Contents

Reports & Opinions

Reports

Dialogue

Features

Reviews

Copy Dates

C Vu 16.6: November 7th 2004

C Vu 17.1: January 7th 2005

Contact Information:

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

Advertising: Chris Lowe
ads@accu.org

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

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

Secretary: Alan Bellingham
01763 248259
secretary@accu.org

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

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

Membership fees and how to join:

Basic (C Vu only): £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

This is my third edition editing C Vu. It's been a cracking experience which has been made all the more pleasant by the help I've had along the way. It's always good when material comes in well in time and more rewarding when readers write in to the letters page with comments on the material. This edition is a case in point – there are more letters in this edition than I've seen in quite a long time.

Is this down to the material being more thought provoking, the range of articles being more varied or simply people feel like emailing? It's hard to say.

Not all emails get published though. For example, I've had some through expressing concern on the focus I've brought to open source coding (the GTK articles, wxWidgets [starting this edition] and Qt have raised a few eyebrows). This is in part quite deliberate, but also down to one other reason : nobody has submitted material on the likes of MFC. We can only publish what is submitted!

Why Deliberate?

C,C++, C#, Python and Java (to name a few) are platform independent languages. It makes no difference which platform code is developed on, as long as they use the pure language and the compiler used at either end is standards compliant (or close to it!), the outcome will be the same. C on my RiscPC will compile on my Linux box and on my friend's OS X machine. The outcome will be the same. That is part of the beauty of these languages. Write once, compile many, outcome same.

The same applies for widget libraries. I enjoy writing code which compiles on my Linux box, take it into work and compile the results on MSVS.NET and see the same results. The independence of the widget library is great in that way. To me, this is an extension of the same ideas as are behind the platform neutrality of the languages we all love and know.

By using these cross platform libraries, it is my firm opinion that there can be a massive increase in productivity as well as stopping some of these "we have this, you don't" arguments you see from time to time. Imagine something like 3D Studio Max or Sibelius 2 being written using (say) wxWidgets – the companies behind them could very quickly and easily produce versions for many platforms (for wxWidgets, it ranges from 16 to 64 bit platforms). Upshot would be a great deal more money for the companies.

What does need to be asked though is why is this not done? Many pieces of software are available for both OS X and Win32. They're not using the same widget sets and required different teams of programmers. To me, this seems very wasteful. Fair enough for something which is Win32 only and requires MFC and other proprietary libraries, but for the rest where the two platform system is used, it would make more sense to use an independent widget set. One code base, compile many, lots of money!

Despite what some of the worlds largest companies say, the use of cross platform libraries is gaining in popularity and moreover, gaining in speed.

One of the largest problems though with some of the cross platform libraries are the licences. Qt is free for X11 and OS X, yet the Windows version requires licences. Many managers don't understand the implications and ramifications of using GPL libraries. I don't know the solution for this, though a simplification of the licences would certainly help.

Books Books Everywhere and Not a Drop to Read

One of the problems you have when you review lots of books is what to do with them after you've reviewed them. Now, this is not a problem for the better books. They are on my bookshelves, waiting for their next set of being used.

The problem comes with the books which are dangerous. You know the sort – they have "*not recommended*" in the book reviews and after you read the review, you can imagine the reviewer dancing around a small bonfire made from the dead trees wasted on such a pile of rubbish. No responsible person would try to sell them on eBay or put them into a recycling box in case some poor person at the recycling plant gets hold of the book and decides to read it.

You could leave them on the shelf, but then they're taking up valuable good book space. I suppose putting them in the loft would be an idea – however even that has its drawbacks (mice).

What would be quite fun would be to get all of the authors of these utter turkeys in a field and have people pelt them with pages from the books while chanting "*you shall not write such utter tosh in future*". For good measure, some of the technical editors who have supposed to have read these books should also be put in the same field. Okay, it wouldn't be very productive, but it's one way of getting rid of the books! That said, I have a feeling that one or two authors in particular would pay little or no attention to such activity... (no names, no pack drill).

It's actually a pity that book companies don't do the same as record companies. I have two books by Ammeral; *C++ for Programmers* and *STL for C++ Programmers*. Both very worthwhile books and both of which (in their time) have been frequently referenced. There is a lot of crossover between the two books (sorry Francis, I know you'd disagree with me here – but there is). What would be great is if there was a sort of "Best of" for these books. One volume without the crossover material, but all of the great information.

This idea could be applied to a number of other books – some of the XP ones have a good chunk of similar (not the same) material. A bit of rewording and instead of 4 books, it becomes 2. Less space occupied and more information for the page count.

All right, some books you would never dream of doing that to. Josuttis's *C++ Standard Library* being one of them. That book is just so crammed pack full, it would be pointless to try and merge it with (say) *C++ Templates* (which was co-written with Vandevoorde).

Dead Websites (or A Tale of Two Websites)

One of the most annoying aspects of any book is when they reference a website in the text or on the book itself. Now, I'm not that daft to imagine for one moment that a book company or a person's ISP is going to exist forever. However, book companies take over other book companies, so at least some material should still exist.

Case 1

Company 'A' have produced a lot of books. I have 7 of their books on my shelf currently. Their books make references to a number of websites, all of which are required to some extent to service the code in the book and in one particular book, an entire chapter is pretty pointless without one of the libraries listed.

The original company was bought out and the new company doesn't recognise the book as being one of theirs, leaving the person with a book which is practically useless for a couple of the most important chapters. There is a CD ROM with the book, but in a break with tradition, it is filled almost entirely with material that someone at the book's original company thought would be a good idea at the time.

The original author's website has vanished. Waybackmachine can't find the download and even Google draws a blank.

Case 2

Company 'B' publishes a book which is a couple of years old and hasn't been updated. The libraries referenced still all exist but have been greatly modified since the original release of the book. A person undertakes the job to update the codebase, tells both the author and the book company of the update and where it can be found. Company B takes a copy and posts it on the support area for the book and drop a quick email back to say thanks. The download is amazingly popular for both the company and the person who has done the update.

Company B publishes their books through another company.

I suppose there is only one thing worse than a company like company 'A' and that is one which has updates but the updates are broken and refuses to even email back to say "thanks,

but we're not going to fix it for reason a, b and/or c".

And So It Begins

By the time this edition hits the doormats, the new academic year in the UK Universities will be well underway. A fresh intake, all ready and eager to learn. Plenty of parents worry about their offspring being away from home for the first time and hoping they'll be fine.

While my child is only 6 (and so is not ready for University yet!), I can say that they will be. First year student life is a gas. Stop worrying – the worst they can do is have some really weird tattoo done and miss a couple of lectures.

With all of this spare time you now have as you've stopped worrying (a bit), what should you do? Watch TV? Listen to that collection of I'm Sorry I'll Read That Again you have on CD? Have a meal out?

Why not write for C Vu or Overload? We're always after new articles, book reviewers and contributors to the Student Code Competition. Let's make both magazines even bigger and better value for everyone!

Paul F. Johnson

View From the Chair

Ewan Milne <chair@accu.org>

The programme for the 2005 conference is rapidly taking shape, and we are very happy to be welcoming back Jim Coplien as a keynote speaker. Always a highly entertaining and challenging speaker, I can confidently say, even from the early drafts I have seen, that Cope's talks will be unmissable highlights of the event next year. Also lined up are Ross Anderson, leading off a track dedicated to security issues, and another big name whose appearance is still subject to final confirmation. But as a very broad hint, let's just say that there really isn't a bigger name in the C++ community.

As well as these heavy-hitting head liners, the conference needs new, first-time speakers. It is encouraging that we have had some very strong proposals for short 45 minute sessions from members, however as it stands we need a few more. The deadline for proposals, still ten days away as I write, will have passed by the time you read this – and it is possible that there will be a last minute influx. But if you had a rough idea for a 45 minute session that you didn't quite get round to sending in, please get in touch. It's quite possible that there is still a space for it in the programme. Remember, we are not looking for exhaustive explorations of large topics – an explanation of a useful development technique, or experience report of some aspect of a recent project would be ideal.

Urgently Required - Advertising Officer

At this year's AGM, Chris Lowe gave notice that he wishes to step down as Advertising Officer. Chris is generously still giving his time to carrying out this role as a replacement is sought, but the ACCU now has an urgent need for someone to volunteer to replace him. The post involves contacting potential advertisers to drum up business, and once contracts are in place, shepherding the advertising copy to our printers, Parchment.

This is a very important role, with a direct impact on the financial health of the organisation. If you're interested in this role, please contact either `ads@accu.org` or `chair@accu.org`.

Membership Report

David Hodge <membership@accu.org>

We are in the thick of the main renewal period (11 Sept). So far 74% of the membership have renewed, which is better than last year, so thank you for that.

Don't forget that next year you could have a £5.00 reduction on your subscription by paying by standing order. Note that this is not direct debit, so it requires you to set it up with your bank. If you want to do this just email me for details.

Standards Report

Lois Goldthwaite <standards@accu.org>

The programming world needs good standards. And we need them right now – or at least we think we do.

The benefits of standardisation are many and obvious: programs (and programmers!) become more portable across platforms, components from different sources can be integrated more readily, and (it is greatly to be hoped) a stable foundation is established for future enhancements not yet envisioned.

There are some disadvantages also, it must be recognised, but one prominent one is that standardising a technology too early may burden its users with features which turn out to be inadequate or even undesirable.

The most successful standards have been those that codified existing mature practice, such as the 1990 C Standard. But standards committees that are seduced into inventing technology on the fly, without adequate implementation experience, nearly always come to regret some of their decisions. An example is the C++ keyword `export` – though controversial, it was included in the C++ Standard to address issues which were very important to some vendors. But six years after the standard was adopted, only one compiler vendor has actually implemented `export` in a public release, and programmers in general have shown little demand for the feature.

One of Java's early attractions was its boast was that it hid the complexities of various platform APIs and provided a standard, portable, 'Write Once, Run Anywhere' way to perform I/O and create GUI screens. But these fundamental portions of the Java 1.0 library have been extensively revised in subsequent versions. I guess in making standards, as well as software, you'd better 'plan to throw one away.'

The ISO standards process is frequently criticised for being too slow, too bureaucratic, too tied up in red tape, not nimble enough to cope with an industry that moves in internet time. An increasing number of standards are being developed by trade groups and industry consortia working to aggressive schedules, partly because they aren't hampered by ISO's leisurely pauses for preliminary and final international ballots.

Which is better: to delay adoption and publication while working to refine the best possible standard, or to get something into the marketplace as soon as possible, with the intent to follow up with a frequent revision cycle? How long can a committee linger over a document before it becomes irrelevant? How soon is too soon to rush into print with a half-baked draft?

I don't think there is one right answer to those questions. But I do know that correctness and consensus take time to mature, and that stability is one of the primary benefits of having standards. If the standards themselves are changing every few months, industry can't keep up.

Experience has shown that it takes at least five years for new innovations to graduate to common practice. With that statistic in mind, I would argue that it's worthwhile for a standards committee to favour getting everything right over getting something out the door. Once a mistake is set in standards concrete, it becomes a boil on the industry bum for eternity. Even if it is deprecated in a later revision, vendors feel obligated to support it for backwards compatibility.

And the much-maligned ISO standards process, which forces pauses for review and reflection, has a lot to recommend it.

Dialogue

Student Code Critique Competition 30

Set and collated by David A. Caabeiro <scc@accu.org>
Prizes provided by Blackwells Bookshops & Addison-Wesley

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

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

Before We Start

It seems that praying for more participation among members is giving good results, but anyway, let's hope I don't have to repeat the plea issue after issue. Please note that you can participate not only by submitting critiques, but also by contributing code snippets you came across which attracted your attention. Remember that you can get the current problem set on the ACCU website (`http://www.accu.org/journals/`). This is aimed at people living overseas who get the magazine much later than members in the UK and Europe.

Student Code Critique 29 Entries

Looks like an ordinary snippet, doesn't it? Amazingly, it contains various mistakes for such a few lines. Please provide a correct version.

```
#include <iostream>
using std::cout;
using std::endl;

#include <list>
using std::list;

int main() {
  list<double>::iterator it;
    list<double> lst;
  *it = 34;
  *++it = 45;
    *++it = 87;
    it = lst.begin();
    for (;it < lst.end(); ++it){
      cout << it << '\t' << *it << endl;
  }
    system("pause");
  return 0;
}
```

From Tony Houghton <h@realh.co.uk>

There are three errors preventing the program from compiling. The first is that `<` is not a valid operator for an iterator and needs to be replaced with `!=`. Then `it` can not be printed to `cout`; we could print the the address of the data it references, but that's a useless piece of information in this context, so I think it's better to introduce a second variable called `n` showing the numerical position we've reached in the list. This is only meaningful as a cue for the user.

I've chosen to initialise it outside the loop and increment it in the loop body rather than in the loop statement to emphasise that the loop is iterating through the list and `n` is supplemental. The final compile error is that we haven't included `<cstdlib>` so `system` is undefined. The `pause` command is not portable anyway, and C++ makes such a meal of waiting for a user to press Enter that I've just deleted that line and left it up to the user to run the program in a shell or IDE that will give them a chance to read the output.

Even though it will now compile the code is still badly bugged and likely to crash. We're derefencing `it` without initialising it. Not only that, it is not possible to add elements to a list by writing off the end of it. We need to explicitly create a new element by appending to the list with its `push_back` method.

Here is my version of the code:

```
#include <iostream>
using std::cout;
using std::endl;
#include <list>
using std::list;

int main() {
  list<double> lst;
  lst.push_back(34);
  lst.push_back(45);
  lst.push_back(87);
  int n = 0;
  for(list<double>::iterator it = lst.begin();
      it != lst.end(); ++it)
    cout << n++ << '\t' << *it << endl;
  return 0;
}
```

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

The problem posed is to provide a correct version of the code.

The first question which needs answering is what is the purpose of this code? It looks very much like someone's first experiment with STL collections and iterators – but they don't really understand what they're trying to do.

Solution one:

```
#include "Josuttis/The C++ Standard
                        Library/Chapter 5"
```

There's little point simply fixing the code since the basic misunderstanding seems so great; a good tutorial/reference is probably the best place to start. However, if you insist...

Input

The code seems to be trying to fill a list using an iterator and then print out the list to verify that it filled properly.

```
11   *it = 34;
12   *++it = 45;
13   *++it = 87;
```

Unfortunately, although a list can be populated with an iterator, a standard `list::iterator` is not the right sort! There are classes of iterators which are designed to allow insertion, so we need one of those to insert either at the back or the front of the list. I'll pick a back iterator since that means the items will be printed in the order they're inserted, which seems more intuitive. So let's replace these lines with:

```
back_insert_iterator<list<double> >
ins_it(lst);
*ins_it = 34;
*++ins_it = 45;
*++ins_it = 87;
```

This code is perfectly OK, but there is a potential performance issue with using pre-increment. It's probably not worth worrying the writer of the code with this just yet (but you could refer them to one of the Effective C++ books...)

Now we'll need to include another header file, `<iterator>`, to be compliant with the standard and add a `using std::back_insert_iterator`.

On that note, programmers vary on their attitude to `using`. Again, I wouldn't worry this programmer too much about it at this point (unless company coding rules apply!) Now the code.

Output

The code to output the list relies on `operator<` for the iterator – this pretty well implies that the iterator is a random access iterator, such as an iterator over a vector. The normal paradigm for iterators in STL is to use `!=` for the loop condition.

The loop can be coalesced: currently we have 3 parts to the loop control structure:

a) Declaration
```
list<double>::iterator it;
```
b) Initialisation
```
it = lst.begin();
```
c) Condition and update expression
```
it != lst.end(); ++it
```

I'd recommend putting these all into the `for` statement for clarity and to reduce the scope of `it`:

```
for(list<double>::iterator it = lst.begin();
    it != lst.end(); ++it)
```

or, since we're trying to output from the list only,

```
for(list<double>::const_iterator it
                                = lst.begin();
    it != lst.end(); ++it)
```

Again, more advanced techniques to avoid the `for` loop completely are possible but would be likely to simply confuse the programmer at this point. Now, what are we actually outputting inside the loop? The code as written tries to print the iterator itself and then its contents. By analogy with 'iterator is a generalised pointer' I guess the purpose is to display the address of each item and its value. However there's not a standard `operator<<` defined to do this – the easiest solution is to use the `&` operator:

```
cout << &*it << '\t' << *it << endl;
```

or possibly, for clarity, use a helper variable:

```
const double &value = *it;
cout << &value << '\t' << value << endl;
```

Now we're almost done... on a Microsoft compiler on Windows anyway :-) The last statement, `system("pause")`, is target environment dependent. This might be fine and if so I've no problem with do the job like this. I might like to include `<cstdlib>` of course, since `system` currently works because, on my version of the standard library, one of the other header files is pulling in `<cstdlib>`.

If the code has to be portable then you'd need to replace it with something equivalent (or nearly so) from the C++ library. I'm assuming the code is fine on the target OS. And that's it – to get the code working anyway. Explaining the changes – and in particular the two types of iterators needed – might be a little more work!

From Roger Leigh <rleigh@whinlatter.ukfsn.org>

Overall, the intentions of the author are obvious, but it is clear that some misunderstandings over the use of containers and iterators resulted in non-functional code. The use of headers and `using` statements was fine, and the general structure of the code was also acceptable, bar two lines that required more indentation.

The first major error is with the use of iterators. When assigning values to the list, the iterator is not initialised and so is invalid (cannot be dereferenced). To compound the error, on the second and third assignments, the iterator is incremented in addition to dereferencing. All these mistakes will result in undefined behaviour.

The `push_back()` method is probably what the student wants. It looks like there is some confusion over how iterators work. The student needs to understand that iterators "point" to items within a container, and that they are not by themselves responsible for inserting values. Likewise, while an iterator can be incremented, this is only useful when there is a next element in the container, and error checking should be done to check that the new iterator is valid. Like pointers, they need to point to a valid location, and only then may they be dereferenced to access the contained value. For insertion, an iterator would typically only be used when inserting in a specific location in the list (used with the `insert()` methods), or when using an insert iterator, neither of which are applicable in this case.

The `for` loop iterator is initialised outside the `for` statement. While valid, it's not necessary and is bad style. Also, the `for` loop conditional uses `it < lst.end()` rather than `it != lst.end()`. Not all iterators implement `operator<`, but all implement `operator==` and `operator!=`. We only need to know if we are at the end of the list. When outputting the list contents, the iterator is output to an `ostream`, which is not possible (the operator is not implemented). Iterators do not have a (user-visible) index or a meaningful address, and so if the elements should be numbered, a suitable container should be used (e.g. a vector, which allows access by index), or the numbering should be done "by hand". `std::endl` is used after outputting each element. This adds a newline and flushes the `ostream`. The flushing is unnecessary, and would have a negative impact on performance when outputting the contents of a larger container. `'\n'` is adequate here. A more general issue is the use of `list<double>`. The numbers could more easily be stored in an `int`, or `short int`. I would also have used a `vector<int>` myself, given that the additional features a list provides are not used, and impose a greater overhead than a vector (e.g. memory usage).

Lastly, `system("pause")` is both system-dependent (non-portable) and mostly useless. I've only come across its use in Windows environments in order to stop the terminal window closing on program exit. This won't work on platforms without a `pause` command (e.g. UNIX), and is a terminal configuration issue, not something to "fix" in the program code itself. The solution is to configure the terminal window not to close on exit, or to run the program directly from the shell. A version of the code rewritten to take the above into consideration follows:

```
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
using std::vector;

int main() {
  vector<int> coll;
  vector<int>::iterator pos;
  coll.push_back(34);
  coll.push_back(45);
  coll.push_back(87);

  int n = 0;
  for(pos = coll.begin(); pos != coll.end();
      ++n, ++pos)
    cout << n << '\t' << *pos << '\n';

  return 0;
}
```

From Nevin Liber <nevin@eviloverlord.com>

Where to begin, where to begin...

Syntax error #1:

```
cout << it
list<double>::iterator it;
//...
cout << it //...
```

`it` is an iterator, not a pointer, and there is no standard way to output `it` to a stream. Guessing here that the intent was to display the address of the element in question, the following will work:

```
cout << &*it //...
```

The dereference `operator*()` is called on the iterator, returning a reference to the element. Then the address-of `operator&()` is called upon that, yielding the address of the element. Note: if the container were of a user defined type instead of `double`, this idiom might not work if the user defined type overloaded `operator&()`.

Syntax error #2:

```
system("pause")
```

`system()` is not a built in function. One way to get its function prototype would be to `#include` a header which contained it, such as `#include<cstdlib>`. Typically, `system("pause")` would call another program called `pause`. While not a syntax error, my computer does not contain such a program, and rather than guess at its semantics (wait for a certain amount of time, wait for a key to be pressed, wait for a signal, etc.), I'm going to leave it out for the rest of this discussion.

Syntax error #3:

```
it < lst.end()
```

Iterators are not pointers. `list` in particular has bidirectional iterators, and should only be compared for equality (`operator==()`) or inequality (`operator!=()`). Note: not all compilers will catch this at compile time, depending on how its particular implementation of `list<T>::iterator` was written. Whether or not it is a syntax error, it is definitely a semantic error, and the fix is

```
it != lst.end()
```

Now that we are done with syntactical errors, on to the purely semantic ones.

Semantic error #1: what list does `it` refer to?

Looking at our declaration:

```
list<double>::iterator  it;
list<double>            lst;
```

`it` does not refer to any particular list. I'll actually fix this later, since it won't be needed in the fix for the next three lines of code anyway...

Semantic error #2: `*it` does not synthesize space in the list

```
*it = 34;
*++it = 45;
*++it = 87;
```

This is one of the most common errors I've seen when people start using the standard containers. I believe the intent here is to have a list of three items. However, `*it` is illegal, not only because it doesn't refer to `lst`, but even if it did, `lst` started out empty, and the dereference of any iterator into an empty collection is illegal.

Since there are only three elements, the simplest way to create this is:

```
lst.push_back(34);
lst.push_back(45);
lst.push_back(87);
```

Where `push_back()` places the element on the very end of the list. Note: there is an implicit conversion of each of these numbers from type `int` to `double`, which may or may not happen at compile time.

Since `it` isn't actually needed until the `for` loop, just declare it there.

Before doing so, I'm going to add the following `typedef` to the beginning of `main()` for a little bit of defensive programming:

```
typedef list<double> Collection;
```

The reason is that the iterator always has to match the type of the collection, so if you need to change this, it only has to be changed in one place. Now, one should pick a better name than `Collection`; however, better names will not include the word list or double, as that is what we are trying to abstract away. Names should reflect what something is for, not how it is implemented.

The `typedef` is placed inside `main()` itself because no one outside of `main()` cares about it. Always limit scope as much as possible.

