does seem to be rather flexible and adaptable.

As it stands, the book is mostly a tutorial in using her system. In two appendices, however, she begins to show the underlying code, walking through the important bits. The frequent forward references suggest that these chapters are in the wrong order.

Although generally well written, sometimes the book looks as if it has been written from a series of articles in a hurry, with a lot of repetitive remarks. I cannot see why every single chapter has to explain the meaning of the acronym XML. These glitches are relatively few, and certainly not enough to become annoying, but can be corrected in the second edition, along with the typos.

The overall architectural design of her harness is quite sound, and she puts a lot of emphasis on safety and validation. She gives detailed walk-throughs of both vital parts of the VB code for the harness and the most important XSLT style sheets for performing the generation of code from XML metadata. Her explanations are almost always very clear; she quite obviously knows what she is talking about.

To conclude, she presents a very interesting approach to code generation, and she has certainly made me think more about this topic. That alone makes the book well worth reading, even though I have a few small reservations. Recommended!

Methodologies & Management

Extreme Programming Explained 2ed by Kent Beck (0-321-27865-8), Addison-Wesley*, 189pp @ £24-99 (1.40)
reviewed by Alan Lenton

This second edition of Kent Beck's seminal book on extreme programming is even better than the first edition. Extensively rewritten, it now draws on the lessons of the last five years to deepen and extend the philosophy behind extreme programming.

I do not know why it is, but it must be significant, that of all the computing books I have read or reviewed in the last two years, the ones about agile and extreme programming stand out for good writing and an ability to project enthusiasm. This book is no exception. One caveat though, do not expect it to give you a check list to tick off for extreme programming – this is a book concerned with the philosophy of extreme programming.

This book will be of use to a number of different categories of programmers. For those who would like to find out about extreme programming, it is a classy advocacy and explanatory book. For those who would like to introduce extreme programming into their workplace it is chock full of information that can help you win the struggle. Finally, for those already practicing extreme programming, this book can fill in the gaps and give you ideas on how to take things forward.

Of course, there are other things I would have liked to have seen in it. More discussion on large projects for instance, or at the other end, discussion of extreme programming practices for the lone programmer. As the only programmer in my company, I find it difficult to practice pair programming.

However, these are not deficiencies – no reasonable sized book can cover everything - and I would heartily recommend this book to any programmer who wants to deepen his or her understanding of the craft.

Just one question Kent: How is it that the prophet of the short cycle and refactoring took five years and a complete re-write to produce the second iteration of his book?

Highly recommended.

Organizational Patterns of Agile Software development by James Coplien & Neil Harrison (0-13-146740-9), Prentice Hall, 401pp @ £?
reviewed by Francis Glassborow

This book was a run-away seller at the recent ACCU Conference. Blackwells sold every single copy available in the UK. I know that because the publishers shipped 40 copies from the US for sale at the conference on the strict condition that they might not be sold anywhere else. Apparently it is not currently for sale in the UK. Neither the lead author nor anyone else I know has been able to discover why this is the case.

Now to the book itself. As you would expect from the lead author it is well written and challenges your thinking. It covers almost 100 organizational patterns culled from a study of more than 100 software development organizations.

The idea for Patterns was imported from the works of Alexander who was concerned with architecture (of buildings). The pattern concept was first applied to the ideas of software architecture. While I think that the idea has sometimes been abused or at least greatly over used, I am sure that it has done much to help software development.

In this book, we move on from the architecture of software to the 'architecture' of organisations that develop software. In my opinion, there is nothing very special about such organisations. Good organization is good organization regardless of the product. The reason I make that comment is that I find this to be not only an excellent study of software development organisation, but also an excellent study of organization.

Very little other than the examples in this book is specific to software. It is sad that the title will result in the book being stuck on the specialist shelves of bookstores. I would love to see this book rewritten and re-titled so that it will be read far more widely.

In the meantime, if you are involved in managing software development, buy this book, read it carefully and consider how you can apply the ideas to your own work.

In addition, if the publishers are still not distributing it in your country ask them what they are playing at because you deserve access and the authors deserve their royalties.