Putting this all together:

```
#include <iostream>
using std::cout;
using std::endl;
#include <list>
using std::list;
#include <cstdlib>  // for system(char const*)

int main() {
  typedef list<double> Collection;
  Collection lst;
  lst.push_back(34);
  lst.push_back(45);
  lst.push_back(87);

  for(Collection::iterator it = lst.begin();
      it != lst.end(); ++it)
    cout << &*it << '\t' << *it << endl;

  system("pause");
  return 0;
}
```

Planning for a reasonable future

At this point, we could be done. The above code works, is simple and straightforward, and does the job requested.

However, if this were the foundation of a bigger project, I might add a little more complexity for future flexibility. This is ultimately a judgement call; do too little, and you have under engineered the solution; do too much, and you have over engineered it. That is why I say plan for a reasonable future, and not all futures.

Future: Data driven initialization of `lst`

A future that I consider reasonable is one where the number of initial elements may be larger than 3. `push_back()` works fine for 3 elements. But what about 30? Or 300?

Making it data driven is more scaleable. The data driven way is to initialize the list from an array, as in:

```
static const Collection::value_type
initialElements[] = {34, 45, 87,};
Collection lst(initialElements,
initialElements + 3);
```

Note: I prefer using `Collection::value_type` over `double` for the type of the elements in the array to emphasize that the type here should correspond to the type of the elements in `lst`. By `typedef`ing `Collection`, I have self-documented that there is a relation between `initialElements`, `lst`, and `it`. While iterators are not (necessarily) pointers, pointers can be used in place of iterators, and `list` has a templated constructor that can take any type of input iterator. There still is the matter of `initialElements + 3`. It is less error prone to have the computer count the number of elements than for me to count it. Using a C idiom:

```
Collection lst(initialElements,
       initialElements
         + sizeof initialElements
               / sizeof initialElements[0]);
```

While I can do this without any helper functions, it is still error prone. If `initialElements` was a pointer instead of an array (which could happen if this code had changed to pass in a pointer to an array of initial values), the calculation would be wrong, yet the code would still compile and run. To solve this, I have a set of templates that always gets this calculation right:

```
#include <cstddef>  // for size_t
template <typename T, size_t N>
inline size_t size(T (&)[N]) { return N; }
template <typename T, size_t N>
inline T* begin(T (&a)[N]) { return a; }
template <typename T, size_t N>
inline T* end(T (&a)[N]) { return a + N; }
```

Basically, they make an array look like a collection, with `begin()`, `end()`, and `size()` functions (although they are free functions, not member functions). The array is passed in by reference to these functions, and uses template argument deduction to determine the number of elements

in the array. Note: if we pass in a pointer instead of an array, it is a compile time error. In this case, they are used as follows:

```
Collection lst(begin(initialElements),
               end(initialElements));
```

Note: technically, instead of `begin(initialElements)` as the first iterator, I could have passed in `initialElements` directly, as the array will decay into a pointer when passed by value. I prefer the former, both as it is self-documenting, and I get an extra level of checking that `initialElements` is an array. Combining all of this, we get the following solution:

```
#include <iostream>
using std::cout;
using std::endl;
#include <list>
using std::list;
#include <cstdlib>  // for system(char const*)
#include <cstddef>  // for size_t
template <typename T, size_t N>
inline size_t size(T (&)[N]) { return N; }
template <typename T, size_t N>
inline T* begin(T (&a)[N]) { return a; }
template <typename T, size_t N>
inline T* end(T (&a)[N]) { return a + N; }

int main() {
  typedef list<double> Collection;
  static const Collection::value_type
initialElements[] = {34, 45, 87,};
  Collection lst(begin(initialElements),
end(initialElements));
  for(Collection::iterator it = lst.begin();
it != lst.end(); ++it)
    cout << &*it << '\t' << *it << endl;
  system("pause");
  return 0;
}
```

From Mick Brooks <michael.brooks@physics.ox.ac.uk>

This is my first attempt at an SCC solution, so I may learn more than our student. Anyway, here goes:

Trying to compile your code, GCC flags the line with the `for` loop as an error. Trying to evaluate `it < lst.end()` is the problem, since the `list::iterator` is a bidirectional iterator, and so doesn't have the less-than operation defined. It would work if we used a vector container instead of the list, since `vector::iterator` is a random-access iterator which is more powerful, and has more operations defined for it, including one for less-than. In order to make this loop do what was intended, we can look to a C++ idiom for help: use `!=` as the loop condition check. This operation is defined for all of the five iterator categories, and so will work for iterators over any of the standard container types. The loop still looks unusual, because the idiomatic style is to make use of the initialiser part of the `for`-loop syntax. The maintenance programmer (which could be you in about 6 months) will see that combination of `(;` and will needlessly have to wonder if that's a mistake. Make it explicit, and put in the initialization. This has the wonderful side-effect of limiting the scope of the `it` variable. While we are here, we can notice that you don't modify the value pointed to by the iterator `it`, and can make that explicit in the code by using a `const_iterator` instead. So now the loop looks like this:

```
for(list<double>::const_iterator it =
lst.begin();
    it != lst.end(); ++it)
cout << it << '\t' << *it << endl;
```

which is made much clearer to a C++ programmer through its use of the standard idioms.

Unfortunately, the code still won't compile; this time because there is no output operator (`<<`) defined for the `const_iterator` type. I assume that the intention of `cout << it` was to print the address of the object pointed to by the iterator. This might just happen to work if list iterators were actually pointers, but they are more complicated than that. What I think you want here is `&*it`, which is the address-of operator applied to the result of dereferencing the iterator, and gives us the memory address of the `double` that is pointed to by the iterator.

Well, now the code will compile, but running it gives a segmentation fault on my Linux machine, which tells us that we aren't finished yet. This is due to using an iterator to access memory that we don't own, and means that there's some work to be done on understanding how to create our list. In the first line of `main()`, you define the iterator `it` but don't give it a value, which leaves it in an unknown state. Trying to dereference that iterator is a big mistake, which causes the segfault. The iterator would have to be made to point to a valid member of a list before we can use it, but there are no valid members of an empty list. We have to find another way of populating the list. My preferred solution would be to use `push_back` on the list and drop the iterator altogether, which leaves us with the following working code:

```
#include <iostream>
using std::cout;
using std::endl;
#include <list>
using std::list;

int main() {
  list<double> lst;
  lst.push_back(34);
  lst.push_back(45);
  lst.push_back(87);

  for(list<double>::const_iterator it =
lst.begin();
      it != lst.end(); ++it)
    cout << &*it << '\t' << *it << endl;

  system("pause"); // if we must...
  return 0;
}
```

As an aside, if you really want to use an iterator interface to do the job, you'll have to learn about Insert Iterators. A `back_insert_iterator` will call `push_back` for you, and you can then fill the list with the iterator interface, like this:

```
#include <iterator>
using std::back_insert_iterator;
// other includes, as before

int main() {
  list<double> lst;
  back_insert_iterator<list<double> > it(lst);
  *it = 34;
  *it++ = 45;
  *it++ = 87;
  // ... as before
```

I'm not sure what this buys you though, except that you get to learn about the added confusion that `it++` and `++it` are no-ops, and so both the increments in that snippet could be dropped.

From Terje Slettebø <tslettebo@broadpark.no>

First, some comments about style, before we examine correctness. The program starts with:

```
#include <iostream>
using std::cout;
using std::endl;
#include <list>
using std::list;
```

Now, in this case, for a source file (not header file), it's generally safe to do:

```
using namespace std;
```

instead of the `using`-declarations. Any name clashes will be flagged by the compiler, and you may save yourself considerable amounts of typing this way. I know this is a hot topic, but anyway. Another alternative is explicit qualification of the names in the code: `std::cout << it`.

The code has a weird indentation, with some lines indented a couple of places for no apparent reason at all. This gives the code a messy/untidy look, and clarity **is** important; unclear code is a good breeding ground for bugs. Moving beyond pure layout, there are some other general comments that can be made:

1. It's a good idea to initialise the variables at the point of declaration, if possible. This avoids the chance of accidentally accessing an uninitialised variable. This is actually happening in the code (and that's just one of the bugs): `it` is declared but not initialised, and then subsequently used, leading to undefined behaviour.
2. Also, you should not "reuse" a variable for a different purpose, as the variable `it` is in the code (it's reused for the loop) (this is a **bad** case of reuse. ;)).
3. Keep scopes as small as possible. If `it` is declared in the `for`-loop, it only exists in the loop, and you avoid accidentally using it after it should no longer be used.
4. Moreover, you may want to use `const_iterator`, rather than `iterator`, when the code doesn't alter the container (despite what Scott Meyers may say about preferring `iterator`).
5. The naming used in the code is not very good, to say the least. Names should generally be chosen based on the **role** of the variable, not its type (although some Hungarian Notation like `..._ptr` might be acceptable, as it reminds us that it requires a different usage). In the code, the words `lst` and `it` are used. Avoid acronyms and abbreviations, unless the name might be rather long without it, or the acronym or abbreviation is well-known. However, this code doesn't just have bad style, it also have some real bugs (including the one mentioned in point 1, above):
6. When the iterator `it` is assigned to, even if it had been a valid iterator to the start of the container, the container is empty, so the iterator is the past-the-end iterator. Assigning to and incrementing `it` leads again to undefined behaviour.
   The problematic assumption in the code seems to be that the programmer thinks assigning to the iterator, and incrementing the iterator, will insert the values into the container. It is not so. You have to explicitly insert the values using the container object, for example with `push_back`:

```
lst.push_back(34):
lst.push_back(45);
lst.push_back(87);
```

7. One small thing to note here is that there's an integer to floating point conversion when the values are inserted, but it gives the expected behaviour. To avoid it (and its possible overhead), you might use values of type `double`, instead: 34.0, etc.
8. The list iterator doesn't have the less-than operator defined, only equal and not equal, so the program won't compile as it is. It may be recommended to always use not equal, even for containers supporting less-than, for consistency, and stronger post-conditions to the loop (detecting bugs earlier).
9. The body of the `for` loop tries to print out the iterator, which fails, as there isn't a stream operator defined for it. The intention was possibly to write out the index of each element. This isn't available from the list, so you have to track it separately, if you need it.
10. One final point to note is that `system()` is not a standard C++ function (and there's no other header to include it in the program), so even if the right header was included, the program wouldn't be portable to systems lacking that function.
11. `endl` is a manipulator that, in addition to writing a newline to the stream, also flushes it, and you may want to avoid needless flushing, especially if it's done a lot. The output stream is flushed before any input, anyway. (Ok, so **this** was the final note.) Let's say the list is a list of percentages. Here's a possible corrected version of the program (the `using` part has several correct possibilities, as mentioned):

```
#include <iostream>
#include <list>
int main() {
  std::list<double> percent_list;
  percent_list.push_back(34.0);
  percent_list.push_back(45.0);
  percent_list.push_back(87.0);
  for(percent_list_type::const_iterator it
                      = percent_list.begin();
      it!=percent_list.end(); ++it)
    std::cout << *it << '\n';
}
```

This corrects the problems mentioned in points 1-11.

If you thought I would stop here, you don't know me well enough. ;) Let's step back, and try to see what is the **intent** of the code. The code should express the intent as clearly as possible. Well, does it? Let's find out. The code inserts few values into a list, and then prints them out. The code above says quite a bit more than this. One thing that is commonly mentioned is to use `for_each`, rather than `for`, in such situations. However, using only the standard library, you need to then create another class to do the printing. This isn't necessarily an improvement, as you can't define the class at the point of use. However, as Kevlin Henney shows in his "Omit Needless Code" article, there are several alternatives to printing out the values. One is to use stream iterators:

```
typedef std::ostream_iterator<double> out;
std::copy(percent_list.begin(),
          percent_list.end(),
          out(std::cout,"\n"));
```

This makes it rather more succinct. Now, the code focuses on **what** to do – printing or copying the values to the output stream – rather than **how** to do it.

If you need more advanced formatting (such as enclosing each value in braces), this won't do, though. Fortunately, there are libraries that allow us to create function objects on the fly, usable with algorithms, such as Boost.Lambda [1]. With it, we may substitute the above with:

```
std::for_each(percent_list.begin(),
              percent_list.end(),
              std::cout << _1 << "\n");
```

That takes care of the printing. What about the insertion of values? That looks rather repetitive, doesn't it? Well, Boost can again help us, here, with the Assignment library [2]:

```
percent_list+=34.0,45.0,87.0;
```

Here's the last revised version of the code:

```
#include <iostream>
#include <list>
#include <algorithm>
#include <boost/assign/std/list.hpp>
#include <boost/lambda/lambda.hpp>
using boost::assign::operator+=;
using boost::lambda::_1;

int main() {
  std::list<double> percent_list;
  percent_list+=34.0, 45.0, 87.0;
  std::for_each(percent_list.begin(),
                percent_list.end(),
                std::cout << _1 << "\n");
}
```

Now, there's no fluff; the code states what it does (at least when you learn the abstractions involved). A further improvement might be if there were overloaded versions of the standard algorithms taking containers, rather than iterators:

```
std::for_each(percent_list,
              std::cout << _1 << "\n"):
```

David wasn't kidding when he said the code contains "various mistakes for such a few lines"... This small snippet also turned out to be a good

opportunity to demonstrate some software development fundamentals, as well as more advanced techniques.

[1] http://www.boost.org/libs/lambda/doc/index.html

[2] Available in the CVS, but not yet in the current release.

The Winner of SCC 29

The editor's choice is:

Mick Brooks

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

Francis' Commentary

```
#include <iostream>
using std::cout
using std::endl

#include <list>
using std::list;
```

Let me start with a small style issue. I do not like interspersing `using` declarations and headers. I like to see all the `#include`s up front. Preferably I like to see any user header files first (in alphabetical order) followed by the necessary standard headers (also in alphabetical order). Placing the user header files first avoids accidentally masking a dependency that should have been in visible in the user header. Placing the includes in alphabetical order just makes it easier to check whether one has or has not been included.

While I notice, `std::endl` is not required to be declared as a consequence of `#include <iostream>`, it usually is but only because `iostream` normally drags in `ostream`.

When it comes to `using` declarations and `using` directives I think we should tend to use fully elaborated names (i.e. do not use either `using` directives or `using` declarations) until the user knows enough to understand the implications of each. I know this is contrary to what I did in 'You Can Do It!' but the motive in that book was to get inexperienced programmers writing code so I was willing to make some sacrifices. However, even there I started with fully elaborated names and required that they be used in all reader written header files.

```
int main() {
  list<double>::iterator it;
    list<double> lst;
```

I cannot say that I am enamoured by the choice of `it` as a name for an iterator but I can live with it, but the choice of `lst` as a variable is beyond my tolerance levels (well it is today). And what is that extra indent for? Indents without purpose only serve to confuse.

Up until now, I have just being cantankerous. Now it is about to get serious:

```
*it = 34;
```

Do you know if `list<T>::iterator` has a default constructor? No, neither do I and I do not care to take time to look it up. Whether it does or not, `*it` is surely introducing undefined behaviour because `it` has never been initialised to point to any storage.

And now it gets worse:

```
*++it = 45;
   *++it =87;
```

More purposeless indentation coupled with incrementing what is, at best, an iterator that points nowhere. And only now does the student make any attempt to relate it to `lst`. Had the program no had multiple instances of undefined behaviour the next line would have been perfectly OK, just not exactly the idiomatic way to do it.

```
it = list.begin();
```

Time to wind back to the beginning and write the code properly by avoiding early or unnecessary declarations.

```
#include <iostream>
#include <list>

typedef std::list<double> list_of_double;

int main() {
  list_of_double data;
```

Note that there are no `using` declarations, but I have used a `typedef`. That is often a much more useful tool, and one that provides a modicum of documentation. In fact it is the exact reverse of `using` declarations because it adds information (not much in this case, but the problem is pretty abstract) rather than removing it (the library that a name belongs to).

Next the list starts empty so there is nowhere to store values. My preferred choice is to use `push_back()`, however we could have created the object data is bound to with three default initialised nodes by changing the definition to `list_of_double data(3);`. Sticking with my preferred option we get:

```
  data.push_back(34);
  data.push_back(45);
  data.push_back(87);
```

The reason that I prefer this option is because it makes it very clear to even the rawest novice that data has nothing in it until we start pushing things in. If teaching, I would break off here and have a brief discussion as to what `push_back()` does.

Having created a list of values, I am ready to write them out:

```
  for(list_of_double::iterator iter
                              = data.begin()
      iter != data.end(); ++iter) {
```

You did notice that the student had used the wrong comparison? `std::list::iterator` is not a random access iterator and so values of it are not ordered. We simply cannot use a less than comparison. The only thing that will work is to keep going until equality with the end marker is achieved. Less than will work for most of the sequence containers but not for this one. Comparison for inequality is idiomatic for C++, those that want to do something else should understand why sticking with idioms is helpful.

```
    std::cout << &*iter << '\t' << *iter
              << '\n';
```

I am guessing that the student wants to see the address used to store the value. If he didn't then he is completely out of luck because there is no requirement that a value of a `std::list::iterator` object be acceptable to an `ostream` object. The standard technique for getting the address of an object in a container is to apply the address-of operator to the dereferenced value of the iterator for the object; another idiom of modern C++.

Another feature is that I do not use `std::endl` unless I actually want to force both an end-of-line and a flush to output. I only need an end-of-line so I use the correct character for that; `'\n'`.

Now to finish:

```
  }
  std::cout.flush();
  std::cin.get();
  return 0;
}
```

Now I force the flush by calling the correct function. I don't want to hunt around to see which header declares `system()` and I certainly do not want to introduce that kind of system dependence into my code unless I really have to.

There are several other things that I might do to polish this program a bit, but I think the above will do. Now I wonder how the rest of you got on, and how many things I missed. The sad thing about much of the code we see here is that it shows just how badly instructors are explaining what is happening. Most of the code we publish comes from students who really want to get it right rather than ones who went to sleep during the lectures. The kind of errors they make expose fundamental misunderstanding of C, C++ etc.

[Code for SCC 31 at foot of next page]

Letters to the Editor

Quite a few emails in this edition. As always, I welcome your comments about C Vu, the articles and about the ACCU in general, keep them rolling in!

Student Critique

On reading the various answers for the code critique competition 27 in C Vu Vol 16 No 3 (yes, I'm an issue behind at the moment!), involving "ugly" numbers, I feel I must point out what I find to be some poor advice from Francis in his commentary. He develops a solution, as most of us did, with an `IsUgly()`, or `is_ugly()` function. This he gets to a near working stage, i.e. it detects numbers whose prime factors are not all 2, 3 or 5. Then he realises that the spec said that ugly numbers have to have at least *2* prime factors, i.e. they cannot be 2, 3, and 5 themselves. Fine, except for the implementation. He realises that he need not worry testing numbers below 6, oh, except for 4, because that is 2x2.

I can't decide whether this is a form of premature optimisation or not. Either way, the most significant problem is that we've now introduced the literals 4 and 6 into the code. Why? Because 6 is bigger that the biggest of the ugly prime factors, and 4 is the only number between them which can be made by more than one factor of one of them. That to me is not the kind of complicated derived logic you should have in code.

Suppose we made the definition of ugly such that its prime factors were 2, 7, 23. Now, what happens to the 4 and 6? Hmm, we need to tackle it generically this time, or else make special cases of 4, 8, 14, 16.

Simon Sebright
simonsebright@hotmail.com

I put this to Francis and had this back..

We can spend an awful lot of time trying for generic solutions to specialised problems. In my opinion the time for considering generic solutions is when we are actually presented with several similar problems. At that stage it may be worth looking for a suitable abstraction.

I contend that human time is one of the most expensive resources and we need to develop work habits that reduce that cost. We can argue interminably about how to do this and that in itself is a waste of precious human resources. The degree to which a solution is specialised will always be a judgement call and as such different people will draw the line in different places.

By the way, you see the logic of my code as complicated, I don't but here again we differ and I am certainly not going to lose any sleep about such differences of opinion.

Which in turn led to...

Of course, there is always a line to be drawn on the amount of time you can scrutinise a piece of code. In a production environment, there'll be thousands or even millions of lines in a project, and spending half an hour looking at 40 lines of code could have a dramatic effect on productivity.

But, in the context of a student code critique (real or ficticious), I think investing time is a good thing. Not discouraging bad habits or shortcuts at this stage has to be a bad thing. Making the novice aware of more expressive, or generic ways of doing things gives them a greater toolkit later on, and then it's going to be more likely that future code reviews won't need such fine-scale attention. I've seen plenty of these kinds of shortcuts cause bugs later in a program's life, and they can be very hard to track down.

I'm increasingly of the opinion that maintenance starts the moment you hit the compile button, not just two years later. The current project I'm on has about 1 million lines of code, has been in development for 3-4 years, is on the third or fourth generation of programmers, and won't be finished and out of the door for a number of months. Equipping students with the awareness of issues in these environments is very important.

Applying that to this example, I'd rather see logic or assumptions in the code than the comments. E.g.

```
const int small_ugly_exceptions[] = { 4 };
const int smallest_possible_ugly = 6;
```

and then we don't need comments. I note that there was no comment on the `< 6` test, which I'd like to see a put in in a production environment (except that I advocate coding it so comments are not necessary).

I'll leave it there. As always, if you wish to comment, feel free.

Fortran and Professionalism in Programming

I am writing to you as Chairman of the British Computer Society Fortran Specialist Group as well as a member of the ACCU of 10 years standing.

I would like to expand on your mention of restrictions on coding style and formatting imposed by Fortran 77 in Pete Goodliffe's article in C Vu Volume16 No 4 and to let the readers of C Vu know that Fortran is alive and well in the 21st century.

I spent 15 years writing and maintaining Fortran 66 and 77 code. While the fixed format source form put some restrictions on the layout of program

Student Code Critique 31

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

Here is a program Francis collected which is riddled with poor design, naming, etc. as well as the actual problem:

I'm getting a "parse error before `else` *" at the line indicated by the arrow*

```
void IS_IT_A_DDR(string& mtgrec,
                 string& temprec,int& ddrrc) {
  string Day2="SunMonTueWedThuFriSat";
  string Daytoken="0123456";
  int badday=0;
  if (mtgrec.size() < 8) {
    ddrrc=0;
    return;
  }
  for (int i=0; i <= 6; i++) {
    if (mtgrec.substr(0,3)
              == Day2.substr((i+1)*3-3,3)) {
      if ((mtgrec.substr(3,1) == "0")
            || (mtgrec.substr(3,1) == "1")) {
        if ((mtgrec.substr(7,1)).
          find_first_of("BCLMOPSTW*") != -1) {
          temprec=Daytoken.substr(i,1)
                                + mtgrec.substr(1);
          ddrrc=1;
          return;
          }
            else {
          ddrrc=2;
          return;
          }
        else { <<< compiler diagnostic
          ddrrc=3;
          return;
        }
        }
      }
      else badday++;
      }
      if (badday == 7) {
        ddrrc=4;
        return;
      }
      else ddrrc=5;
      return;
    }
```

statements, they had to be within columns 7 to 72 of each line, there was no restriction on using indentation to show code structure and you could continue long statements over several lines. If I remember correctly the standard allowed for 9 continuation lines but many compilers allowed up to 99.

The next revision of the ISO standard, Fortran 90, introduced the free format source form, where there was no restriction on the positioning of statements on a line other than a maximum line length of 132 characters, which some compilers increased. It introduced more modern features to complement Fortran's well known strengths in numerical computation. Fortran 90 and 95 introduced operations which could be carried out on whole arrays or sections of arrays, rather than just on individual elements. Also dynamic memory allocation and abstract data types were introduced.

The most recent version of the language, Fortran 2003, is due to be published in the autumn of 2004 and contains features to enable object orientated programming to be carried out in Fortran. For more information on the development of Fortran standards since Fortran 95 please see the Standardisation page on the Group's web site at `http://www.fortran.bcs.org/standards/stanhome.htm`.

If anyone is interested in exploring the modern features of Fortran a version known as F has been developed. F is a subset of Fortran 90/95 that enforces correct coding practices by removing antiquated and dangerous features in F90/95. There are new versions for Linux, Solaris, and Windows available for free download from `http://www.fortran.com`.

In response to some of the questions posed in Pete's article I can say that I tried to code in a consistent manner when writing new code, using 2 spaces for each level of indentation in both Fortran and C, and to "improve" the layout and structure of existing Fortran code when I had to modify it and had the time for cosmetic changes!

From my own experience I agree with Pete that tabs should not be used for indenting. We were programming across several platforms, each of which had its own editor, which handled tabs differently so that tab indented code could look OK in one editor but be almost unreadable in another. I aimed to globally replace all tabs with 6 or 8 spaces whenever I came to work on a tab-indented file. This was possible because we were only a small team, three to five developers, and we each tended to work on a particular area of the code so formatting changes did not often get changed back by someone else!

I should like to take this opportunity to say how much I have enjoyed Pete's articles on Professionalism in Programming over the last four years. I have found relevant and informative points in every one.

While writing about professionalism I would like to remind members of the ACCU that the British Computer Society undertook a major relaunch earlier this year using the slogan "Making IT the profession for the 21st century" and aimed at making individual membership more relevant to professionals in IT. See `http://www.bcs.org` for more information.

Peter Crouch
`pccrouch@bcs.org.uk`

Book Reviews

The proposed change to the book reviews was enough for this email.

On the book reviews/ratings etc. discussion, I think that the base issue is what a book rating is for. One of the main reasons for such a rating is for someone to chose which book might be a good investment for some particular purpose (e.g. learning, reference etc).

If we give ratings on books (i.e. just a number or a conclusion separately from the full review) it would generally be used so that people can find the excellent books quickly. (Why would you want to know whether a book is average, or really, really bad? You should probably avoid it anyway. If you inherit it, you might want to read the entire review to find out what is right/wrong with it.). Perhaps we would have to qualify what the review rating is designed to be used for.

The meaning of 'excellent' is probably going to be different depending on who you are (super-expert/beginner) and quite possibly what you are going to do with it (games programming v. financial applications v satellite control. Reference, or discussion of finer points of syntax? etc.). Also, what one would consider excellent would be expected to change with time. (What would the original K&R book on C rate as 15 years ago? And what today?)

I think that rather than attempting to rate all books, an ACCU rating of books that we would consider indispensable might be useful. This could represent a general consensus of the membership, rather than just a single reviewer, or even a review panel. (As lots of people would have read Stroustrup, Meyers and a host of other top-rated books it would not involve a huge amount of postage or even necessarily of organisation). It would be then be reasonable to review this list once a year, to see if there are any missing or ones that should be removed from this list. The number of times books crop up in references in Overload might be an interesting place to start.

How many books should be on the list? Possibly only 10 core C++ ones – and another 5 or 10 for specialist purposes (and an appropriate number for other languages)

It would also mean that we might be able to supply different people's opinions and any caveats on the books – which would be interesting reading in its own right.

There are almost certainly problems with this scheme. Perhaps there are other reasons people like to have ratings. Perhaps this could be just an adjunct to the existing book review rating scheme (recommended, highly recommended etc).

James Roberts
`James.Roberts@logicacmg.com`

Thanks for such a great email which more or less reflects what I've been saying for quite a while!

The point over what constitutes the ratings is something which does have to be ironed out.

As you're aware, we have 4 ratings; not recommended, nothing, recommended and highly recommended, with recommended being like a grade 2 degree (2i or 2ii – recommended or recommended with reservations). There is nothing to say what has to done to achieve one of these grades.

What has been proposed is that the reviewers have a set of criteria to judge the books against. It is not a tick list as it still allows for the reviewer to use his/her judgement – I have reviewed some books which while technically correct, have been written so badly that their use is very limited. A simple tick system would have gained it (say) a recommended, but the judgement would drop it down.
In lay terms:

Highly recommended : It's been written by Stroustrup, or Josuttis
going down to

Put it back on the shelf or if you've bought it, demand a refund : anything in the "for dummies" series, Schildt or "C++ in 21 days" type books.

Of course, the review system is still in the early days domain, so what will happens is still to be determined.

And Finally...

Having just read the 'Time for Change' segment by Francis Glassborow, I realised how much I resembled the description! ACCU has changed considerably since its inception, and the change in the Committee make-up does reflect the change in balance of the membership. Back then C++ had not made it out of the laboratory and C did not have a standard... hmmm.

These days I certainly do not do much programming, and essentially none of it in C/C++/Java/Python. My work is all systems administration, which means about 50% security. I keep with ACCU for several reasons. It is interesting, and I suppose I have a proprietary interest of sorts having spent a few years doing administration for the organisation. I don't begrudge the fees because I think I still get value, and the organisation deserves the support.

I wouldn't object to management and administration items, but I don't think it should be at the expense of the current design/coding bias. In fact, principles in software design are definitely valuable to anyone. Explaining to management why there is so much 'thinking time' in any project is a perennial issue.

I wonder if the matter of book reviews being ACCU or individual opinion may be arising because ACCU is succeeding in being considered as a serious organisation. The comparable commercial journals – and I think we can make that comparison now – are entities with staff writers for this purpose. The ACCU reviews are done by individuals with either expertise in the subject, or a desire to gain that expertise. Some may be plain curious. But that is a very real audience. I sometimes wonder if publishers do not set themselves up with the cover synopsis. A book may meet the expectations raised by the synopsis, fall short, or exceed them. And that is before comparison of content with competing books and current standards and practice.

Graham Patterson

If you'd like to send me a letter or email (I'm happy to get either!), please drop me a line to `editor@accu.org` *– you can send post to the address at the front of C Vu.*

Paul F. Johnson

Francis' Scribbles

Francis Glassborow <francis@robinton.demon.co.uk>

Professional What?

Pete Goodliffe has written 27 columns on Professionalism in Programming, so presumably readers know what claims such as 'I am a professional' and 'I am professional' mean. But do you? Both those apparently complete statements leave much unsaid.

What does the claim to being a professional mean? Let me be more precise; what does a claim to being a professional software developer mean?

One answer is that it is a statement that the speaker earns their living by developing software. It says nothing about competence nor about any ethical dimension. It also says nothing about any qualifications to earn a living in software development.

The claim to being professional in one's software development may seem the same but is a different claim. It is a claim concerned with competence and ethics.

We have to watch the choice of words very carefully. In some countries the claim to be a software engineer requires some form of certification. For example Germany reserves the term 'engineer' to people who are certified as such. Some people have the mistaken belief that certification guarantees competence. I wish that were so because then we would have not need to de-register or un-certify people because they are incompetent. The best that most certification does is to 'guarantee' that the individual has received appropriate training and satisfied the certification board that they knew what they were supposed to know and had acquired the skills that were deemed necessary.

I still hold valid certification as a teacher and as a sailing instructor. My certification as a teacher is unlimited and qualifies me to teach at any level. It was only my professional standards that prevented me from attempting to teach ages or subjects for which I lacked the appropriate skills and experience.

My NSSA certification as a tidal waters Sailing-master qualifies me to be responsible for groups of young people sailing both inland and on tidal waters. My qualification as a RYA Senior Instructor allows me to hold similar responsibility for groups of adults. However it is too many years since I last sailed on tidal waters and I would never consider taking responsibility for any group of people sailing until I had taken several refresher courses. We have to distinguish between what we are officially certified as being able to do and an awareness of the current limits of our competence. Part of being professional lies in that latter quality.

Exactly what does certification imply? I think it is a way to absolve an employer from some of the responsibility if an employee is incompetent or does something that has bad consequences. It certainly is not some magic that makes the holder more skilful or knowledgeable.

Certification has no impact on an individual's competence to do a job though it does, often, have implications as regards employability. However there are, in my opinion, other far more important issues that distinguish professionalism.

Knowledge of one's limitations is essential. A willingness to continue to develop skills and knowledge is important. Some forms of certification require periodic re-endorsement based on either a demonstration of ongoing practical experience or on retraining. My life-saving certificate is an example; as I have neither applied the skills nor taught others those skills that qualification lapsed three years after the last time I had it re-endorsed. That does not mean that I am unable to act to save someone's life, but it does mean that I am not currently employable in jobs that require I be certified as a lifesaver.

Should we extend the requirement for regular re-endorsement of professional skills to all jobs that have safety implications?

Another feature of being 'professional' as opposed to being 'a professional' is respect for the skills and knowledge of other people. There were numerous occasions during my career as a teacher when I had unqualified (i.e. not qualified as teachers) people present lessons. These people had skills, knowledge and understanding that gave them something worthwhile to contribute to my pupils. I respected that and mostly these non-teachers also understood the limits of what they were allowed to do in the context of a classroom.

Since retiring as a teacher I have turned my hand to quite a few things. I believe that I have acted professionally throughout. I hold no qualification as a journalist, conference organiser, book reviewer etc. But in each case I have taken time to discover how such jobs should be done. I sometimes make mistakes. When I recognise them, I willingly, though not happily, put my hand up. To me, admitting mistakes is part of professional behaviour.

I recently had a member of ACCU tell me that my claim to be a programmer was meaningless because there was no qualification for doing that. The same person opined that only certified engineers should be involved in Standardising C (but chose not to add that requirement to Standardising C++).

I would have some sympathy for his view had the qualifications for certification as a software engineer got anything to do with language design as opposed to language use. I would have even more sympathy if said certification was limited to developing software in a language or languages in which the individual had proven competence. However that latter requirement is left to the professionalism of the individual.

Like many other tasks, working on computer language standards requires professionalism. It actually requires far more skill and knowledge than can be contributed by any single individual. That means that those involved must be able to respect the skills and contributions of other participants. It also means that those involved must be willing to spend time both understanding the current standards and understanding the issues raised by others.

I know of several individuals in the UK who put my knowledge of C to shame but they are currently committed to other work. When a job has to be done we sometimes have to make do with the people who are willing to do it even if someone who is not available could do it better.

While the above is largely personal, I hope that it gives you food for thought. There is no harm (indeed probably much good) in making software development a job that requires appropriate certification. However we should not consider certification as proof that an individual is competent, nor should we require it unless it is relevant to the job.

'I am a certified engineer therefore I am better than you.' should be relegated to the same garbage heap where 'I am older therefore I know better' (a view so often held by adults when dealing with children) belongs.

Undefined Behaviour

We all know that programs that contain undefined behaviour are abhorrent and no professional programmer would consciously write source code that included undefined behaviour unless they had verified that the actual behaviour on the specific platform was acceptable. So consider the following program:

```
#include <stdio.h>
int main() {
  int i = 0;
  puts("Please type in a number: ");
  scanf("%d", &i);
  printf("%d", i*i);
  return 0;
}
```

I have been lazy by using `scanf()` rather than a more robust mechanism. Just pretend that I have carefully written code that will extract an integer value from `stdin`. Given that, where is the undefined behaviour in the above program? How do we justify both C and C++ making that behaviour undefined? Should we do anything about it?

The problem is that signed integer overflow is undefined behaviour in both C and C++. All the five main arithmetic operators (`+`. `-`, `*`, `/` and %) can result in integer overflow. The simplest one is the modulus operator, which can only cause overflow if the divisor is 0. We can easily check for this condition before using the operator.

The division operator is rather subtler because there are two conditions for overflow; the first is division by zero. The second is restricted to 2s complement machines where division of `INT_MIN` by -1 results in overflow. Unfortunately 2s complement is the most common architecture for desktop machines.

Addition and subtraction can both overflow, but again there is a fairly simple pre-test. I leave it to the reader to write one.

Multiplication is the worst case because we have to pre-test by using a division. Let me assume that we start with two positive numbers `a` and `b`. Now compute `INT_MAX/a`. If the result is less than `b` then the result of `a*b` definitely overflows. If the result is equal to `b` we must now check the remainder, because if it is 0 `a*b == INT_MAX`. However if either but not both of the values are negative we have to test against `INT_MIN`. If both are negative, we have to test the absolute value of one of the values against `INT_MAX`.

We can reduce the number of tests if we are willing to accept some false negatives (i.e. rejected cases where the actual calculation does not overflow).

I have seen the argument for allowing overflow to be undefined because any program in which it happens is erroneous. The proponents of the status

quo then add that undefined behaviour will actually not cause anything really bad to happen, such as reformatting your hard drive. Am I alone in finding that argument to be specious? We ask programmers to treat undefined behaviour as a serious issue and then tell them not to worry too much about one of the primary instances of it.

Writing beyond the end of an array is not only undefined behaviour but can result in genuinely bad things happening. I once reprogrammed the BIOS of an expensive graphics card by accidentally writing of the end of an array. In addition buffer overflows are one of the major sources of exploits for malware.

[I'm wouldn't go that far. A large number of software exploits are down to poorly written, insecure code – network code is riddled with such problems. It is not always the case that buffer overflows are the problem. – Ed]

However before I go further along this line, let us see if there is any legitimate code that is vulnerable to this undefined behaviour and that cannot be eliminated by pre-testing. Consider this code snippet:

```
#include <time.h>

void work(void);

int main() {
  clock_t start, end;
  double elapsed;
  start = clock();
  work();
  end = clock();
  elapsed
     = ((double)(end - start))/CLOCKS_PER_SEC;
  printf("Elapsed time = %f", elapsed);
  return 0;
}
```

Now the above program contains irremovable undefined behaviour if `clock_t` is a signed integer type. Neither C nor C++ places any constraint on the type of `clock_t` other than it be an arithmetic type. It does not seem that the actual type requires to be documented (though you could look in the `time.h` header if it has been provided as a file (again not required by either Standard).

Of course most student programs will not have a problem because most implementations will survive just over half an hour of CPU usage before overflow might occur in the return from `clock()`.

However suppose your application runs for much longer and you want to use `clock()` to 'time out' a process. Given defined behaviour for signed integer overflow you have a chance to write code that can handle the problem but without defined behaviour you have no hope and `clock()` is useless to you if you want to write clean code devoid of undefined behaviour.

What concerns me is that a number of C and C++ heavyweight experts take the view that anything we have lived with for thirty years cannot be a problem. So, am I wrong to be concerned with this issue?

Should we be comfortable with undefined behaviour that will do nothing disastrous? Do we need another classification of behaviour that basically says that the worst that can happen is that the program aborts? Of course such behaviour is not acceptable in the control software for a nuclear power station but it is acceptable for many other purposes. Yes the program does not always do what the programmer intended but neither does it try to reformat my hard drive.

Pure Functions

Those who come from a functional programming background will be familiar with the concept of a pure function but for the rest, a pure function is one that has no side effects, the opposite of a procedure that has only side effects and no return value.

In C++ a pure function would be a free function that does not access any globals, does not have any local statics in its definition and whose parameters are all value based (no pointers either). Under such circumstances the return value is solely based on the function's arguments. A pure function can only call pure functions.

Until recently the concept of pure functions has been interesting but of no great direct value to languages such as C++. However hardware is moving on. Multiple CPU machines are increasingly common. The latest hardware from Intel allows a single CPU to look like two. CPUs have multiple processing lines built in to them. In pursuit of ever faster hardware the next logical step is to put array processors into our CPUs. Single Instruction Multiple Data (SIMD) parallelism can result in great performance improvements for certain types of processing (and graphical processing is an example of such a specialist domain).

Given such hardware pure functions begin to become useful. Pure functions are obvious candidates for SIMD parallelism.

Is it time that C++ considered adding some function qualifiers so that we can identify functions as pure. Such information is statically enforceable. We would need to consider such details as including such qualifiers in the type of function pointers (the address of a pure function should be assignable to any suitable function pointer, but only addresses of pure functions should be assignable to pointers to pure functions.)

Please give some thought to this idea, write them down and email them to the editor or to me.

Commentary on Problem 16

Here is what I invited you to comment on:

> Have a look at the following tiny function. The problem is insidious; the same code is legal in Java and does exactly what you want, while in C++ it compiles without error.
>
> ```
> string to_string(int n) {
>   if(n == 0) {
>     return "NULL";
>   }
>   else {
>     return "" + n;
>   }
> }
> ```

The problem is because of the very different ways that the `operator +` is overloaded in the two languages. In the case of Java `operator +` creates a new string object that is the result of concatenating the left-hand operand with the conversion of the right-hand one to a string by using whatever method is provided by the rhs' type.

In the case of C++ there is no `operator +` overload that takes either an array of one `char` on the right. However there is an `operator +` that takes a `char const *` and an `int`; the one that increments the pointer by the specified value. If the `int` is 0 the result is to leave the pointer unchanged (but in this function we trap that case and handle it differently). If `n` is one the result is a one beyond the end pointer, which is OK until it gets used to initialise the return value where the string constructor will dereference the pointer.

In all other cases the evaluated `"" + n` expression has undefined behaviour even though nothing really bad happens on most systems. You just get garbage as the program treats the bytes that start at the computed address as if they were part of a null-terminated array of `char`.

Problem 17

Here is a minimalist version of `main()`:

```
int main() {
  a * b;
}
```

Given suitable precursors it will compile and execute. Can you provide suitable precursors so that the resulting program executes and outputs:

```
int main() {
  a * b;
}
```

I will send the author of the solution that I like best (yes, entirely subjective) a copy of *Exceptional C++ Style* (and if I remember to get one autographed by Herb Sutter when I am at the WG21 meeting in Redmond it will be an autographed copy).

Cryptic Clues For Numbers

I had several clues offered. Ainsley Pereira offered 'Before he ate, he had to wait'. James Roberts offered both 'A new slant to infinity, and again?' (A good start but I think it needs some polish) and 'Produce of the disciples working every hour of the day and night.' (I think 'product' works better). Louis Lavery came up with 'Told to double weight after initial loss? I'd say that's too gross!' Any one of those would deserve to win and correctly identify 288. I chose James' so if he contacts me to tell me where to send it he gets the copy of *The Elements of C++ Style*.

For next time, what are your clues for:

Oh for love in the sea! It only values the fifth bit.

Francis Glassborow

Features

An Introduction to Programming with GTK+ and Glade – Part 2

Roger Leigh <rleigh@debian.org>

Last Time...

As you will recall from the first part, I showed how to construct a GTK window with all of its associated parts in some detail, and it was clear that to do this for every application would not be a good idea. In this part, I will be showing how to use the Glade application for rapid window creation.

Analysis

The `main()` function is responsible for constructing the user interface, connecting the signals to the signal handlers, and then entering the main event loop. The more complex aspects of the function are discussed here.

```
g_signal_connect(G_OBJECT(window), "destroy",
                 gtk_main_quit, NULL);
```

This code connects the `"destroy"` signal to the `gtk_main_quit()` function. This signal is emitted by the window if is to be destroyed, for example when the "close" button on the titlebar is clicked). The result is that when the window is closed, the main event loop returns, and the program then exits.

```
vbox1 = gtk_vbox_new(FALSE, 0);
gtk_container_add(GTK_CONTAINER(window), vbox1);
```

`vbox1` is a `GtkVBox`. When constructed using `gtk_vbox_new()`, it is set to be non-homogenous (`FALSE`), which allows the widgets contained within the `GtkVBox` to be of different sizes, and has zero pixels padding space between the containers it contains. The homogeneity and padding space are different for the various `GtkBox`es used, depending on the visual effect intended.

`gtk_container_add()` packs `vbox1` into the window (a `GtkWindow` object is a `GtkContainer`).

```
eventbox = gtk_event_box_new();
gtk_widget_show(eventbox);
gtk_box_pack_start(GTK_BOX(hbox2), eventbox,
                   FALSE, FALSE, 0);
```

Some widgets do not receive events from the windowing system, and hence cannot emit signals. Label widgets are one example of this. If this is required, for example in order to show a tooltip, they must be put into a `GtkEventBox`, which can receive the events. The signals emitted from the `GtkEventBox` may then be connected to the appropriate handler.

`gtk_widget_show()` displays a widget. Widgets are hidden by default when created, and so must be shown before they can be used.

It is typical to show the top-level window last, so that the user does not see the interface being drawn.

`gtk_box_pack_start()` packs a widget into a `GtkBox`, in a similar manner to `gtk_container_add()`. This packs `eventbox` into `hbox2`. The last three arguments control whether the child widget should expand into any extra space available, whether it should fill any extra space available (this has no effect if expand is FALSE), and extra space in pixels to put between its neighbours (or the edge of the box), respectively. Figure 1 shows how `gtk_box_pack_start()` works.

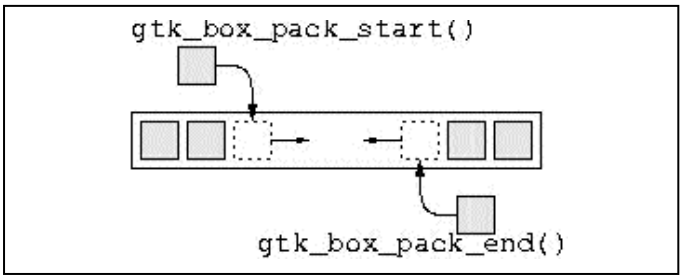


Figure 1: `gtk_box_pack_start()`

The `create_spin_entry()` function is a helper function to create a numeric entry (spin button) together with a label and tooltip. It is used to create all three entries.

```
label = gtk_label_new(label_text);
```

A new label is created displaying the text `label_text`.

```
spinbutton = gtk_spin_button_new(adjustment,
                                 0.5, 2);
gtk_spin_button_set_numeric(
          GTK_SPIN_BUTTON(spinbutton), TRUE);
```

A `GtkSpinButton` is a numeric entry field. It has up and down buttons to "spin" the numeric value up and down. It is associated with a `GtkAdjustment`, which controls the range allowed, default value, etc. `gtk_adjustment_new()` returns a new `GtkAdjustment` object. Its arguments are the default value, minimum value, maximum value, step increment, page increment and page size, respectively. This is straightforward, apart from the step and page increments and sizes. The step and page increments are the value that will be added or subtracted when the mouse button 1 or button 2 are clicked on the up or down buttons, respectively. The page size has no meaning in this context (`GtkAdjustment`s are also used with scrollbars).