MDA Explained: the Model Driven Architecture by Anneke Kleppe et al (0-321-19442-X), Addison-Wesley, 170pp @ £26-99 (1.30)
reviewed by Fazl Rahman

This book is part of Addison-Wesley's Object Technology Series edited by the three Amigos (Booch, Jacobson & Rumbaugh). Its basic premise is that the way we develop software nowadays using languages like C++ and Java (3GLs) can be likened to the way machine code and assembler were used in the past: Nobody in their right mind nowadays would try to develop, say, a new word processor application in assembler; the authors predict the same will be true in the not too distant future to our current crop of 3GLs. Instead, they propose a world where (certain classes of) software systems will be built by simply constructing an evolved version of a UML model, then automatically generating the resultant deployment executables.

They admit that the current state of the practice does not match the promise, but they show how things are moving nicely along: they provide a (concrete) toy example. Enterprise information systems that allow access to a relational database via a web-based front end with business logic held in EJBs have been around long enough for common idioms and patterns to have become well established and amenable to automatic generation. The authors' case study, Rosa's Breakfast Service, is precisely such a system. I had difficulty getting hold of the source off the web (in particular the associated MDA tool).

The authors believe that once the gap between a design model and the final deployment executables is bridged, the arguments of Agile Methods proponents (which currently belittle UML models as 'mere designs' drawn up once and obsolete soon thereafter) will fade, and that people will become able to focus on building systems to fulfil business needs while letting the MDA transformation tools take care of the details concerning the technologies used to deploy the system. An order of magnitude hike in productivity, with no loss in maintainability of the system because full semantic content will be preserved in the models; much of this content being otherwise taken away from a project once programmers have left in today's world.

In the world painted by this book, the majority of software developers would become high-level designers, and coders will be heading for extinction. A relatively small elite group of programmers would focus on producing the transformation definitions that would allow the MDA transform engine to convert these high-level 'UML diagrams on steroids' into deployable executable code for a given combination of deployment platforms (OS and programming languages).

I have a sense of déjà vu about this prediction – how 4GLs and CASE tools came with a similar promise in the 80s and failed spectacularly to deliver.

I am willing to consider this may just be stick-in-the mud thinking and that developers using MDA will be able to enjoy their work, but I can't help a sense of loss of something important if we arrive in this predicted future. This despite the fact that I am sure it makes good economic sense and perhaps good engineering sense to allow the many to leverage the skills of the few.

The book is not very dense, and most of it could be read in a weekend. I found it got a bit tedious and switched off at the detailed description of the transformations needed to turn Rosa's model into a deployable EJB Java source base, but some people may find the detailed recipe for setting up JSPs as the GUI for an EJB system useful in its own right.

There is also a short chapter on meta-modelling which explains the four OMG modelling layers.

Overall I remain unconvinced by the authors enthusiasm for MDA. I can easily see it take off in certain niche areas (even large niches like J2EE systems), but I have difficulty imagining a future where there will be less demand for 3GL programmers than today, unless we are not using computers anymore.

Participatory IT Design by Keld Bo/dker et al. (0-262-02568-X), MIT, 337pp @ £32-95 (1.52)
reviewed by Alan Lenton

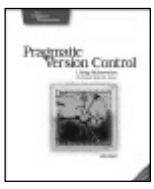
I found this book hard to read. The style is very pedestrian which does not make for a particularly smooth read. That is probably unfortunate, because the book is attempting to grapple with one of the key problems facing any bespoke business IT project – the fact that the program you have yet to write is a critical part of the environment you are trying to design it for.

The book has been written to support a method called MUST (the authors do not say what the acronym stands for, other than the fact that it is Danish). MUST is a method that separates the design of project from its implementation, and the methodology applies only to the design.

The essence of the book's argument (as you would expect from the title) is that users must be genuinely involved in the design process otherwise it will fail. All well and good, but by failing to carry this through to the implementation of the product, the authors allow no space for changing expectations once the users start working with the resulting program.

It is difficult to see how non-technical users can be expected to envisage how they will actually use software, when merely presented with an abstract description – especially when the designers, who are presumably experts, frequently fail to anticipate such uses themselves.

This, to my mind, makes the process flawed from one of its key initial premises.



Pragmatic Version Control Using Subversion by Mike Mason (0-9745140-6-3), O'Reilly, 207pp @ £20-95
reviewed by Pete Goodliffe

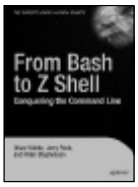
This book is part of the Pragmatic Programmer's "Starter Kit" series. As you would expect from these guys, it is an excellent book with well-balanced coverage of the subject area. Mason does a good job of describing the common problems and challenges, and shows how to implement a sound version control system using Subversion. (Subversion is a relatively new CVS-replacement version control system that is really very good).

The book is well written, with a good balance of humour and readability, and is well laid out and structured. It is clearly not a rehash of their CVS book, but has been thoughtfully created to stand in its own right.

Mason covers basic and advanced version control topics in a way that is not overwhelming. The tour includes an introduction to version control, a Subversion tutorial, recipes for common Subversion operations, clear guidance on how to organise a repository, how to manage third party code, and how to administer repositories.

This is not a definitive reference (get O'Reilly's Version Control with Subversion for that), but it is highly recommended. If you are using Subversion or thinking of deploying it, get this book.

Non-Programming



From Bash to Z Shell: Conquering the Command Line by Oliver Kiddle, et al. (1-59059-376-6), Apress*, 446pp @ £22-00 (1.60)
reviewed by Alistair McDonald

This book is a comprehensive guide to Unix shells, focussing particularly on Bash and Z shell.

The book is very well written and thought out. The team have done a good job of ensuring that each chapter has a similar feel; this is not a series of chapters by different authors, but a single, coherent book. The writing style is engaging but not chatty.

Part 1 contains an overview of shells. This includes the basics of entering commands, command retrieval, and also covers many basic utilities used from the shell, such as man and find. This section covers sh, ksh, csh, as well as bash and zsh. Even experienced users may find something useful here.

Part 2 describes more advanced shell topics, including startup options and customisation of the shell. The shell history - previously entered commands – gets a chapter of its own, as do customising the prompt, pattern matching, job control and pattern matching. This is the bulk of the book, and will probably be the most relevant to most readers. Each topic is covered comprehensively in a logical manner

Part 3 is on extending the shell. It covers scripting techniques such as the use of arrays and hashes, completing functions and editor commands. These are advanced topics, and as far as I can tell are covered in the same manner as the earlier chapters.

On the back cover, the book is marked for Beginner to Intermediate level. This should probably say Beginner to Advanced. This book benefits a complete read-through, as there are snippets of useful information throughout.

The Best of Verity Stob by 'Verity Stob' (1-59059-442-8), Apress*, 315pp @ £17-99 (1.39)
reviewed by Mark Symonds

This book consists of highlights from the Verity Stob columns published in EXE and Dr Dobb's Journal magazines and now on The Register web site.

Verity Stob is a fictional programmer working in the UK and the light-hearted articles cover her career, general industry stories and gossip and poems to the rhymes of popular songs.

The articles date from 1988 and cover industry events up to the present day including such essentials as why Mrs Bill Gates calls in Visual Basic programmers to fix her plumbing, the millennium bug (with articles dated 19100) and extreme programming.

Some articles are a trip down memory lane to the time predating GUIs and many programming methodologies old and new are thoroughly debunked.

Recurring problems at Borland appear in several articles as the History of the Borlandites, which is biblical in tone and concerns the renaming of Borland as Imprise (and back again).

Of course, more technical issues are covered such as why the 'I Love You' virus is an excellent introduction into the arcane machinery of OLE automation.

The Google phenomenon is mentioned with a surreal article about how the search engines boxes gradually get personalities defined by the search engines that they have spidered.

Thirteen ways to loathe VB is one of my favourite articles and shines light on many a dusty corner such as 'The index of the first element is 0, unless it is set to 1 by a directive' and "The four magic constants of the apocalypse: Nothing, Null, Empty, and Error."