`gtk_spin_button_new()` creates a new `GtkSpinButton`, and associates it with `adjustment`. The second and third arguments set the "climb rate" (rate of change when the spin buttons are pressed) and the number of decimal places to display.

Finally, `gtk_spin_button_set_numeric()` is used to ensure that only numbers can be entered.

```
tooltip = gtk_tooltips_new();
gtk_tooltips_set_tip(tooltip, eventbox,
                     tooltip_text, NULL);
```

A tooltip (pop-up help message) is created with `gtk_tooltips_new()`. `gtk_tooltips_set_tip()` is used to associate `tooltip` with the `eventbox` widget, also specifying the message it should contain. The fourth argument should typically be `NULL`.

The `create_result_label()` function is a helper function to create a result label together with a descriptive label and tooltip.

```
gtk_label_set_selectable(
              GTK_LABEL(result_value), TRUE);
```

Normally, labels simply display a text string. The above code allows the text to be selected and copied, to allow pasting of the text elsewhere. This is used for the result fields so the user can easily copy them.

```
button1
  = gtk_button_new_from_stock(GTK_STOCK_QUIT);
```

This code creates a new button, using a stock widget. A stock widget contains a predefined icon and text. These are available for commonly used functions, such as "OK", "Cancel", "Print", etc..

```
button2 = gtk_button_new_with_mnemonic(
                                  "_Calculate");
g_signal_connect(G_OBJECT (button2),
      "clicked",
      G_CALLBACK(on_button_clicked_calculate),
      (gpointer) &cb_widgets);
GTK_WIDGET_SET_FLAGS(button2,
                     GTK_CAN_DEFAULT);
```

Here, a button is created, with the label "Calculate". The mnemonic is the "_C", which creates an accelerator. This means that when Alt-C is pressed, the button is activated (i.e. it is a keyboard shortcut). The shortcut is underlined, in common with other graphical toolkits.

The "clicked" signal (emitted when the button is pressed and released) is connected to the `on_button_clicked_calculate()` callback. The `cb_widgets` structure is passed as the argument to the callback.

Lastly, the `GTK_CAN_DEFAULT` attribute is set. This attribute allows the button to be the default widget in the window.

```
g_signal_connect_swapped
  (G_OBJECT(cb_widgets.pg_val), "activate",
     G_CALLBACK(gtk_widget_grab_focus),
     (gpointer)GTK_WIDGET(cb_widgets.ri_val));
```

This code connects signals in the same way as `gtk_signal_connect()`. The difference is the fourth argument, which is a `GtkWidget` pointer. This allows the signal emitted by one widget to be received by the signal handler for another. Basically, the widget argument of the signal handler is given `cb_widgets.ri_val` rather than `cb_widgets.pg_val`. This allows the focus (where keyboard input is sent) to be switched to the next entry field when Enter is pressed in the first.

```
g_signal_connect_swapped
  (G_OBJECT(cb_widgets.cf_val), "activate",
      G_CALLBACK(gtk_window_activate_default),
      (gpointer)GTK_WIDGET(window));
```

This is identical to the last example, but in this case the callback is the function `gtk_window_activate_default()` and the widget to give to the signal handler is `window`. When Enter is pressed in the CF entry field, the default "Calculate" button is activated.

```
gtk_main();
```

This is the GTK+ event loop. It runs until `gtk_main_quit()` is called.

The signal handlers are far simpler than building the interface. The function `on_button_clicked_calculate()` reads the user input, performs a calculation, then displays the result.

```
void on_button_clicked_calculate(
                              GtkWidget *widget,
                              gpointer data) {
  struct calculation_widgets *w;
  w = (struct calculation_widgets *) data;
```

Recall that a pointer to `cb_widgets`, of type `struct calculation_widgets`, was passed to the signal handler, cast to a `gpointer`. The reverse process is now applied, casting `data` to a pointer of type `struct calculation_widgets`.

```
  gdouble pg;
  pg = gtk_spin_button_get_value(
                    GTK_SPIN_BUTTON(w->pg_val));
```

This code gets the value from the `GtkSpinButton`.

```
  gchar *og_string;
  og_string = g_strdup_printf("<b>%0.2f</b>", og);
  gtk_label_set_markup(GTK_LABEL(w->og_result),
                       og_string);
  g_free(og_string);
```

Here the result `og` is printed to the string `og_string`. This is then set as the label text using `gtk_label_set_markup()`. This function sets the label text using the Pango Markup Format, which uses the `<b>` and `</b>` tags to embolden the text.

```
  gtk_spin_button_set_value(
              GTK_SPIN_BUTTON(w->pg_val), 0.0);
  gtk_label_set_text(
              GTK_LABEL(w->og_result), "");
```

`on_button_clicked_reset()` resets the input fields to their default value, and blanks the result fields.

GTK+ and Glade

Introduction

In the previous section, the user interface was constructed entirely "by hand". This might seem to be rather difficult to do, as well as being messy and time-consuming. In addition, it also makes for rather unmaintainable code, since changing the interface, for example to add a new feature, would be rather hard. As interfaces become more complex, constructing them entirely in code becomes less feasible.

The Glade user interface designer is an alternative to this. Glade allows one to design an interface visually, selecting the desired widgets from a palette and placing them on windows, or in containers, in a similar manner to other interface designers. Figure 3 (see next page) shows some screenshots of the various components of Glade.

The file `C/glade/ogcalc.glade` contains the same interface constructed in `C/plain/ogcalc.c`, but designed in Glade. This file can be opened in Glade, and changed as needed, without needing to touch any code.

Even signal connection is automated. Examine the "Signals" tab in the "Properties" dialog box.

The source code is listed below. This is the same as the previous listing, but with the following changes:

- The `main()` function does not construct the interface. It merely loads the `ogcalc.glade` interface description, auto-connects the signals, and shows the main window.
- The `cb_widgets` structure is no longer needed: the callbacks are now able to query the widget tree through the Glade XML object to locate the widgets they need. This allows for greater encapsulation of data, and signal handler connection is simpler.
- The code saving is significant, and there is now separation between the interface and the callbacks.

The running `C/glade/ogcalc` application is shown in Figure 2. Notice that it is identical to `C/plain/ogcalc`, shown in the last article. (No, they are not the same screenshot!)

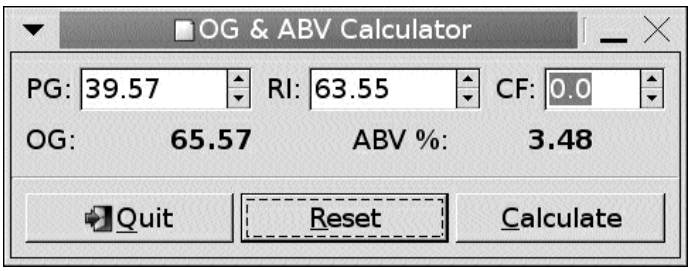


Figure 2: `C/glade/ogcalc` in action

Analysis

The most obvious difference between the code using Glade (see listing at end of article) and the previous code is the huge reduction in size. The `main()` function is reduced to just these lines:

```
  GladeXML *xml;
  GtkWidget *window;

  xml = glade_xml_new("ogcalc.glade", NULL, NULL);

  glade_xml_signal_autoconnect(xml);

  window = glade_xml_get_widget(xml,
                        "ogcalc_main_window");
  gtk_widget_show(window);
```

`glade_xml_new()` reads the interface from the file `ogcalc.glade`. It returns the interface as a pointer to a `GladeXML` object, which will be used later. Next, the signal handlers are connected with `glade_xml_signal_autoconnect()`. Windows users may require special linker flags because signal autoconnection requires the executable to have a dynamic symbol table in order to dynamically find the required functions.

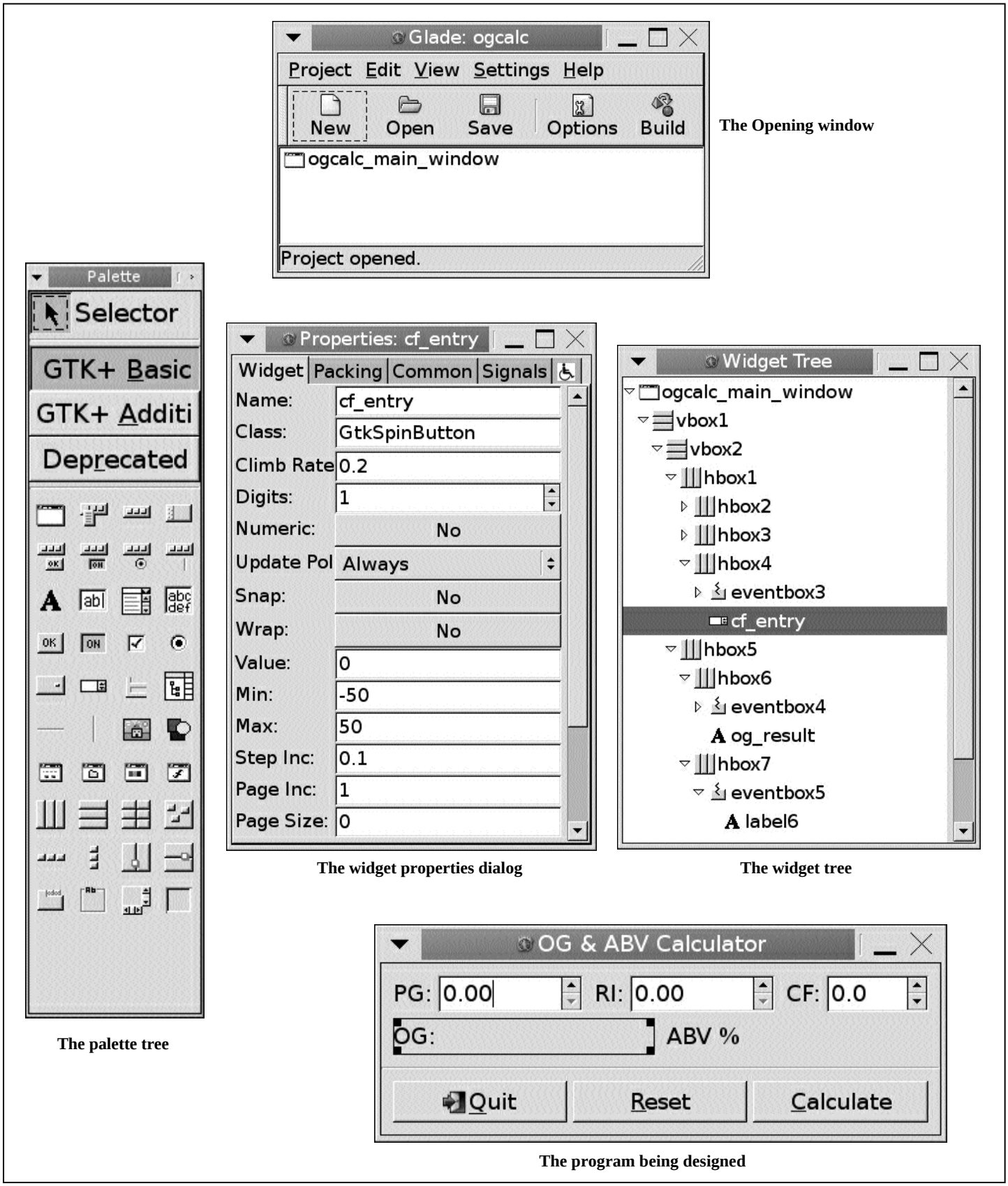
The signal handlers are identical to those in the previous section. The only difference is that `struct calculation_widgets` has been removed. No information needs to be passed to them through the data argument, since the widgets they need to use may now be found using the `GladeXML` interface description.

```
GtkWidget *pg_val;
GladeXML *xml;
xml = glade_get_widget_tree(
                          GTK_WIDGET (widget));
pg_val = glade_xml_get_widget(xml, "pg_entry");
```

Firstly, the `GladeXML` interface is found, by finding the widget tree containing the widget passed as the first argument to the signal handler. Once `xml` has been set, `glade_xml_get_widget()` may be used to obtain pointers to the `GtkWidget`s stored in the widget tree.

Compared with the pure C GTK+ application, the code is far simpler, and the signal handlers no longer need to get their data as structures cast to `gpointer`, which was ugly. The code is far more understandable, cleaner and maintainable.

Roger Leigh



The palette tree

The widget properties dialog

The widget tree

The program being designed

Figure 3: The Glade user interface designer

Listing: C/glade/ogcalc.c

```
#include <gtk/gtk.h>
#include <glade/glade.h>

void on_button_clicked_reset(GtkWidget *widget,
                             gpointer data);
void on_button_clicked_calculate(GtkWidget *widget,
                                 gpointer data);

/* The bulk of the program.  Since Glade and
   libglade are used, this is just 9 lines! */
int main(int argc, char *argv[]) {
  GladeXML *xml;
  GtkWidget *window;

  /* Initialise GTK+. */
  gtk_init(&argc, &argv);

  /* Load the interface description. */
  xml = glade_xml_new("ogcalc.glade", NULL, NULL);

  /* Set up the signal handlers. */
  glade_xml_signal_autoconnect(xml);

  /* Find the main window and then show it. */
  window = glade_xml_get_widget(xml,
                                "ogcalc_main_window");
  gtk_widget_show(window);

  /* Enter the GTK Event Loop.  This is where all
     the events are caught and handled.  It is
     exited with gtk_main_quit(). */
  gtk_main();

  return 0;
}

/* This is a callback.  This resets the values of
   the entry widgets, and clears the results. */
void on_button_clicked_reset(GtkWidget *widget,
                             gpointer data) {
  GtkWidget *pg_val;
  GtkWidget *ri_val;
  GtkWidget *cf_val;
  GtkWidget *og_result;
  GtkWidget *abv_result;

  GladeXML *xml;

  /* Find the Glade XML tree containing widget. */
  xml = glade_get_widget_tree(GTK_WIDGET (widget));

  /* Pull the other widgets out the the tree. */
  pg_val = glade_xml_get_widget(xml,
                                "pg_entry");
  ri_val = glade_xml_get_widget(xml,
                                "ri_entry");
  cf_val = glade_xml_get_widget(xml,
                                "cf_entry");
  og_result = glade_xml_get_widget(xml,
                                "og_result");
  abv_result = glade_xml_get_widget(xml,
                                "abv_result");

  gtk_spin_button_set_value(GTK_SPIN_BUTTON(pg_val),
                            0.0);
  gtk_spin_button_set_value(GTK_SPIN_BUTTON(ri_val),
                            0.0);
  gtk_spin_button_set_value(GTK_SPIN_BUTTON(cf_val),
                            0.0);
  gtk_label_set_text(GTK_LABEL(og_result), "");
  gtk_label_set_text(GTK_LABEL(abv_result), "");
}

/* This callback does the actual calculation. */
void on_button_clicked_calculate(GtkWidget *widget,
                                 gpointer data) {
  GtkWidget *pg_val;
  GtkWidget *ri_val;
  GtkWidget *cf_val;
  GtkWidget *og_result;
  GtkWidget *abv_result;

  GladeXML *xml;

  gdouble pg, ri, cf, og, abv;
  gchar *og_string;
  gchar *abv_string;

  /* Find the Glade XML tree containing widget. */
  xml = glade_get_widget_tree(GTK_WIDGET(widget));

  /* Pull the other widgets out the the tree. */
  pg_val = glade_xml_get_widget(xml,
                                "pg_entry");
  ri_val = glade_xml_get_widget(xml,
                                "ri_entry");
  cf_val = glade_xml_get_widget(xml,
                                "cf_entry");
  og_result = glade_xml_get_widget(xml,
                                "og_result");
  abv_result = glade_xml_get_widget(xml,
                                "abv_result");

  /* Get the numerical values from the entry
     widgets. */
  pg = gtk_spin_button_get_value(
                          GTK_SPIN_BUTTON(pg_val));
  ri = gtk_spin_button_get_value(
                          GTK_SPIN_BUTTON(ri_val));
  cf = gtk_spin_button_get_value(
                          GTK_SPIN_BUTTON(cf_val));

  og = (ri * 2.597) - (pg * 1.644) - 34.4165 + cf;

  /* Do the sums. */
  if (og < 60)
    abv = (og - pg) * 0.130;
  else
    abv = (og - pg) * 0.134;

  /* Display the results.  Note the <b></b> GMarkup
     tags to make it display in Bold. */
  og_string = g_strdup_printf("<b>%0.2f</b>",
                              og);
  abv_string = g_strdup_printf("<b>%0.2f</b>",
                               abv);

  gtk_label_set_markup(GTK_LABEL(og_result),
                       og_string);
  gtk_label_set_markup(GTK_LABEL(abv_result),
                       abv_string);

  g_free(og_string);
  g_free(abv_string);
}
```

To build the source, do the following:

```
cd C/glade
cc 'pkg-config --cflags libglade-2.0'
                                -c ogcalc.c
cc 'pkg-config --libs libglade-2.0'
                                -o ogcalc ogcalc.o
```

Rapid Dialog Design Using Qt

Jasmin Blanchette

In this third installment of our series on GUI programming with the Qt C++ toolkit, we're going to show how to design dialog boxes (or "dialogs") using Qt. Dialogs can be created entirely from source code, or with *Qt Designer*, a visual GUI design tool. Whichever approach is chosen, the result is invariably good looking, resizable, platform-independent dialogs.

Writing Dialogs in Code

Writing dialogs entirely in code using Qt isn't the chore you'd expect if you're familiar with other toolkits such as Swing, GTK+ or MFC. Qt's layout manager classes take care of positioning widgets on screen. Qt provides a horizontal box layout, a vertical box layout and a grid layout. These can be nested to create arbitrarily complex layouts.

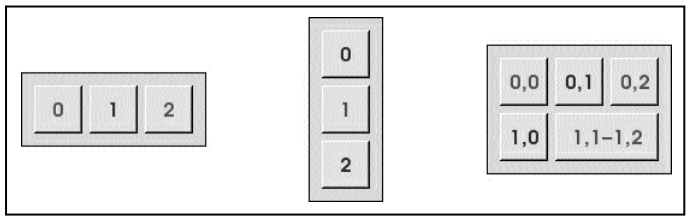


Figure 1: Qt's layout managers

Qt's layouts feature automatic positioning and resizing of child widgets, sensible minimum and default sizes for top-level widgets, and automatic repositioning when the contents, language or font changes. For cross-platform applications, Qt's layouts are a huge time-saver.

Layouts are also useful for internationalization. With fixed sizes and positions, the translation text is often truncated; with layouts, the child widgets are automatically resized. Furthermore, if you translate your application to a right-to-left language such as Arabic or Hebrew, layouts will automatically reverse themselves to follow the direction of writing.

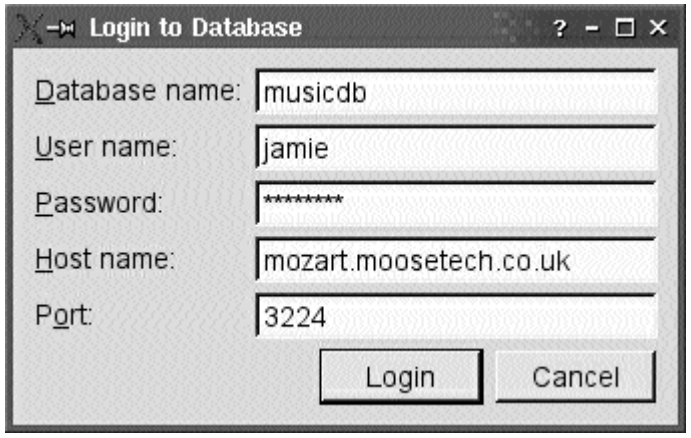


Figure 2: The "Login to Database dialog under KTE

To see how this works in practice, we will implement the "Login to Database" dialog shown above. This is achieved by deriving from `QDialog` (which in turn derives from `QWidget`) and writing the code for a few functions. Let's start with the header file:

```
// include guards omitted
#include <qdialog.h>

class QLabel;
class QLineEdit;
class QPushButton;

class LoginDialog : public QDialog {
  Q_OBJECT

public:
  LoginDialog(QWidget *parent = 0);
private slots:
  void enableLoginButton();
private:
  QLineEdit *dbNameLineEdit;
  QLineEdit *userNameLineEdit;
  QLineEdit *passwordLineEdit;
  QLineEdit *hostNameLineEdit;
  QLineEdit *portLineEdit;
  QLabel *dbNameLabel;
  QLabel *userNameLabel;
  QLabel *passwordLabel;
  QLabel *hostNameLabel;
  QLabel *portLabel;
  QPushButton *loginButton;
  QPushButton *cancelButton;
};
```

The `LoginDialog` class has a typical Qt widget constructor that accepts a parent widget (or window), a slot called `enableLoginButton()`, and a dozen data member that keep track of the dialog's child widgets. The `Q_OBJECT` macro at the top of the class definition is necessary because we are using Qt's "signals and slots" mechanism in the class.

Let's now review the implementation file:

```
#include <qlabel.h>
#include <qlayout.h>
#include <qlineedit.h>
#include <qpushbutton.h>
#include "logindialog.h"

LoginDialog::LoginDialog(QWidget *parent)
             : QDialog(parent) {
  dbNameLabel
    = new QLabel(tr("&Database name:"), this);
  userNameLabel
      = new QLabel(tr("&User name:"), this);
  passwordLabel
       = new QLabel(tr("&Password:"), this);
  hostNameLabel
      = new QLabel(tr("&Host name:"), this);
  portLabel = new QLabel(tr("P&ort:"), this);
  // more follows
```

The constructor passes on the `parent` parameter to the base class constructor. If `parent` is non-null, the dialog automatically centers itself on top of the parent window and shares that window's taskbar entry. In addition, if the dialog is modal (which is achieved by calling `setModal()` or `exec()` on the dialog), the user won't be allowed to interact with the parent window until the user closes the dialog.

Next, the constructor creates five `QLabel` widgets showing the texts "Database name:", "User name:", " Password:", "Host name:" and "Port:". The ampersand character ('&') indicates which letter is the shortcut key. The `tr()` function that surrounds the string literals marks the strings as translatable.