Recommended

Linux Specific

Linux Device Drivers by Jonathon Corbet, Alessandro Rubini & Greg Kroah-Hartman (0-596-00590-3) O'Reilly 636pp @ £28-50
reviewed by Ian Bruntlett

This book has been updated for v2.6.10 of the Linux Kernel and does not cover 2.4 due to lack of space. It includes a 45-day trial of O'Reilly's online library (Safari), something I regard as a dubious privilege. The authors have very generously chosen to make this book available under the Creative Commons "Attribution – Share alike license, V2.0" at `http://www.oreilly.com/catalog/linuxdrive3 .`

This book shows how to write Linux (v 2.6.10) device drivers in C. It also discusses how the kernel works, its device driver model, locking & concurrency and various peripheral busses (including USB, PCI) and how kernel code can be written in a way that is not locked into a particular vendor's platform.

Kernel code must be re-entrant to make sure there are no race conditions – this applies to single processor systems as well as SMP systems - because the Linux Kernel is now pre-emptible. This book requires an understanding of C & UNIX system calls and it discusses the Linux device model, introduced in Kernel 2.6. Therefore, I recommend that if you rely on this book, you should also read "Linux Kernel Development 2e" by Robert Love, ISBN 0-672-32720-1.

This book is in two parts. Part One introduces kernel modules and all the details necessary to write a char-oriented device. Part Two is more sophisticated describing block (i.e. hard disk, floppy disk etc) devices, network interfaces and advanced topics (working with the virtual memory subsystem and the PCI and USB buses). Also, there is a reference section at the end of each chapter – a good idea that more authors should do. This book presents source code from example device drivers (I did not have time to test them out) and appears to rely on either GNU or C99 (?) extensions to C. I would like to know where I could get hold of information about these extensions.

The challenge of concurrency in the kernel is raised and it is emphasised that Linux kernel code, including device driver code, must be re-entrant – it must be capable of running in more than one context at the same time.

Recommended for all Linux programmers – kernel programmers need it for a reference. User space programmers need it to understand the operating system that they are using.

Linux Shell Script Programming by Todd Meadors (0-619-15920-0), Thomson, 560pp+CD @ £32-00 (1.84)
reviewed by Tim Penhey

To paraphrase the preface this book was designed to provide the intermediate programmer with the knowledge to write shell scripts. It mentions that it would be helpful to have taken at least one programming or programming design theory and logic class, and that familiarity with basic Linux or UNIX concepts is essential.

The book seems to contradict itself in a number of places with regards to the knowledge assumptions mentioned in the preface. In the first sentence of the preface it says that the book is aimed at the intermediate programmer, and then in the second it says that it is helpful to have taken at least one programming class. The book then tries to give all information right from first principles. The book says that familiarity with basic *nix commands are essential but then has pages on things like echo, history, cal, date etc. The book spends a large amount of text explaining basic concepts like conditional statements, looping, and functions, which should be understood by even a beginner programmer.

The layout of the book seems to be aimed very much at the tutorial experience; descriptive text with numerous step-by-step examples with a multi-choice review quiz at the end followed by project ideas. However the step-by-step examples tend to give every key press, for example, type vi .bash_logout, and then press Enter. The shell script is displayed on the screen. Press Shift+G. Your cursor moves to the bottom of the screen, et al. Perhaps this again is targeted at the wrong level. In addition, the "projects" at the end of the chapters go something like this: login, edit script xyz, type in the following, execute with these parameters, record result, logout.

Even if we assume that the book is targeted at beginners and not intermediate programmers, why teach the basics of programming with bash. Surely Python or modern C++ would be better.

I would say that this book would be sufficiently slow to use for a first year paper in script programming, and even then the patronising tone of the book would drive some to distraction. In addition, even given this, I would not want a university to teach programming basics with shell programming, so given that I do not see where this book would have a place.

This book is definitely not recommended. If you want to learn the basics of bash programming I would suggest "Learning the bash shell" from O'Reilly, or "Unix power tools" for more nitty gritty bits.