The second argument to the `QLabel` constructor is the parent widget or window, in this case the dialog (`this`). Child widgets are shown on screen inside their parent.

```
  dbNameLineEdit = new QLineEdit(this);
  userNameLineEdit = new QLineEdit(this);
  passwordLineEdit = new QLineEdit(this);
  passwordLineEdit->setEchoMode(
                        QLineEdit::Password);
  hostNameLineEdit = new QLineEdit(this);
  portLineEdit = new QLineEdit(this);

  dbNameLabel->setBuddy(dbNameLineEdit);
  userNameLabel->setBuddy(userNameLineEdit);
  passwordLabel->setBuddy(passwordLineEdit);
  hostNameLabel->setBuddy(hostNameLineEdit);
  portLabel->setBuddy(portLineEdit);
  // more follows
```

We create five `QLineEdit` widgets and set the "Password" widget's echo mode to `QLineEdit::Password`, so that characters typed by the user

are replaced by asterisks or bullets. After creating the widgets, we call `setBuddy()` to create associations between the labels and the line editors. When the user presses a label's shortcut key (for example, Alt+P for "Password:"), the associated line editor gets the focus.

```
connect(dbNameLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
connect(userNameLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
connect(passwordLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
connect(hostNameLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
connect(portLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
// more follows
```

We connect the `textChanged()` signal of each line editor to the dialog's `enableLoginButton()` slot. Whenever the user types in some text, the `textChanged()` signal is emitted and the `enableLoginButton()` slot is called. Based on the contents of the line editors, `enableLoginButton()` enables or disables the dialog's "Login" button. Disabled widgets are typically greyed out.

```
loginButton = new QPushButton(tr("Login"),
                              this);
cancelButton = new QPushButton(tr("Cancel"),
                               this);
loginButton->setDefault(true);
loginButton->setEnabled(false);

connect(loginButton, SIGNAL(clicked()),
        this, SLOT(accept()));
connect(cancelButton, SIGNAL(clicked()),
        this, SLOT(reject()));
// more follows
```

We create the "Login" and "Cancel" buttons, make "Login" the default button (meaning that pressing Enter will effectively click that button), and disable it. Then we connect the "Login" button's `clicked()` signal and `QDialog`'s `accept()` slot, and connect the "Cancel" button's `clicked()` signal and `QDialog`'s `reject()` slot. Both slots close the dialog, but they set `QDialog`'s return code to a different value, which applications can query afterwards.

Now that we're done with creating the child widgets, we must set their positions and sizes relative to the parent widget. This could be done using `QWidget::setGeometry()`, but the result would be a hard-coded, unresizable dialog. Furthermore, determining pixel coordinates for the dialog's widgets is a tedious task that is better performed by a machine.

To obtain the desired result, we need two layout managers, one nested into the other. The dialog's main layout (the outer layout) is a grid layout with six rows and two columns. The inner layout is a horizontal box layout that contains the "Login" and "Cancel" buttons. The inner layout occupies the bottom row of the grid.

Here comes the code:

```
QHBoxLayout *buttonLayout = new QHBoxLayout;
buttonLayout->addStretch(1);
buttonLayout->addWidget(loginButton);
buttonLayout->addWidget(cancelButton);

QGridLayout *mainLayout
                    = new QGridLayout(this);
mainLayout->setMargin(10);
mainLayout->setSpacing(5);
mainLayout->addWidget(dbNameLabel, 0, 0);
mainLayout->addWidget(dbNameLineEdit, 0, 1);
mainLayout->addWidget(userNameLabel, 1, 0);
mainLayout->addWidget(userNameLineEdit, 1, 1);
mainLayout->addWidget(passwordLabel, 2, 0);
mainLayout->addWidget(passwordLineEdit, 2, 1);
mainLayout->addWidget(hostNameLabel, 3, 0);
mainLayout->addWidget(hostNameLineEdit, 3, 1);
mainLayout->addWidget(portLabel, 4, 0);
mainLayout->addWidget(portLineEdit, 4, 1);
mainLayout->addMultiCellLayout(buttonLayout,
                               5, 5, 0, 1);
// more follows
```

We start by creating the `QHBoxLayout` that contains the buttons. We insert a stretch item, the "Login" button and the "Cancel" button into the layout. The layout will place them side by side. The stretch item is there to fill the space on the left of the buttons; without it, `QHBoxLayout` would stretch the "Login" and "Cancel" buttons to fill the entire width of the dialog.

Then we create a `QGridLayout`. We set the layout's margin to 10 pixels and the spacing between widgets in the layout to 5 pixels. Then we add the widgets to the layout. The `addWidget()` function takes a widget, a row and a column as parameters. At the very end, we insert the `QHBoxLayout` into the `QGridLayout` using `addMultiCellLayout()`, and specify that it should extend from row 5 to row 5 and from column 0 to column 1; i.e. occupy cells (5, 0) and (5, 1).

Here comes the rest of the constructor, where we set the window title:

```
  setCaption(tr("Login to Database"));
}
```

The constructor code might have felt a bit long. The good news is that we're pretty much finished now. The only missing part is the `enableLoginButton()` slot:

```
void LoginDialog::enableLoginButton() {
  loginButton->setEnabled(
        !dbNameLineEdit->text().isEmpty()
        && !userNameLineEdit->text().isEmpty()
        && !passwordLineEdit->text().isEmpty()
        && !hostNameLineEdit->text().isEmpty()
        && !portLineEdit->text().isEmpty());
}
```

When the user edits the contents of one of the line editors, the `enableLoginButton()` slot is called. The slot sets the button's state to enabled if and only if all the `QLineEdit`s contain some text.

At this point, you might wonder why `LoginDialog` has no destructor. After all, who will delete all the objects created with `new` in the constructor? The answer is that when you create a widget or layout with a parent, the parent assumes ownership for the child. There is therefore no need for a `LoginDialog` destructor that simply deletes the child widgets and layouts; this is exactly what the `QWidget` destructor does. (Recall that `LoginDialog` inherits `QDialog`, which inherits `QWidget`.)

Designing Dialogs Visually With Qt Designer

Qt Designer is a visual GUI design tool included with Qt. Although Qt's nice API makes it easy to write dialogs purely in code, most Qt developers find that Qt Designer is faster to use and allows them to make prototypes very quickly. In addition, if you work in an organisation where the user interface design is done by a team of designers, the designers can use Qt Designer themselves to create the dialogs instead of producing specifications that the developers then need to implement.

To show how Qt Designer works, we will use it to redo the "Login to Database" dialog.

Creating a dialog in Qt Designer usually consists of the following steps:

- Put child widgets on the form.
- Set up their properties.
- Group them into layouts.
- Specify the tab order.

The first step, putting the required child widgets on the form, is accomplished by clicking the desired widget from the toolbox on the left of Qt Designer's main window followed by clicking the desired position on the form. For the moment, we don't need to worry too much about the precise position and size of the child widgets; soon enough, we will put them in layouts, which will take care of those aspects automatically.

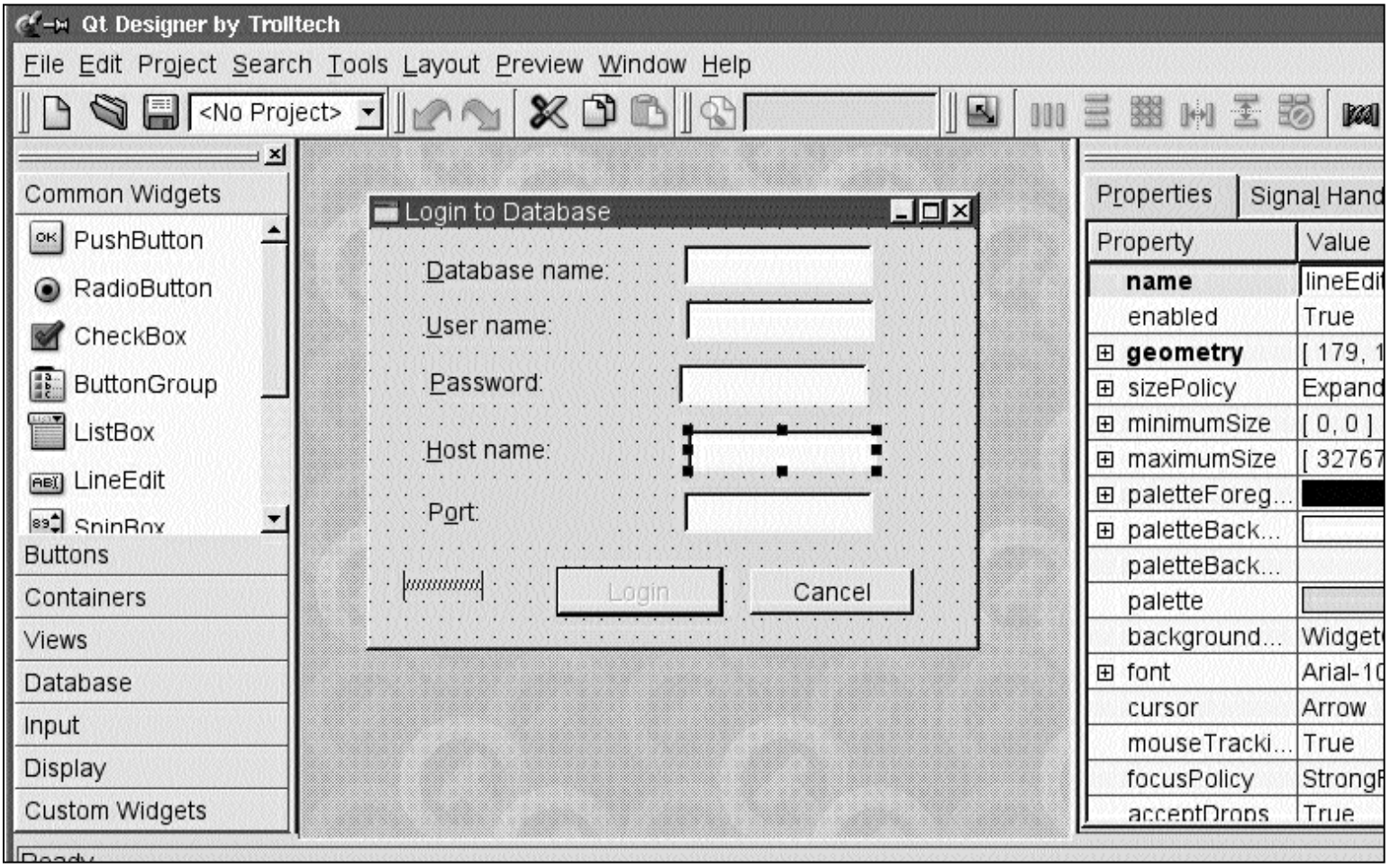


Figure 3: Qt Designer in action

We also need a stretch item to fill the extra space in the buttons' layout. It is represented by a blue "spring" in Qt Designer.

Next, we must set the child widgets' properties using the property editor located on the right side of Qt Designer's main window. Start by renaming all the widgets so that they have the same names as in the previous example. Then click the background of the form and set the form's "name" property to "LoginDialogBase" and its "caption" property to "Login to Database".

The following table summarises the properties to set for each widget:

Widget	Property	Value
`dbNameLabel`	`text`	`"&Database name:"`
`userNameLabel`	`text`	`"&User name:"`
`passwordLabel`	`text`	`"&Password:"`
`hostNameLabel`	`text`	`"&Host name:"`
`portLabel`	`text`	`"P&ort:"`
`passwordLineEdit`	`echoMode`	`Password`
`loginButton`	`text`	`"Login"`
	`default`	`Truex`
	`enabled`	`Falsex`
`cancelButton`	`text`	`"Cancel"`

We need to set the labels' "buddies". This is done by setting the "buddy" property of each label to the corresponding widget. Once the properties are set, the dialog should look like the one shown in Figure 4.

The next step is to put the widgets inside layouts. This is done by selecting multiple widgets and choosing "Lay Out Horizontally", "Lay Out Vertically" or "Lay Out in a Grid" from the "Layout" menu.

First, we select the stretch item and the two buttons, and click "Lay Out Horizontally". The resulting layout is rendered as a red frame in Qt Designer, to make it tangible. Then we click the background of the form and click "Lay Out in a Grid". This will produce the layout shown in Figure 5.

If a layout doesn't turn out quite right, we can always click "Undo", then roughly reposition the widgets being laid out and try again.

When everything else is done, we are ready to set the dialog's tab order. This is done by pressing F4, clicking the widgets in the order we want them to be in the tab chain, and pressing Esc to terminate. Qt Designer will display the tab order as numbers in blue circles.

We can now save the dialog as a `.ui` file that contains the dialog in an XML format that Qt Designer can load and save. This file is converted to C++ using a separate tool called `uic` (User Interface Compiler). Assuming the `.ui` file is called `logindialogbase.ui`, the resulting C++ code would appear in the `logindialogbase.h` and `logindialogbase.cpp` files.

The dialog looks identical to the one we developed earlier purely in code, but right now if the user fills in the line editors or presses "Cancel", nothing happens! This is solved by subclassing the `uic`-generated class and adding the missing functionality there, as follows.

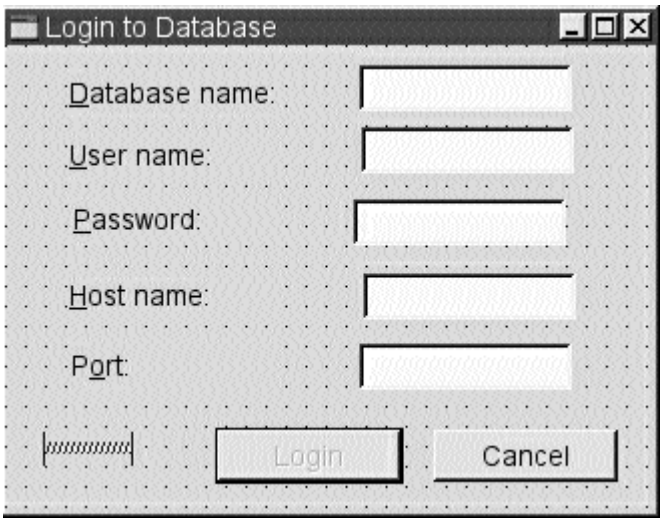


Figure 4: The dialog with properties set

Introduction to STL (Standard Template Library)

Rajanikanth Jammalamadaka <rajani@ece.arizona.edu>

A template is defined as "*something that establishes or serves as a pattern*"

Websters

In C++, a template has more or less the same meaning. A template is like a skeleton code which becomes "*alive*" when it is instantiated with a type.

An algorithm is a *well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time* [1].

A class holding a collection of elements of some type is commonly called a *container class*, or simply a *container* [2].

A class is a user defined type which is very similar to the pre-defined types like `int`, `char`, etc. So, the standard template library (STL) is a collection of generic algorithms and containers which are orthogonal to each other. By the word orthogonal, we mean that any algorithm can be used on any container and vice-versa.

In C++, there are various generic classes like `vector`, `string`, etc.

Since STL is a very large topic to be covered in an article or two, we will focus on the most commonly used generic classes: `vector` and `string`. Before we discuss the standard containers let us take a simple example to understand the word *template*.

```
// template.cpp

#include<iostream>
using std::cout;

template<typename T>
         // Declares T as a name of some type
/* It is also common to see template<class T>.
   The two mean the same */

/* The following template defines a function
   which takes two constant references of type
   T and returns the maximum value of type T.
   */

T Max(const T& a, const T& b) {
  return (a > b)? a : b;
}

int main() {
  cout << Max('i', 'r') << "\n";
  cout << Max(1, 3) << "\n";
}

// Output of template.cpp
r
3
```

In the above example, a function `Max` is defined which takes two constant references of type `T` and returns the maximum value.

But in the main function there are two function calls, one is `Max('i','r')` and the other one is `Max(1,3)`. We did not get any compilation errors because we have used the template mechanism in this function. At run-time, the compiler resolves what types are being passed to the `Max` function and hence knows which output to return. It should be noted that the final compiled binary will be larger as the compiler has to create a function for every type in the code passed to it.

[concluded from previous page]

```
  // Header file:

  // include guards omitted

  #include "logindialogbase.h"

  class LoginDialog : public LoginDialogBase {
    Q_OBJECT
  public:
    LoginDialog(QWidget *parent = 0);
  private slots:
    void enableLoginButton();
  };

  // Implementation file:

  #include <qlabel.h>
  #include <qlayout.h>
  #include <qlineedit.h>
  #include <qpushbutton.h>
  #include "logindialog.h"

  LoginDialog::LoginDialog(QWidget *parent)
      : LoginDialogBase(parent) {
    connect(dbNameLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
    connect(userNameLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
    connect(passwordLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
    connect(hostNameLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
    connect(portLineEdit,
        SIGNAL(textChanged(const QString &)),
        this, SLOT(enableLoginButton()));
    connect(loginButton, SIGNAL(clicked()),
        this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()),
        this, SLOT(reject()));
  }
```

The `enableLoginButton()` slot is not listed here, since it's identical to the slot of the same name in the original version of the `LoginDialog` class.

One of the main advantages of Qt Designer is that the code generated by `uic` is kept totally separate from the application's hand-written code. This gives you the flexibility to change your user interface without needing to rewrite the code or fearing that your modifications to generated code will be lost.

This completes our review of creating dialogs with Qt. In the next article, we will see how to create custom widgets with any look and behaviour we want.

Jasmin Blanchette

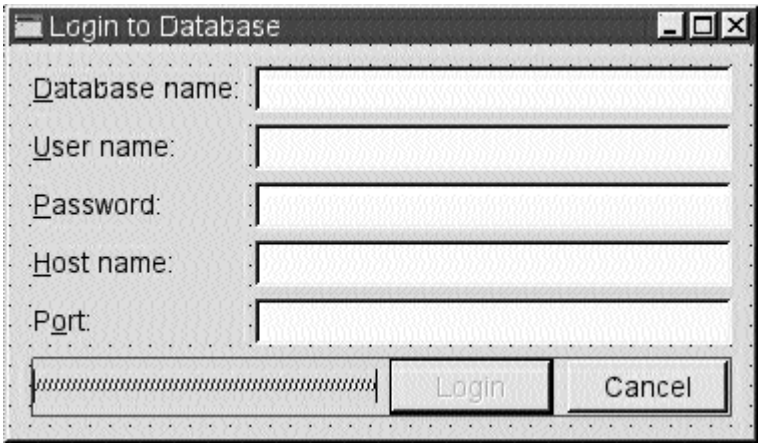


Figure 5: The dialog with layout

Now, let us start with the example of the `vector` container.

A `vector` is very similar to an array but has more advanced features, some of which are utilized in the example below.

```
#include<iostream>
#include<vector>

using namespace std;

int main() {
  /* I'll now create a vector of integers. It
     is instantiated here, so vint can hold
     integers */
  vector<int> vectorint;
  vectorint.reserve(5);

  /* reserve pre-allocates memory for holding
     five integers*/

  for(int j = 1; j < 6; j++)
    vectorint.push_back(j);

  typedef vector<int> Vector_Of_Ints;
  for(Vector_Of_Ints::const_iterator i
                          = vint.begin();
      i != vint.end(); ++i)
    cout << *i << "\t";
    cout << endl;
}
```

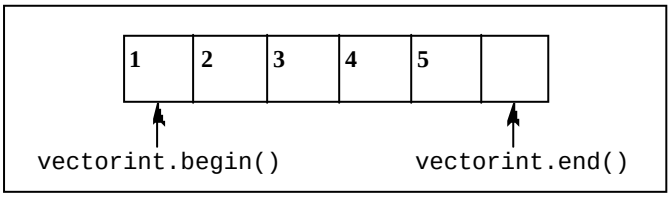
It should be noted that the `reserve` function allocates memory but the allocation is more akin to that of an array than using `new`. The capacity of a `vector` is defined as the *minimum number of objects that it can hold.* The reserve `method` makes sure that the `vector` has a capacity *greater than or equal to its argument* [3]. In the above example, after this statement

```
vectorint.reserve(5);
```

the `vector vectorint` has a capacity of at least 5.

The `push_back()` method function pushes the five integers into the `vector`. An iterator behaves in the same way as a pointer but is more generic. Note that the iterator is made a constant in this example because an iterator seldom modifies the contents of its `vector`.

In the above example, `vectorint.begin()` points to the first element of the `vector`, which is `1`. `vectorint.end()` points to an element which is after the last element of the `vector`, which is `5` in this case. Therefore, we cannot dereference `vectorint.end()`. This is as shown below:



When we dereference the iterator using the `*` operator, we get the value stored at the address to which the iterator points to.

When the above code is executed, gives the following output:

```
1    2    3    4    5
```

An even more convenient way of printing a `vector` would be to use the `copy` algorithm as shown below:

```
copy(vint.begin(), vint.end(),
     ostream_iterator<int>(cout, "    "));
```

which basically means: *copy the contents of the* `vector` *starting from* `vint.begin()` *to* `vint.end() -1` *(remember that* `vint.end()` *points to an element* ***after*** *the last element of the* `vector`*) to the standard output (*`cout`*) separating each of the numbers with four space characters.*

The header `<algorithm>` must be included for the `copy` algorithm to work.

Of course, the `vector` can be of any type, `int`, `char` or even another class such as `string`, and they are all handled in the same way. This is possible because a `vector` is a generic container, or in other words, a `vector` is a template class. For example after the statement

```
vector<T> foo;
```

`foo` can hold objects of type `T`. Therefore, `T` can even be any class.

Consider a normal class `foo`. It will have a standard constructor, destructor, some methods, and a couple of variables.

```
class foo {
public:
  foo();
  foo(int a, int b);
  foo(int a, double b);
  ~foo() {};
  int showValue() {
    return something;
  }
private:
  int something;
};
```

A template class can be thought of as being the same, except that now when we instantiate it, we do so with an unspecified type `T` rather than an explicit `int`, `char`, etc. Everything else remains the same (more or less).

```
template <typename A, typename B>
class foo {
public:
  foo();
  foo(A a, A b);
  foo(A a, B b);
  ~foo() {};
  A showValue() {
    return something;
  }
private:
  A something;
};
```

Next let us consider the `string` class. A `string` container is very similar to that of a `vector` class. A `string` object can be declared as follows:

```
std::string sobject;
```

A `string` object can be initialized in some of the following ways (there are plenty of other ways)

```
1 string sobject("hello");
2 string sobject = "hello";
3 string s1 = "Hello, ";
4 string sobject = s1;
5 string s2 = "World";
6 string sobject =s1 + s2;
```

In the first form of initialization the constructor of the `string` class is invoked with the value of `"hello"` and therefore the `sobject` has the value of `"hello"` once this line is executed.

In the second, third and fifth forms of initialisation, a string is assigned to the `string` object and so the `string` objects will hold the respective strings after these statements are executed.

In the fourth and fifth form of initialization, the copy constructor of the `string` class is invoked in order to copy the contents of the `string`s at the right hand side to the `string` objects.

Let us take another example which uses both the `string` and `vector` class to understand how both of them work together.

```
// vecstringSimple.cpp

#include<iostream>
#include<string>
#include<vector>
#include<iterator>

using namespace std;

int main() {
  vector<string> vector_of_strings;
  vector_of_strings.reserve(5);
  string text;
  cout << "\n";
  cout << "Enter the strings\n";
  cout << "\n";
  for (int a = 0; a < 5; ++a) {
    cin >> text;
    vector_of_strings.push_back(text);
  }

  cout << "\n";
  cout << "Output of the program\n";
  cout << "\n";
  for(vector<string>::iterator i
            = vector_of_strings.end()-1;
      i >= vector_of_strings.begin();
      i--)
    cout << *i << "\n";
}
```

In the above example a `vector` of `string`s is created and `string`s are read into the `vector` until the input is `end`. The `string`s are then output in the reverse order in which they were entered, as shown below

Enter the strings:

```
Rajanikanth
Ravikanth
Srimannarayana
VijayaLakshmi
hello
```

Output of the program:

```
hello
VijayaLakshmi
Srimannarayana
Ravikanth
Rajanikanth
```

Let us take a more complex example which uses both the `string` and `vector` classes.

```
// vecstr.cpp

#include<iostream>
#include<vector>
#include<algorithm>
#include<iterator>
#include<string>

using namespace std;

int main() {
  vector<string> vector_of_strings;
  string s;

  cout << "Enter the strings to be sorted: "
       << "\n";
  while(getline(cin,s) && s != "end")
    vector_of_strings.push_back(s);

  sort(vector_of_strings.begin(),
       vector_of_strings.end());
  vector<string>::const_iterator pos
        = unique(vector_of_strings.begin(),
                 vector_of_strings.end());
  vs.erase(pos, vector_of_strings.end());
  copy(vector_of_strings.begin(),
       vector_of_strings.end(),
       ostream_iterator<string>(cout,
                                "\t"));
  cout << '\n';
}
```

Let us first understand how this code works and then we will look at how it runs.

First of all, the line

```
vector<string> vector_of_strings;
```

declares `vector_of_strings` as a `vector` of `string`s.

Next, the condition

```
while(getline(cin,s) && s != "end")
```

means read a line of input from `stdin` as a `string` until the input is `end`. So, `"end"` cannot be an input string for this program. Next, the program pushes each of these `string`s into the `vector` `vector_of_strings`.

The algorithm sorts the `string`s in the `vector` `vector_of_strings` in ascending order. The `unique` algorithm moves all but the first `string` for each set of consecutive `string`s to the end of the unique set of `string`s in the `vector` container. It returns an iterator which points to the end of the unique set of `string`s; in our code this iterator is stored in `pos`. Next, we call the `erase` iterator which actually deletes the duplicate elements from `pos` to `vector_of_strings.end()`.

The `copy` algorithm then copies the unique set of `string`s to the standard output.

Following is the output of the program `vecstr.cpp`

```
Enter the strings to be sorted:
Srimannarayana Jammalamadaka
VijayaLakshmi Jammalamadaka
Rajanikanth Jammalamadaka
Ravikanth Jammalamadaka
aaaaa
a
a
k
k
end

Rajanikanth Jammalamadaka
        Ravikanth Jammalamadaka
        Srimannarayana Jammalamadaka
        VijayaLakshmi Jammalamadaka
        a       aaaaa   k
```

We will discuss a more complicated program using the `vector` and `string` containers in the next article.

Rajanikanth Jammalamadaka

References

[1] Schneider, M. and J. Gersting (1995), *An Invitation to Computer Science*, West Publishing Company, New York, NY, p. 9.

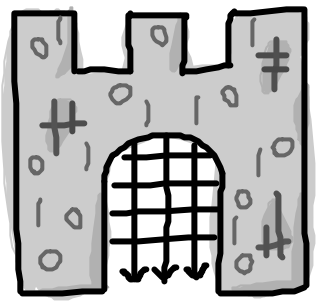
[2] Stroustrup, B., *The C++ Programming Language (Special Edition)*, Addison Wesley, p. 41.

[3] Sutter, H., *Exceptional C++ Style : 40 New Engineering Puzzles, Programming Problems, and Solutions (C++ in Depth Series)*, Pearson Education; (July, 2004).

Professionalism in Programming #28

An Insecurity Complex (Part One)

Pete Goodliffe <pete@cthree.org>



Security is mostly a superstition. It does not exist in nature... Life is either a daring adventure or nothing.

Helen Keller

Not so long ago computer access was a scarce commodity. The world contained only a handful of machines, owned by a few organisations, accessed by small teams of highly trained personnel. In those days computer security meant wearing the right labcoat and pass card to get past the guard on the door.

Fast forward to today. We carry more computational power in a pocket than those operators ever dreamt of. Computers are plentiful and, more pertinently, highly connected.

The volume of information carried by computer systems is growing at a fantastic rate. We write programs to store, manipulate, interpret, and transfer this data. Our software must guard against data going astray: into the hands of malicious attackers, past the eyes of accidental observers, or even disappearing into the ether. This is critical; a leak of top-secret company information could spell financial ruin. You don't want sensitive personal information (your bank account or credit card details, for example) leaking out for anyone to use. Most software systems require some level of security[1].

Whose responsibility is it to build secure software? Here's the bad news: it's *our* headache. If we don't consider the security of our handiwork carefully, we will inevitably write insecure, leaky programs and reap the rewards.

Software security is a really big deal, but generally we're very bad at it. Nearly every day you'll hear of a new security vulnerability in a popular product, or see the results of viruses compromising system integrity.

This is an enormous topic, far larger than we have scope to go into here. It's a highly specialised field, requiring much training and experience. However, even the basics are still not adequately addressed by modern software engineering teaching. The aim of this series is to highlight security issues and explore the problem. We'll learn a number of basic techniques for protecting our code.

Why Do We Get It So Wrong?

Building secure software requires a mindset that is sadly lacking in the average programmer. In the day-to-day madness of the software factory we're too focused on getting the program working, on getting it out of the door on time and in a reasonable state. We sit back and breathe a sigh of relief when our streamlined application appears to be doing what its supposed to. Rarely do we turn our attention to how secure the code is. Unless the test department is particularly skilled in this area, it's easy to ignore the whole issue – we'd rather not think the worst of a new creation.

If you do eventually turn your gaze to security issues, perhaps with a little test department prodding, it's probably too late anyway. Once a system is built, patching up any security problems is a hard job; the problems are either too fundamental, too prevalent, or far too hard to identify.

It's probably hard to believe that anyone would take the time and effort to hack your applications. But these people exist. They're talented, motivated, and they are very, very patient. Why do they do it? Some malicious crackers intend to steal, commit fraud, or cause damage, but their motive can equally be to prove superior skills or to cause a little mischief. They might not want to compromise your application specifically, but won't hesitate to exploit its flaws if you leave a hole open.

Sadly, no application is totally hack-proof. Writing a secure program is no easy task. Yet even the most secure application must run in its operating environment: under a particular OS, on some specific piece of hardware, on a network, and with a certain set of users. An attacker is just as likely to compromise one of these as your actual code. Indeed they're probably more likely to; *social engineering* – the art of acquiring important information from people, items in an office, or even the outgoing trash – is usually a lot easier (and often quicker) than worming a way into your computer system.

Software security presents a myriad of problems and challenges for the poor overworked programmer.

The Risks

Better be despised for too anxious apprehensions, than ruined by too confident security.

Edmund Burke

Why would anyone bother to attack your system? It's usually because you've got something that they want. This could be:

- your processing power,
- your ability to send data (e.g. send spam emails),
- your privately stored information,
- your capabilities; perhaps the specific software you have installed, or
- your connection to more interesting remote systems.

They might even attack you for the sheer fun of it, or because they dislike you and want to cause harm by disrupting your computer resources. Of course, we must remember that whilst malicious people *are* lurking around looking for easy, insecure prey, a security vulnerability might equally be caused by a program that accidentally releases information to the wrong audience. Sometimes this won't matter. More often it's just embarrassing. In the worst case, though, that lucky user can opportunistically exploit the leak and cause you harm.

To understand the kinds of attack you might suffer, it's important to mark the difference between protecting an entire computer *system* (comprising of several computers, a network, and a number of collaborating applications) and writing a single secure *program*. Both are important aspects of computer security; they blur together since both are necessary. The latter is a subset of the former. It takes just one insecure program to render an entire computer system (or network) insecure, so we must take the utmost care at all times.

Let's take a look at the ways you can be caught with your pants down. These are some of the most common security risks and compromises of a live, running computer system:

- **Physically acquiring a machine**, for example by stealing a laptop or PDA containing unsecured sensitive data. This data is freely readable by anyone with the inclination. Similarly, the stolen device might be configured to automatically dial into a private network, allowing a simple route straight through all your company's defences. This is a serious security threat, and one that you can't easily guard against in code! What we can do is write systems that aren't immediately accessible to computer thieves.

- **Exploiting flaws in a program's input routines**. Not checking input validity can lead to many types of compromise, even to the attacker gaining access to the whole machine. We'll see examples of this later.

- **Breaking in through an unsecured public network interface** is a specific variant of the previous point. This is particularly worrying. UI flaws can only be exploited by people actually *using* that UI, but when your insecure system is running on a public network the whole world could be trying to break down your doors.

A Promise Kept

In the last article I promised you the answer to my riddle: *how many programmers does it take to change a light bulb?* How many answers did you come up with? Here are mine:

1. None. The bulb's not broken. It's a power saving feature.
2. Just the one, but it will take all night and an inordinate amount of pizza and coffee.
3. Twenty. One to fix the initial problem, and nineteen to debug the resultant mess.
4. The question's wrong. It's a hardware problem, not a software one.

1 As we'll see, this is true whether they handle sensitive data or not. If a 'non-critical' component has a public interface then it poses a security risk to the system as a whole.

- **Malicious authorised users copying and sharing data** they're not supposed to. It's hard to guard against this one. You have to trust that each user is responsible enough to handle the level of system access they've been designated.

- **Malicious authorised users entering bad data** to compromise the quality of your computer system. Any system has a small set of trusted users. If they're not trustworthy then you can't write a program to fix them. This shows that security is as much about administration and policy as it is about writing code.

- **Setting incorrect permissions**, allowing the wrong users to gain access to sensitive parts of your system. This could be as basic as setting the correct access permissions on database files so casual users can't see everyone's salary details.

- **Privilege escalation**. This occurs when a user with limited access rights tricks the system to gain a higher security level. The attacker could be an authentic user, or someone who has just broken into the system. Their ultimate aim is to achieve *root* or *administrator* privilege, where the attacker has total control of the machine.

- **"Tapping into" data** as it is transmitted on the wire. If communication is unencrypted and traverses an insecure medium (e.g. the internet) then any computer en route can syphon off and read data without anyone else knowing. A variant of this is known as a *man-in-the-middle* attack – when an attacker's machine pretends to be the other communicant and sits between both senders, snooping on their data.

- **Virus attacks** (self-replicating malicious programs, commonly spread by email attachment), trojans (hidden malicious payloads in seemingly benign software), and spyware (a form of trojan that spies on what you are doing, the webpages you visit, etc). These programs can capture even the most complex password with keystroke loggers, for example.

- **Careless users** (or bad system design) can leave a system unnecessarily open and vulnerable. For example, users often forget to log off, and if there is no session timeout anyone can later pick up your program and start using it.

- **Storing data 'in the clear'** (unencrypted). Even leaving it in memory is dangerous; memory is not as safe as many programmers think. A virus or trojan can scan computer memory and pull out a lot of interesting titbits for an attacker to exploit. This depends on how secure your OS is – does it allow this kind of memory access and can you lock your applications's memory pages manually?

- **Copying software**. For example: running multiple installations in an office only licensed for one user, or allowing copies to spread on the internet.

- **Allowing weak, easily guessable passwords**. Many attackers use dictionary-based password cracking tools that fire off many login attempts until one works. It's a sad fact that easily memorable passwords are also easily guessable passwords. More secure systems will suspend a user account after a few unsuccessful logins.

- **Out-of-date software installations**. Many vendors issue security warnings and software patches. They come at a phenomenal rate and should really be carefully checked before being deployed. A computer system administrator can easily fall behind the cutting edge.

The problem scales as the number of routes into a system grows. It gets worse with: the more input methods (web access, command line, or GUI interfaces), the more individual inputs (different windows, prompts, web forms, or XML feeds), and the more users (there is more chance of someone discovering a password). With more outputs there is more chance for bugs to manifest in the display code, leaking out the wrong information

How do you know when your program has been compromised? Without detection measures you'll have no idea – and will just have to keep an eye out for unusual system behaviour or different patterns of activity. This is hardly scientific. A hacked system can remain a secret indefinitely. Even if the victim (or their software vendor) *does* spot an attack, they probably don't want to release detailed information about it to invite more intruders.

Cracker vs Hacker

These two terms often get confused and used inappropriately. Let's stop briefly and set the record straight. Their correct definitions are:

Cracker: Someone who purposefully attacks computer systems by exploiting their vulnerabilities to gain unauthorised access.

Hacker: Often used incorrectly to mean cracker, a hacker is really someone who 'hacks' at code. This is a term used with pride by a particular breed of programming geek. A hacker is a computer expert or enthusiast.

What company would publicise that their product has security issues that are effectively wide-open doors? If they are conscientious enough to release a security patch not everyone will upgrade, leaving a well-documented security flaw in many operational systems.

The Opposition

To defend yourself adequately it's important to know whom you're fighting against. As they say: *know your enemies*. We must understand exactly what they're doing, how they do it, the tools they're using, and their objectives. Only then can we formulate a strategy to cope.

Your attacker might be a common crook, a talented cracker, a 'script kiddie'[2], a dishonest employee cheating the company, or a disgruntled ex-employee seeking revenge for unfair dismissal.

Thanks to pervasive networking they could be anywhere, in any continent, using any type of computer. When working over the internet attackers are very hard to locate; many are skilled at covering their tracks. Often they crack easy machines to use as a cover in more audacious attacks.

They could attack at any time, day or night. Across continents one person's day is another's night. You need to run secure programs around the clock, not just in business hours.

There is a cracker subculture where knowledge is passed on and easy-to-use cracker tools are distributed. Not knowing about this doesn't make you innocent and pure, just naive and open to the simplest of attack.

With such a large bunch of potential attackers, the motives for an attack are diverse. It might be malicious (a political activist wants to ruin your company, or a thief wants to access your bank account), or it might be for fun (a college prankster wants to post a comical banner on your website). It might be inquisitive (a hacker just wants to see what your network infrastructure looks like, or practice their cracking skills), or might be opportunist (a user stumbles over data they shouldn't see, and works out how to use it to their advantage).

Excuses, Excuses

How do attackers manage to break into code so often? They're armed with weapons we don't have or (due to lack of education) know nothing about. Tools, knowledge, skills: these all work in their favour. However, they have one key advantage that makes all the difference: time. In the heat of the software factory, programmers are pressed to deliver as much code as humanly possible (probably a little bit more), and to do so on time: or else. This code has to meet all requirements (for functionality, usability, reliability, etc) leaving precious little time to focus on other 'peripheral' concerns, like security. Attackers don't share this burden, have plenty of time to learn the intricacies of your system, and have learnt to attack from many different angles.

The game is stacked heavily in their favour. As software developers we must defend all possible points of the system; an attacker can pick the weakest point and focus there. We can only defend against the known exploits; an attacker can take their time to find any number of unknown vulnerabilities. We must be constantly on the lookout for attacks; the attacker can strike at will. We have to write good, clean software that works nicely with the rest of the world; an attacker can play as dirty as they like.

What does this tell us? Simply that we *must* do better. We must be better informed, better armed, more aware of our enemies, and more conscious of the way we write code. We must design in security from the outset and put it into our development processes and schedules.

Next Time

We'll conclude this topic by investigating some specific code vulnerabilities, and working out good techniques to defend our code from attack.

Pete Goodliffe

2 A derogatory name for crackers who run automated 'crack er scripts'. They exploit well-known vulnerabilities with little skill themselves.

Wx – A Live Port

Part 1: The Rationale

Jonathan Selby <jon@xaxero.com>

Moving beyond MFC and opening new horizons with wxWidgets.

This is a collection of notes I have made while porting an application from MFC to wxWidgets. It is intended partly as a tutorial and partly to document some of the roadblocks I met on the way. Right now it has gaping holes in it that are gradually being filled.

When I upgraded from DOS to Windows 3.1 and purchased Microsoft Visual C++ version 1, a whole new world opened up. The massive framework allowed me to be writing fully fledged GUI applications in a few short weeks and a world of creative opportunity was opened up

The software was exceptional value coming with a formidable array of manuals that I would leaf through in my spare time. Over the years I and my software company built up a large software library of applications related to marine weather, communication and navigation, and our products sold well on the world market. With recent events in the software world and a global shifting towards Linux as a viable alternative there is a growing need for code that is portable.

As this trend seemed to gather momentum, I started to refine my search and start to look more actively for avenues to port my software. With the launch of Visual Studio .NET I had to make a decision. Most of what .NET was about seemed to lack co-ordination and it did not seem to have relevance to ships at sea far away from broadband internet. So I chose to stop with Visual Studio 6 and keep up to date with MSDN subscriptions. Not allowed – my next set of disks had "For Use with Visual Studio .NET only" written on them. And that was the end of that – I was out in the pasture and in need of a new approach to keep 20 man-years of work viable long term.

I looked again at Linux and KDevelop, to mention a few environments. While I am extremely encouraged by the progress Linux is making it is still not up to Windows when it comes to bringing new users in, and hardware installation though robust is still extremely intimidating if you are not an expert. The documentation is excellent but it does need somebody with a leaning towards systems to get into it. This is changing though very rapidly and I am predicting that Linux will overtake Windows in the very near future as the momentum builds up. So this makes a port even more urgent.

I discovered wxWidgets almost by accident. A friend mentioned in passing and I visited the website, downloaded the kit and started to play with it.

The Initial Impression

I started with version 2.4.0 and compared it to MFC. I was surprised to discover that the framework is 10 years old, very mature and stable. Quite a few hands have contributed over the years. Most important it is all source code, well written and intuitive. Well the first job I had to do was compile it all. Oh dear – here we go I thought, gritting my teeth, and selected `Batch build` from the Visual Studio options. I went off to make a cup of tea and almost spilled it over the keyboard when I got back because the whole thing had compiled properly and built all the libraries. A very encouraging start.

So what could the kit do?

I ran up the samples and they all compiled. I looked at the code and although the architecture was not the same as I was used to I could see the similarities. I was used to the Stygian macros and confusing comments App Wizard puts in. Not being too concerned with the nitty gritty, I would normally let App Wizard build a template and I would plug in the bits as I learned in my old scribble tutorial many years ago. So it took a little bit of exploring and tracing to see what was actually going on.

At this point I went back on the internet to see if I could find some resources.

I found two excellent articles by Marcus Neifer: *Porting MFC Applications to Linux*, and also a good wxWidgets primer: *Looking Through wxWidgets: An Intro to the Portable C++ and Python GUI Toolkit.*

These two articles convinced me that there was something worth pursuing here. So I started my evaluation in earnest to see how the kit could meet our needs. Firstly I looked at the DocView MDI sample that emulates the Scribble sample that all MFC trainees used to start with. The code is a good deal lighter in weight though. I was particularly interested to see if all the little nuances our clients were used to were supported. Tool tips and Help on the status bar ware two I items I went looking for. Menu creation and toolbar interaction with the code were other areas that differed quite a bit from the old MFC way of doing things. I could not find an easy way to make the toolbar dockable but I did not look very hard – that is a bit of a gimmick at least in our apps. There was the option of trying to convert the MFC `.rc` to the new XML format that wxWidgets is using. I had only very limited success with that route and preferred to go the conservative mothod of putting everything in C++ code, I followed that methodology. In my opinion clearly commented code beats everything else when it comes to reviewing something a few years down the road.

Next the interaction with the users was examined. I latched on to wxDesigner as the tool of choice. The tool uses a completely different concept to the VC++ method of constructing a dialog. It uses what we call sizers to group and encapsulate controls in boxes or groups. Once you get the hang of it, it is a very quick way of creating an attractive looking dialog. The software also generates a wrapper so you can run the dialog as a stand alone program. An excellent way to jump start an application.

However my main focus was to take an existing application and port it to wxWidgets. This required an in-depth view as to how the two architectures differ. It also needs a thorough review of all the underlying capabilities of the GUI, how the tool bars and menus behave, what context help support is around. One of the biggest advantages of Windows is the redundancy of the help system and the embedded context-sensitive actions that are available.

Navigating the Library

Now we have decided that wxWidgets as the tool of choice, we now have to come to grips with it. In my trainee programming days I was taught to help myself and to find the resources I need in the documentation. In those days this meant poring over voluminous books. Today of course the CD-ROM and HTML have made this all a lot easier. The wxWidgets help file comes in many flavours. My choice is the HTML help version. Now we are going to be reading this a lot. Unlike many programming tomes, I am not proposing to fill out the pages with vast chunks from the manual. You are going to have to go out there yourself and find it.

The first thing I did was to add the wxWidgets help into the MSVC++ tools menu – an area where you can add custom features.

In the tools menu:

- Go to `Customize`
- Go to the `Tools` tab and add an entry `wxWidgets help`
- Create a keyboard shortcut `SHIFT F1`
- Add the the following command: `hh.exe`
- Add as arguments:
  `C:\wxWidgets_2.4.0\docs\htmlhelp\wx.chm`

A Tour of the wxWidgets Documentation

The core of the wxWidgets documentation is the alphabetical class library reference. It is largely complete however some sections may be very terse. All the source code is available and if you are struggling – why not use the MSVC browser and take a look through some of the source code. Unlike some GUI libraries it is very clear and well written.

Some of the tutorials I have mentioned to date can help although to get a basic understanding of the framework components I strongly recommend you take a tour of the samples supplied. They are clear and address a very specific point. As a last resort there is is `wx-Users@lists.wxwidgets.org`. Here you will get an answer to a question very quickly. Several of the sticking points I encountered were answered, once within 3 minutes on a Sunday. Try getting commercial support of that quality!

The downside is that you need to be patient and do not expect anything. The open source model is a two-edged sword. The contributors are not paid.

Let's Create a Window

This is the very basis of a nugget of user interface code.

The class you will be using is `wxFrame` – derived from `wxWindow`. Like `CWnd`, wxWidgets has a huge bewildering array of sub functions that drive the application.

Here is the basic code to pop up a single window.

```
wxPoint pt(20,20); // where the window will
                   // appear
wxSize sz (600,400); // the window size

WxFrame *FrameWindow = new wxFrame;;
WxFrame->Create(frame, -1, "Key",
                pt, sz,
                wxDefault,
                "wxWindow sample");
```

A More Complex Window

Both MFC and wxWidgets support the Document/View approach. The concept here is the application launches a Main Frame that contains the primary controls. The Main Frame has Child Frames (in the Multiple Document Interface world) The Child Frame has views and and a class that manipulates the document and supports reading, writing, etc.

The Visual C++ App wizard makes one's life very easy by generating a bare bones application with the following components:

- Application registration – Associating a file type with the application.
- Drag and drop support of files onto the application main frame.
- Context sensitive help
- Command line processing
- Print and Print Preview

In wxWidgets all this has to be done yourself. Of course many simple applications do not need all this baggage, however if you are developing a full fledged application that meets the criteria of a modern day system you must include this functionality. We will start by laying out a bare bones framework and build our application from there.

The MFC Document template approach has close parallels. You draw up a document that has a filter for the `File open` and a specific extension. The framework uses this template to manipulate your data via file open, most recently used document etc.

In MFC the View is subclassed from the Child Frame, whereas in wxWidgets you need to explicitly create your own Frame. This was the most important difference I found and thus the `OnDraw` member of the view class had to be invoked from the Sub Frame Window class. I followed the design of the `docviewmdi` sample and used the `MyCanvas` class from there. Other little differences were the more modular way the menus work and especially getting the help tips to show on the main frame status bar. I ended up creating a second status bar on the Frame window for the time being for reading the menu help tips. It is a good idea to port the App routine first. Command line processing Drag and Drop, Document template, and Registry profile strings should be sorted out at the very first step. The wxConfig function is very useful here for storing profile data, windows size and other variables you need to store. Non persistent applications where you have to set everything up again when you run the program are a trademark of the amateur.

Since we have the view creating a child window we need to make sure it is closed when the view is closed. I followed the examples and used a scroll window called `canvas` locally. This code is lifted almost verbatim from the `DocViewMDI` sample that forms a reasonable basis for starting.

```
// Clean up windows used for displaying the
// view.
bool WXWindPlotView::OnClose(
                          bool deleteWindow) {
  if(!GetDocument()->Close())
    return FALSE;

  // Clear the canvas in case we're in single-
  // window mode, and the canvas stays
  canvas->Clear();
  canvas->view = (WXWindPlotView *)NULL;
  canvas = (MyCanvas *)NULL;

  wxString s(wxTheApp->GetAppName());
  if (frame)
    frame->SetTitle(s);

  SetFrame((wxFrame*)NULL);
  Activate(FALSE);

  if(deleteWindow) {
    delete frame;
    return TRUE;
  }
  return TRUE;
}
```

Once I had everything in place I could start dropping in chunks of code from the MFC app and let the compiler crank through the errors.

The Port in Detail

Step 1 was to remove all includes for `windows.h` and `stdafx.h`. Now we are are a huge step to throwing off the Microsoft yoke and out there in the real world at last. All those windows functions that are not in the wxWidgets library are going to have to be found in the the ANSI C libraries. The first one I came up against was `GlobalLock` to allocate memory. Several lines of code were replaced with a simple `malloc` statement – and a `free` in the destructor.

The sidebar contains a great little macro to speed up your work.

```
'————————————————-
'FILE DESCRIPTION: New Macro File
'————————————————-
Sub wxconvert()
'DESCRIPTION: Convert MFC to wxWindows
'Begin Recording
 ActiveDocument.ReplaceText "Bool","bool"
 ActiveDocument.ReplaceText "CString","wxString"
 ActiveDocument.ReplaceText "CFile","wxFile"
 ActiveDocument.ReplaceText "CTime","wxDateTime"
 ActiveDocument.ReplaceText ".GetLength",".Length"
 ActiveDocument.ReplaceText ".GetBuffer",".GetData"
 ActiveDocument.ReplaceText "CCmdUI* pCmdUI",
                            "wxUpdateUIEvent& event"
 ActiveDocument.ReplaceText "ON_COMMAND","EVT_MENU"
 ActiveDocument.ReplaceText "ON_UPDATE_COMMAND_UI",
                                    "EVT_UPDATE_UI"
 ActiveDocument.ReplaceText "       afx_msg ",""
 ActiveDocument.ReplaceText "CSize","wxSize"
 ActiveDocument.ReplaceText "AfxGetApp()->",
                                      "wxGetApp()."
 ActiveDocument.ReplaceText "AfxMessageBox",
                                     "wxMessageBox"
 ActiveDocument.ReplaceText "CFrameWnd","wxFrame"
 ActiveDocument.ReplaceText "::modeRead","::Read"
 ActiveDocument.ReplaceText "DWORD","unsigned int"
 ActiveDocument.ReplaceText "GlobalAlloc(
                    GMEM_MOVEABLE | GMEM_ZEROINIT,",
                           "(unsigned char *) malloc"
 ActiveDocument.ReplaceText "GlobalAlloc(
                   GMEM_MOVEABLE | GMEM_ZEROINIT, ",
                           "(unsigned char *) malloc"
 ActiveDocument.ReplaceText "LPSTR","char *"
 ActiveDocument.ReplaceText "pCmdUI->SetCheck",
                                      "event.Check"
 ActiveDocument.ReplaceText "pCmdUI->","event."
 ActiveDocument.ReplaceText "CPen","wxPen"
 ActiveDocument.ReplaceText "CBrush","wxBrush"
 ActiveDocument.ReplaceText "CRect","wxRect"
 ActiveDocument.ReplaceText "CPoint","wxPoint"
 ActiveDocument.ReplaceText "pDoc->","doc->"
 ActiveDocument.ReplaceText "->TextOut","->DrawText"
 ActiveDocument.ReplaceText "->LineTo","->DrawLine"
 ActiveDocument.ReplaceText "DWORD","unsigned int"
'End Recording
End Sub
```

Most wxWidgets classes are counterparts of MFC and there is normally a 1-to-1 correlation. `wXstring` and `CString` are very similar as is `DC` – `wxDC`. Others are not quite the same. `CTime` is replaced by `wxDateTime` that has slightly different construction and considerably more functionality. `MoveTo` and `LineTo` are replaced by one call in wxWidgets – `DrawLine`. Eventually somebody will write a porting macro. I was tempted to batch some edit commands together but resisted. The process went quite fast and it is better to keep an eye on where the code is changing.

One thing to look out for is making sure you have all the right includes. If a class is not found whizz over to the documentation. Every class has a `#include` associated with it.

It is certainly not a plug compatibility and where there is a difference you can be sure that wxWidgets is more intuitive and better thought out. Best of all the compiler does all the hard work. I linked the Shift F1 key in Visual Studio to bring up the wxWidgets help file. It is very complete and it is very easy to jump to the relevant section and find out what each class is about. In a very short time I was reaming out whole sections of code and getting them to run. File IO was simplified. `try` and `catch` were not supported but then I do not use exceptions much.

Certain things to look out for: `wxString::Format` has the same syntax as the MFC equivalent but with a difference you need to assign it i.e.

```
CString str;
str.Format("%i",Integer);

wxString str;
str = str.Format("%i",Integer);
```

I personally like to expand out my code for readability and thus the wX implementation makes more sense.

Another difference is the drawing functions of `wxDC`.

Functions like ellipse (`DrawEllipse`) use starting coordinates and dimension as opposed to starting and ending coordinates.

`wXDC::DrawText` is similar to `CDC::TextOut` but the text only is drawn making an initial rectangle draw necessary to wipe out the old text and avoid overlaying. This in fact solved a lot of irritating quirks I experienced with MFC and trying to get a mixture of text and graphics to stop interfering with each other.

Name	Value	Name	Value
wxID_LOWEST	4999	wxID_PASTE	5032
wxID_OPEN	5000	wxID_CLEAR	5033
wxID_CLOSE	5001	wxID_FIND	5034
wxID_NEW	5002	wxID_DUPLICATE	5035
wxID_SAVE	5003	wxID_SELECTALL	5036
wxID_SAVEAS	5004	wxID_FILE1	5050
wxID_REVERT	5005	wxID_FILE2	5051
wxID_EXIT	5006	wxID_FILE3	5052
wxID_UNDO	5007	wxID_FILE4	5053
wxID_REDO	5008	wxID_FILE5	5054
wxID_HELP	5009	wxID_FILE6	5055
wxID_PRINT	5010	ID_FILE7	5056
wxID_PRINT_SETUP	5011	ID_FILE8	5057
wxID_PREVIEW	5012	wxID_FILE9	5058
wxID_ABOUT	5013	wxID_OK	5100
wxID_HELP_CONTENTS	5014	wxID_CANCEL	5101
wxID_HELP_COMMANDS	5015	wxID_APPLY	5102
wxID_HELP_PROCEDURES	5016	wxID_YES	5103
wxID_HELP_CONTEXT	5017	wxID_NO	5104
wxID_CUT	5030	wxID_STATIC	5105
wxID_COPY	5031	wxID_HIGHEST	5999

Table 1: Symbolic Event IDs

So you see that there is some work to do here and it is not a straight conversion exercise at all. However the compiler is a great help and guides your work. My methodology was to open up little bits of functionality at a time. Now the first port is behind me I will probably adopt a more confident holistic strategy.

Designing the User Interface

I started creating the toolbar manually from the samples. This is reasonably easy but is time-consuming and takes a bit of care as the coding is quite critical in places. Using wXDesigner automates this process. The menus and tool bars can be very easily laid out .

To generate an app from scratch using existing bitmaps and menu structures from the old program took an hour and a half with 15 toolbar buttons and 8 main menus.

Visual C++ is a bit easier and more intuitive but of course you are only coding for one platform. After a few minutes with wXDesigner you will get the hang of it and you will find it is a very much worth the money you have paid for it.

You need to be aware of the symbolic event Ids – some of them exist already as system events. See Table 1 for these

You need to add to the file `Resource.h` to manually code in the ID number where your user interface interacts with the above functions You can put the ID anywhere in a header file, however, I like to use `Resource.h`.

Otherwise you leave the ID as –1 and wXDesigner will generate the event entries for you. wXDesigner starts from ID=10000. All you need to do is use the symbols in your code.

To generate the CPP code press the C++ button in the wXDesigner and the file `nnnn_wdr.cpp` is generated, this file contains all the nasty menu generation code you used to have. Our Menu bar was called `MainMenuBarFunc`. All you need to do to invoke it in your main app is to put the following line in the `App::OnInit()` section:

```
m_mainFrame->SetMenuBar(MainMenuBarFunc());
```

wXDesigner has already created a shell for you but you probably want to use your own template. The class browser in MSVC is very useful. The code generated by wXDesigner does not parse well and all the functions are in one file. The wrapper wXDesigner generates it seems is primarily an example shell rather than a starting point for a GUI application and so far we have been evolving a full scale MDI program. The shell provided though is very useful to explain where all the bits plug in.

On the creation of pop-up menus, WXDesigner forces you to use a menu bar. You can try to override this however I found it easier to dump the generated code right into the `canvas` class. This code is fairly static and if we regenerate it is will be fairly easy to dump it in again.

This is what wXDesigner generates:

```
wxMenuBar *PopUpMenuBarFunc() {
  wxMenuBar *item0 = new wxMenuBar;
  wxMenu* item1 = new wxMenu(wxMENU_TEAROFF);
  item1->Append(wxID_NEW,
                wxT("&New\tCtrl-N"),
                wxT("New chart"));
  item1->Append(ID_GRIB,
                wxT("&Open\tCtrl-O"),
                wxT("Open a Grib File"));
  item1->AppendSeparator();
  item1->Append(ID_1X,
                wxT("&1x"),
                wxT("Zoom 1 times"));
  item1->Append(ID_2X,
                wxT("&2x"),
                wxT("Zoom 2 times"));
  item1->Append(ID_3X,
                wxT("&3x"),
                wxT("Zoom 3 times"));
  item1->Append(ID_4X,
                wxT("&4x"),
                wxT("Zoom 4 times"));
  item1->Append(ID_5X,
                wxT("&5x"),
                wxT("Zoom 5 times"));
```

```
  item0->Append(item1,
                wxT(""));
  return item0;
}
```

By stripping off the menubar class and implementing this directly in your code you get:

```
void MyCanvas::ShowDContextMenu(
                        const wxPoint &pos) {

  wxMenu* item1 = new wxMenu(wxMENU_TEAROFF);

  item1->Append(wxID_NEW,
                wxT("&New\tCtrl-N"),
                wxT("New chart"));
  item1->Append(ID_GRIB,
                wxT("&Open\tCtrl-O"),
                wxT("Open a Grib File"));

  item1->AppendSeparator();

  item1->Append(ID_1X,
                wxT("&1x"),
                wxT("Zoom 1 times"));
  item1->Append(ID_2X,
                wxT("&2x"),
                wxT("Zoom 2 times"));
  item1->Append(ID_3X,
                wxT("&3x"),
                wxT("Zoom 3 times"));
  item1->Append(ID_4X,
                wxT("&4x"),
                wxT("Zoom 4 times"));
  item1->Append(ID_5X,
                wxT("&5x"),
                wxT("Zoom 5 times"));

  PopupMenu(item1,
            pos.x,
            pos.y);

  // test for destroying items in popup menus
#if 0 // doesn't work in wxGTK!
  menu.Destroy(Menu_Popup_Submenu);
  PopupMenu(&menu,
            event.GetX(),
            event.GetY() );
#endif // 0
}
```

The above function is invoked via the `Canvas` class that handles all mouse events.

```
void MyCanvas::OnMouseEvent(
                       wxMouseEvent& event) {
  wxClientDC dc(this);
  PrepareDC(dc);
  wxPoint pt=event.GetLogicalPosition(dc);

  // Popup support
  if(event.RightUp())
    ShowDContextMenu(pt);

  // Standard mouse events
  if(event.LeftDClick())
    view->OnLButtonDblClk(0, pt);
  if(event.Moving())
    view->OnMouseMove(0, pt);

  if(!view)
    return;
}
```

Now we have a path where the UI can be altered on the fly and we just recompile. About as simple as using native MFC and Visual C++. Only difference is that you have a lot more control over what is going on.

Finer Points

Status Bar Under Sub-Window

Easy enough to do: in the `Frame` all you need to add is:

```
CreateStatusBar(4);
```

to create four equally sized panes, and then in your body code you set the text:

```
SetStatusText(_("Ready"),1);
```

to put the word `Ready` in the first pane.

Persistence and the Registry

The function `GetApp()` that returns addressability is accessed by simply putting `MyApp.h` in the view path of the class you want to allow access to the `App`. In the case of `MyDoc` putting `MyApp.h` in the front of the file and calling `GetApp()` you can access variables.

This is very important when we use profile variables.

Those from the golden days of windows programming will remember the ini file and `GetPrivateProfile` string and `WritePrivateProfileString`.

Persistence in wxWidgets is similar in concept except the concept is portable. For Win32 we will use the windows registry and `wxRegConfig`. For Unix & Linux we will need to use the `fileConfig`. The underlying philosophy is the same so the definitions will change but the use of the config base will not. The wxWidgets document ion explains the concept very well.

In `MyApp.h` define the config base variable `config`.

This will be our point of contact for all our persistent variables.

```
wxRegConfig *config;
config = new wxRegConfig("MYKEY");
```

Now in the application code we can do something like this:

```
wxGetApp().config->Read("LAT",
                        &LAT,
                        (double)0);

wxGetApp().config->Write("LAT",
                         worklat)
```

Posting Messages

You can issue messages within the code very much like making a mouse click. Very useful for invoking functions or perhaps a timer:

```
wxUpdateUIEvent ev(ID_TIMER);
frame->GetEventHandler()->ProcessEvent(ev);
```

Next time...

Next time, we'll talk about connecting to the user interface.

Jonathan Selby

Resources

wxWidgets: `http://www.wxwidgets.org`
wxDesigner: `http://www.roebling.de/`
Another introduction to wxWidgets:
`http://www.all-the-johnsons.co.uk/accu/index.html`
Porting MFC to wxWidgets:
`http://www-106.ibm.com/developerworks/linux/library/l-mfc/`

An Introduction to Objective-C

Part 2 – Basic Principles

D.A. Thomas

Unlike C++, Objective-C is standard C with a small object-oriented extension; an Objective-C compiler will compile all conforming C code in exactly the same way as a C compiler.

Objective-C source files traditionally have the suffix `.m`; and with the Apple/NeXT compiler, a file that contains a mixture of Objective-C and C++ code (so-called Objective C++) needs to be suffixed with `.mm`.

Objective-C adds one type to the C language, namely `id`, which is a pointer to any object.

```
id myObject;
```

declares an object pointer with the name `myObject`.

If the class of the object is known, the above declaration is equivalent to:

```
MyClass *myObject;
```

which declares a pointer to an object of class `MyClass`. This syntax allows the compiler to perform static type-checking, so that the programmer can be informed at compile time that an object of that class does not support a particular operation.

In Objective-C, unlike C++, all objects are referenced by means of pointers.

Classes are declared as follows:

```
#import "BaseClass.h"

@interface MyClass : BaseClass
{
  // Instance variables
  int n;
  float x;
}

// Method declarations

// Class method
+ (id)new;

// Instance method with arguments
- (id)initWithValues:(int)nn and:(float)xx;

@end
```

This is the public interface of the class `MyClass`, which inherits from `BaseClass`.

By default, instance variables have 'protected' visibility; this means that they can be read and written to directly only by objects of the class in which they are declared or of a class inheriting from it. The default visibility can be varied by means of the compiler directives `@public` and `@private`; public variables are visible to all outside code, while private ones are accessible only to instances of the class in which they are declared.

A class method is prefixed by `+` and could be called in the following manner to create an instance of the class and its address stored in a variable:

```
MyClass *newObject = [MyClass new];
```

Every class is an object in Objective-C and typically contains methods, like `new` above, to create instances of itself.

The return type of a method and the type of its arguments are written within parentheses. If the type is not provided, type `id` is assumed. It is important to understand that type names within parentheses resemble the C syntax for 'casting' a value from one type to another, in this context no casting is involved.

Instance methods belong to an instance of the class and can access its instance variables. Their declarations are preceded by the token `-`.

Arguments are declared after keywords terminating in colons; the name of the method is understood to consist of all the keywords together with the colons. The instance method above has the name `initWithValues:and:`, and it is pronounced: 'init with values colon and colon'. The method could be invoked in the following manner:

```
[newObject initWithValues:42 and:12.576];
```

This style of naming methods will initially seem strange to those who are unfamiliar with Smalltalk, from which it is derived.

The implementation of a class is defined as follows:

```
#import "MyClass.h"

@implementation Myclass

+ (id)new
{
  // Allocate memory for an object and
  // initialise to default values.
  return [[super alloc] init];
}

// -init is a private method and is thus not
// declared in the interface.
- (id)init
{
  // Initialise self to default values.
  return [self initWithValues:0 and:0.0];
}

- (id)initWithValues:(int)nn and:(float)xx
{
  // Call BaseClass init method.
  [super init];

  // Initialise the instance variables and
  // return a pointer to this object.
  n = nn;
  x = xx;
  return self;
}

@end
```

The keyword `self` is a pointer to the current object; `super` is a directive that instructs the compiler to invoke a method in the superclass (i.e. the class from which the current one is derived, otherwise known as 'base class').

The normal way to invoke methods in Objective-C is the write the object name (known as the receiver) followed by the method name (with arguments, if required) between square brackets; the whole is known as a message expression.

The method name is called a 'selector' because the receiving object uses it to select the appropriate method for the required operation. The message expression:

```
[self initWithValues:0 and:0.0]
```

above could be described in English as: 'send self a message with selector `initWithValues` and parameters 0 and 0.0'.

The reader should now have enough information to know how to write Objective-C classes, how instances are created and how to invoke methods.

Apart from knowledge of supplied class libraries, little else is required in order to become a competent user of the language.

D. A. Thomas

Reviews

Bookcase

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

A Note from Francis

About ten years ago I had an appointment to see James Lake who was the proprietor of PC Bookshops in Sicilian Avenue, Holborn, London. I was a bit early so spent a few minutes browsing. While doing so I noticed a book *Morphing Magic*. Nothing would have made me check this book other than having an idle few minutes in a bookshop. However I found a brilliant book that used C to develop, among other things, a great little application for developing and displaying simple cartoons. It ran on pretty minimal hardware even by the standards of the day, MSDOS on a 286-based machine was enough.

Such chance findings are becoming increasingly rare. There is no longer a PC Bookshop on Sicilian Avenue. The company has morphed into Holborn Books and has move to Hampshire. It trades electronically so there is no longer an opportunity to simply browse and find the nugget of gold hidden in the thousand pieces of dross.

This loss of real bookshops where you can go and browse is a matter for concern. We need specialist shops rather than small departments in chain stores. Or alternatively we need better mechanisms for finding both what we are looking for, and the things that we did not even know were there.

Even when you know what you are looking for, browsing for titles on the web can be a very hit and miss affair. Try going to `www.amazon.co.uk` and typing in 'beginners programming' as search criteria. Now repeat the exercise for 'introduction programming' and 'beginner's programming'. Not that my book is listed in the top ten for both the latter criteria but completely ignored by the first search.

I do not even begin to have a solution to finding things that you do not know are there. This means that good sources of information are becoming increasingly important. Most magazines only review a couple of books per issue, and it is in the nature of things that those will be books that are likely to interest the average reader. But those same people will already search for suitable books in the subject area.

This makes large-scale collections of book reviews all the more valuable, but it also places an extra burden on such collections to be as complete as possible as regards their core topics. A comprehensive collection of reviews needs to be actually comprehensive.

In days gone by, ACCU reviews did a pretty good job at covering books for newcomers to C and C++, not least because I covered many of them. As an author of a book for newcomers to programming I no longer feel I should review the work of my potential competitors, but I also feel aggrieved that almost a year on from publication there is still no mention of my book on ACCU's website. I hope this will have been corrected by the time this goes to print, not because the book is by me, but because two ACCU reviewers took copies to review more than six months ago. Accepting the task of reviewing a book is not something casual to be fitted into the odd moment but is an increasingly serious undertaking with a commitment to both the author and the potential readers.

Prize Draw

Now to turn to something positive, and something you can all join in. I would like readers to do three things. First select the book that you have read that you think has been most underrated or overlooked. Just one, and I know that makes it hard for some but the effort of choosing can focus the mind. Of course there are no right answers but it will be interesting if some books turn up more than once (and if only three readers respond...)

The second thing is to choose a category (novice programmer, newcomer to C++, embedded systems developer, games developer, etc.) and list which books you would recommend given a) a budget of £100 ($180) and b) a budget of £250 ($450).

And lastly, given a budget of £2000 ($3600) list what software development tools and references you would take with you for a year's stay on a desert island. The desert island comes equipped with the essentials for life and electric power.

There will be a prize draw for all responses submitted to `francis@robinton.demon.co.uk` by midnight November 30th/December 1st Greenwich Mean Time. The size of the prize will depend on the number of entrants so being the only entrant won't win very much.

Francis

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

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

An asterisk against the publisher of a book in the book details indicates that Computer Manuals provided the book for review (not the publisher.) N.B. an asterisk after a price indicates that may be a small VAT element to add.
The mysterious number in parentheses that occurs after the price of most books shows the dollar pound conversion rate where known. I consider a rate of 1.48 or better as appropriate (in a context where the true rate hovers around 1.63). I consider any rate below 1.32 as being sufficiently poor to merit complaint to the publisher.

C & C++

Exceptional C++ Style by Herb Sutter (0 201 76042 8), Addison-Wesley*, 325pp @ $39.99/£30-99 reviewed by Francis Glassborow

I thought that readers of C Vu would like a quick review of Herb Sutter's latest volume in his Exceptional C++ series. As the author kindly sent me an autographed copy I can safely review it without being accused of skimming cream off the top of the pile of books awaiting review.

This book consists of 40 chapters in the same format as he used in the previous two volumes. Most chapters lead with a one or more Junior Guru questions (things that any competent local expert should be able to tackle, but too many cannot). All the chapters have at least one Guru question. Those require a great deal of expertise to get entirely right. A couple of the Guru questions might stretch even the author's understanding – I am not entirely convinced that everything he writes in the two chapters on `export` is correct.

After the questions come the author's answers and sprinkled among those are sound-bites masquerading as guidelines. Well every good guideline should be expressible as a sound-bite, the skill is in ensuring that the sound-bites are also good guidelines.

One of the features Herb's book shares with Scott Meyer's books is that they are written for normal C++ programmers who are sufficiently professional to want to understand what they are doing and want to write correct code.

A typical example is item 16 (Mostly Private) that has no Junior Guru question. Herb has his feet solidly on the ground in recognising just how extensive misunderstanding of visibility and access is among even pretty expert C++ programmers. This is one of Herb's characteristics that make him an exceptional (pun intended) author, he spends time learning about the things that cause real problems to practitioners in the field and then tries to address them.

I have heard people dismiss some of his writing as dealing with things that are far too difficult for the working programmer. Such dismissal is seriously mistaken; working programmers who think books such as this are beyond them should find some other job because they are accidents waiting to happen.

Now before I am accused of waffling again, let me draw this review to a close. If you already have Exceptional C++ and More Exceptional C++ you will already be planning to buy this book. If you do not have the previous books, buy those first. Whichever group you are in do not confuse this book with the soon to be published book on C++ Coding Guidelines which is co-authored by Herb Sutter and Andrei Alexandrescu. Time to start dropping hints to your loved ones because that book should be out in time for Christmas.

Visual C++ Optimization with Assembly Code by Yury Magda (1 931769 32 X), alist, 450pp + CD @ $39.95/£27-99
review by Francis Glassborow

The title immediately made me doubtful, turning to the back-cover only deepened my sense of unease. Here is the start of the back-cover blurb:

> Describing how assembly language can be used to develop highly effective C++ applications, this guide covers the development of 32-bit applications for Windows. Areas of focus include optimising high-level logical structures, creating effective mathematical algorithms, and working with strings and arrays…

My first problem is that optimisation is always something that should not be taken lightly, indeed we should avoid it unless testing shows that it is necessary. All forms of hand optimisation tend to make code more fragile and harder to maintain, going down to assembly level is an even further step in the direction of maintenance problems.

Attempting to use assembly code as a way to optimise high-level anything seems to be entirely wrong to me. From where I am sitting, assembly code belongs in the lowest layers if at all.

Writing mathematical functions is an extraordinarily skilled task, and one that I am more than willing to consign to talented library implementers. I have some sympathy with regards to working with strings were that to mean using `std::string`, but it does not, it refers to using nul-terminated arrays of `char`.

So let me turn to the content. The first thing that quickly becomes apparent is that for the most part the author is not actually writing C++ nor even Visual C++, he is writing C with a small spicing of C++ and a large dose of Microsoft extensions. What puzzles me is why he is using `_asm` instead of the C++ `asm` keyword where he is putting assembly code into C++ source code. Maybe that is a VC++ issue.

The assembly code parts of the book seem to be fine (but remember that for the last ten years my attitude to knowing assembler code for a machine is that its main value is in identifying bugs in the compiler. The author covers both free standing assembly code in their own modules that will be linked in at link time, together with assembly code embedded in C++.

The author's optimistic estimate of the potential improvement by using assembly code is around 17%. That should be balanced with the way that assembly code will kill some of the optimisations available to Microsoft's most recent link technology. I also think it is being more than a tad optimistic and based on measurement of limited parts of an application rather than on overall performance.

However let me accept the author's estimate, and then pose the question as to how many months would go by before the current hardware was providing more than a 17% performance improvement. There are very few applications where maximal performance is an absolute requirement. In most cases once a certain level of performance has been achieved further improvements are of little added value.

I think that a much shorter book showing the reader how to add assembly code to already well written C++ would be much more to the point. Even such a book would have (or should have) a very limited sale.

If you absolutely need to write assembly code for your C++ application and are already a good C++ programmer you can probably distil what you need to know from reading this book. However most C++ programmers would do better to spend the equivalent time improving the quality of their C++.

I guess the amateur games programmers might find something useful in this book. The professional ones have to worry about issues of portability which makes use of inline assembly code problematic.

You Can Do It– A beginner's introduction to Computer Programming! by Francis Glassborow & Roberta Allen (0 470 86398 6), Wiley, 353pp + CD @ $30/£19-99
reviewed by Ian Bruntlett

Months from now a more detailed review will be posted on the ACCU book reviews web site. This brief review is presented now to answer the question "Should I buy this book?".

"Should I buy this book?" Well, if you are a learner programmer or someone who wants to brush up their C++ skills, this is the book for you. It's not perfect but it is a very good book for beginners. Once you have mastered this book then you should consider buying "Accelerated C++ " (Koenig & Moo) or the "C++ Primer" by Lippman & Lajoie. After that then try "The C++ Programming Language" by Bjarne Stroustrup. Take a look at `www.wileyeurope.com/go/glassborow` or `www.spellen.org/youcandoit/` for more information.

YCDI! Is a book that will take months to read. It took months to review. It is split into chapters but should have been split up into parts. The first set of chapters provide intense study material and the remainder of the book continues at a gentler pace.

The approach taken by the author (Francis Glassborow, aka fgw) is to explain matters and then provide exercises to confirm that the reader has assimilated the subject material. Fgw thinks that programming is fun and provides tools and libraries (Using the MingW port of the Gnu C/C++ compiler and the Quincy IDE on the accompanying CD) so that the beginner can produce rapid results, boosting the student's morale. The library that comes with this book is called "playpen" and provides a canvas to display things on – this is good because it means that beginners gets something visual to look at early on in their career.

Flaws. One is the lack of Linux support – the book demands access to a Windows PC. This is a disappointment because I have seen plenty of messages on Linux mailing lists looking for help in learning to program. The other flaw is lack of information in using an interactive debugger. Another flaw is that fgw insists on pulling all the standard library names into the global namespace – the "using namespace std" command is an aberration and should not appear in a programming book. It is one of the few places where fgw provides bad code that has to be unlearnt later.

The book gives an authentic programming experience, the reader has to dig for certain details to get things going. If you are a complete beginner (with no one to help you) then I would suggest you join the `accu-general` mailing list and ask questions there.

VERDICT: Highly recommended for beginners and intermediate C++ programmers.

C# & Java

Beginning C# XML by Stewart Fraser & Steven Livingstone (1-86100-628-4), WROX*, 729pp @ $39.99/£28-99
reviewed by Paul Usowicz

One of my next tasks at work is to extend an application I am working on so that it can export XML files destined for an SQL server. Conveniently the application is written in C# and I am just beginning to learn XML so this book was eagerly awaited. To say this book does exactly what it says on the cover (Beginning C# XML – Essential skills for C# programmers) is spot on. Upon opening the package, I immediately read the first two chapters ('Why use XML with C#?' and 'Overview of XML'). Over the next couple of evenings I devoured the rest of the book typing in the examples where I felt them necessary.

The speed at which I read the book highlights several factors. Firstly, I was extremely keen to learn XML. Secondly, this book was exactly what I needed. Thirdly, this is a very well written book. I found the explanations very easy to understand and the examples were short enough to actually type in – more on that in a minute! As I am relatively new to C# and have never before used XML I cannot vouch for the authors technical validity or experience but the book provided enough advice to get my application up and running quickly and reliably.

As well as the XML format the book also covers reading and writing XML and various acronyms including DOM, XSLT, XML Schema and XPath. The book also addresses XML in ADO.net and web services. At the end of the book is a case study for a simple news portal.

Now on to the source code I hinted at earlier. As I said, most of the code was very short and not too much trouble to type in. I felt it best to get the source code anyway so duly went to the web site highlighted on the front of the book, `www.beginningdotnet.com`. Instead of a Wrox-owned web site, I was confronted with a search engine. Undeterred I tried the web site on the back of the book, `www.wroxbase.com`. Not even found. Finally, I went to `www.wrox.com` but there was no mention of the book anywhere. After 10 minutes of searching, I finally found a little FAQ that explained that Wrox had recently gone bankrupt and sold a load of titles to Apress (`www.apress.com`), which is where I finally found the source code. Given the high quality of the book this was extremely frustrating, but obviously, the book was printed before the company's troubles.

[However the actual Wrox imprint was sold to Wiley & Sons. Francis]

Java Collections by John Zukowski (1 893115 92 5), APress, 415pp @ $49.95/£35-50
reviewed by Christer Lofving

This is yet one more of these easy-at-hand titles. Excellent to have behind you on the

desktop when working. But to read them through becomes tiring after a while. I started to read my copy with some expectations though, because the cover promise a "Comprehensive coverage of the Java Collections Framework", and "Real world examples, no toy code".

My enthusiasm also remained after the starting chapter about arrays. I learned some odd but interesting facts about this "primitive" collection and often forgotten area of Java programming. The first part of the book is dedicated to the so called "Historical" Collection Classes; Vector, Hash Table and Bit Set classes sort under this label, as well as the Enumeration interface. Later years updates of the Collection classes seems to be well covered. For example, the Bit Set class is not final anymore.

Core of the book is the coverage of Java Collection API. After a brief introduction and some pages about the newer Iteration interface which is meant to replace Enumeration, reading now becomes a little boring. Everything is still well explained, but the style starts to feel more like programmer's documentation.

What about "No toy code" then? Well, in my opinion there is still a lot of toy code. Maybe the code listings presented in the "advanced" ending part (describing COLT) are more professional and useful.

The book gives an interesting and reliable insight in Java Collections, but unless you are particularly interested in the subject or work with very advanced collections, you do not really need it. The Java API documentation gives enough information with good code examples to solve the main part of your Java Collection problems.

Mono – a Developer's Handbook by Edd Dumbill & Neil Borstein (0 596 00792 2), O'Reilly, 302pp @ $24.95/£17-50
reviewed by Paul F. Johnson

If you are new to Mono then you need to buy this book. It covers GtkSharp (the Gtk C# bindings), Monodevelop (a rather snazzy IDE for Mono), Webservices (you can now deploy ASP on a non-Windows platform) and everything else Mono has.

The writing style is clear and concise with plenty of code examples all of which will compile and run. The examples are well explained and as the book is logically set out, helping those wanting to develop under Mono to get going.

What the book does not teach is C#, which is fine and is best left to other books (see the ACCU website for an array of them).

My only bind with the book is that in an attempt to make the book look like a textbook, the pages are made to look like a schoolbook with feint blue squares on every page. It is not that annoying, but when you are trying to find something at 1am...

This is a very new book and replaces the SAMS book "*Mono Kick Start*" very effectively (okay, it is not a SAMS book, but it covers all of the parts not in the Mono Kick Start book). Highly Recommended

Mono Kick Start by Schonig & Geschwinde (0 672 32579 9), SAMS, 400pp @ $34.99/£25-50
reviewed by Paul F. Johnson

With the exception of the chapters on Qt# and GTK# and the simpler reading style, there really is not anything to recommend this book over any other beginners C# book.

This book really does suffer due to it's age – and it's not that old which on one hand is quite worrying, but on the other does say something about the speed of development of Novell's Mono package.

Even the GTK# code has large problems in that some of it does not compile and some is very much out of date. There are no updates on the SAMS website either to correct the mistakes which makes this book even less use for the C# beginner. The Qt# material is easily missed as there really is not very much of it.

This is all quite a pity as the book itself is very easy to read, but in itself, that is not enough for the price. Not Recommended

Other Languages

Perl Template Toolkit by Darren Chamberlain, et al. (0-596-00476-1), O'Reilly, 575pp @ $39.95/£28-50
reviewed by Jon Wilks

Template Toolkit (TT2) is a template processing system typically used for web site creation. The input data could be anything from variables specified at run time, an XML file or a database accessed via DBI for example. The templates could be structured to produce HTML but could just as easily be used to create XML, PDF or conceivably even source code - any application where there is the requirement to separate data and presentation. The book itself was written in Perl's pod system and processed using TT2.

This first edition of the book is based on version 2.1 of TT2. Its chapters describe TT2 in detail and go through, in tutorial fashion, the construction of a web site. The beauty of TT2 is that knowledge of Perl is not actually required to use this tool and the template language itself could be embedded easily by non-technical personnel (for form layout for example). Optional scripts are supplied with the Perl module that will process a single page or an entire tree of templates. Their use is covered well in the book.

The 12 chapters and 1 appendix cover all aspects of this tool from the syntax and directives up to internals and extension. TT2 is not the only template kit around and the "getting started" chapter offers a comparison of the other template systems available. All the code examples in the book are available from the O'Reilly website. After reading the first two chapters the rest of the book is written in a style that can be easily browsed as required. The reader is initially led gently from one concept to the next with later chapters offering a description of the template language, template directives, filters and plugins. Over two chapters, the anatomy of the system is described and information describing how to further extend TT2 is given.

I found the very easy to follow and in fact I have been using the template toolkit extensively for creating Unix system recovery documentation in a format independent manner, creating html, rtf and man pages from a single source tree. The template toolkit has made this simple and for that reason I highly recommend this book.

Learning Python 2nd ed. by Mark Lutz and David Ascher (3-596-00281-5), O'Reilly, 592pp @ $34.95/£24-95
reviewed by Ivan Uemlianin

The book uses the traditional bottom-up approach. After an opening part motivating the language and introducing the interpreter we progress, from data types, through statements, and up through functions, modules, classes and exceptions. Classes get 100 pages; other parts get about 50 pages each. A closing part covers common tasks, advanced uses, and Python resources. Appendices give details on installation and configuration, and provide solutions to all the exercises.

The book is thorough and patient. Topics are discussed in detail and at a steady pace. Repetition is used more than cross-reference. This book would be very good for self-study, as there is plenty of room for the plodder or the dipper. The exercises are worthwhile and to the point, and the solutions are explanatory.

Documentation and design issues are addressed early and often, and are clearly a central part of what is being taught in this book. The example code is of the highest quality.

The book's faults are minor. Although the preface says PyUnit and doctest are in Chapter 11; they are not, being given just a paragraph each in the core language summary in Chapter 26.

The book is not comprehensive (not a fault in itself), and a small number of language features are deemed 'too advanced' to be covered in depth, among them generators and the 'new style' classes. These features are sketched and given use cases, and the interested reader is directed to the documentation. Other features – like the useful little enumerate(object) – are not mentioned, but you have to draw the line somewhere. At 591 pages, the book is already large, but not unwieldy.

This book is a good example of Python culture, in the clarity of its text as much as in the quality of its code. Anyone working their way through it will have a solid foundation upon which to explore Python's potential. Highly recommended.

Patterns

Design Patterns in C# by Steven John Metsker (0-321-12697-1), Addison-Wesley*, 455pp @ $49.99/£37-99
reviewed by Paul Grenyer

The index of the book lists all 23 of the original GoF design patterns. A number of them are described in great depth and some even delve into examples of their use in the .Net framework and a description of what might be found when searching for the pattern name in MSDN. Some patterns also have examples of different methods of implementation. For example, the Adapter pattern describes a solution involving sub-classing and another using composition.

The description of Façade is particularly good and has a well thought out example that most people, and especially people who have written database clients, can relate too.

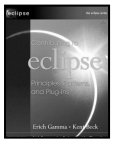
Some patterns such as Composite and Flyweight get the idea behind the pattern across but fail to provide a real world example. Singleton has quite a brief description and covers issues that arise from using singletons in threaded

environments. However, there is no mention of the controversy over the use of Singleton, as there is for Visitor.

The author has left a number of diagrams and code fragments incomplete as exercises for the reader, with the complete diagrams and code in the Solutions appendix. I found this incredibly irritating and frustrating as I was trying to relate the text to the supplied figure. It also makes this book more difficult to use as a reference. Protected data is used throughout the examples even in classes that would most likely remain the most derived (leaf) class.

There were a lot of things I did not like about this book, but most of that was style and a bit of bad practice in the code. However, there was a lot more I liked about it. Comparing with GoF, for the most part, the patterns are explained more clearly in this book and in greater depth and with better examples. If it is the patterns that you are interested in learning about then I rate this book over GoF. I think you should still read *Design Patterns Explained* by Alan Shalloway and James J. Trott first.

Tools



Contributing to Eclipse: Principles, Patterns and Plug-I by Erich Gamma & Kent Beck (0-321-20575-8), Addison-Wesley, 395pp @ $39.99/£30-99

reviewed by Silas Brown

Eclipse is an open-source editor/development environment that is extensible in a way that is reminiscent of EMACS, but based on Java rather than Lisp and the user interface looks more like Visual Studio. This book is aimed at developers who want to extend Eclipse, it assumes you have a working installation of Eclipse 2.x to play with.

The back cover says it's "comprehensive" but it's not. It is more of an introduction than a complete guide, because it aims to give you an idea of how to work and find more information by yourself. It does this by developing an example; there are many details specific to that example, but they may or may not relate to what you want to do. For a start, you will have a problem if you want to make tools for editing anything other than Java (yes I know Eclipse is implemented in Java, but the editing environment is supposed to be language-agnostic).

I do not like their apparently last-minute treatment of accessibility and internationalisation. There is a short chapter on it, along with an admission (on page 262) that the book's example code is inadequate in this respect. I can understand such corner cutting when things need to be kept simple, but this book's code is otherwise very detailed, so it could have set a better example. It looks like they were asked to address this issue shortly before the book went to press, rather than being aware of it from the outset. Even their admission fails to point out the full implications of conveying information using hard-coded red and green coloured rectangles, such as the effect on people with colour-blindness and those in cultures where red means happiness instead of danger. And yet about 30% of the book discusses the code to do it wrongly.

Overall, I think Chapter 1 is good, but how useful the rest will be depends very much on how well it happens to match what you want to do.

Eclipse 2 for Java Developers by Berthold Daum (0-470-86905-4), Wiley, 470pp @ $40.00/£32-50

reviewed by Rob Alexander

This book aims to give a broad overview of the Eclipse platform, and as such is divided into three sections. The first of these covers the use of Eclipse as a Java IDE, including the refactoring and code generation features. The second part describes the SWT and JFace toolkits, and the third details the creation of Eclipse plug-ins. It follows that the book in its entirety is useful only to those who want to create such plug-ins. For anyone using Eclipse purely as an IDE, only the first hundred pages are relevant.

The overview of the IDE is somewhat useful, although I do not expect that experienced users will refer to it frequently.

Some areas are touched on only briefly – the debugger receives a mere 8 pages. The coverage of the SWT and JFace toolkits is thorough. The brief comparison between AWT/Swing and SWT/JFace is concise and quite instructive.

Both the GUI toolkits and the plug-in system are illustrated by large examples (a MP3 player and a spell-checker, respectively). However, Eclipse version 3 is now available, and some of the changes cause problems. For example, the second example crashes because of a change in a core Eclipse library.

The text itself is far from sparkling. There are few actual language errors (oddly enough, the Introduction contains relatively many) but the writing is very drab and lacking in zest. The book is consequently tedious to read.

If your interest in Eclipse is only as an IDE, then this book will not be a worthwhile purchase. If you are interested in Eclipse plug-in development, or at least in SWT and JFace, then it may be of use. There are several similar books available, however, and I would suggest investigating those first. I can find little reason to condemn this book, but little to recommend it either.

MDA Distilled by Stephen J. Mellor et al. (0-201-78891-8), Addison-Wesley, 148pp @ $34.99/£30-99

reviewed by Nicola Musatti

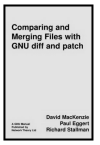
The Model Driven Architecture is an initiative by the OMG Consortium with the ambitious goal of replacing programming with design. This is to be achieved by providing ways to augment design diagrams with enough information to make it possible to automatically generate full applications.

This book is a short, easy to read description of the general principles on which the MDA is based. It only assumes that the reader is familiar with the UML and, possibly, with the relationship between models and metamodels (e.g. how the UML is specified in itself). In my opinion, it is more oriented towards analysts and team leaders, rather than programmers or managers.

The book's worst defect is the lack of concrete examples. The relative youth of the topic and the scarcity of implementations may explain this, but the difficulty in envisioning how the described techniques might work in practice makes the book less convincing than it might be. A great improvement would be the introduction of a detailed case study, so that each chapter could be completed by a few practical examples that showed how each of the MDA features might work in practice.

Another thing I did not like is the authors' apparently conceited attitude, which is better suited for a sales pitch than for a technical book.

Overall, I do not consider this a bad book, but I find it hard to identify a category of readers who might find it really useful. If you're interested in finding out what the Model Driven Architecture really is I think you should start by checking out the documents available from the OMG web site (`http://www.omg.org/mda`); then, if you still feel you would like to read a coherent overview, this book may be a reasonable choice.



Comparing and Merging Files by David MacKenzie et al. (0-9541617-5-0), Network Theory Ltd, 112pp @ $13.97/£12-95

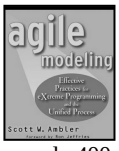
reviewed by Mathew Davies

This is a bound version of the manual that forms part of the GNU diff and patch package. The diff and patch tools provide you with a means of not only spotting differences between files but also distributing (source code) patches for your software. I have used diff a fair bit over the years and can vouch for it being a really useful tool.

Ten years ago, I might have considered buying this book; after all, it used to be time-consuming to load the manual pages into your favourite word processor and relatively expensive to run them off on your dot matrix printer, let alone binding the resulting document afterwards. Nowadays, I would not buy this book, given that I can download the manual in a selection of formats, including HTML, directly from the GNU web site. In fact, I have to admit to being rather baffled by the purpose of this book.

The back cover suggests that the publisher is donating $1 to GNU for each copy sold. Unless you particularly want a soft back, bound copy of the diff and patch manual, my advice would be the following: download the manual from the GNU web site, where you can be assured that it is completely up to date; then make a donation directly to GNU.

Methodologies



Agile Modeling by Scott Ambler (0-471-20282-7), John Wiley & Sons Ltd, 384pp @ $34.99/£22-96

reviewed by Anthony Williams

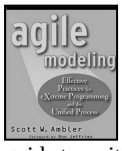
This is a reasonably long book, at nearly 400 pages; it would be even longer if it weren't for the excessively large number of words per page. I found this book hard to read, in part due to the layout, and in part due to Scott's writing style. Though he is a stout member of the Agile camp, Scott clearly also believes in the benefit of repetition to get his message across; there is many a repeated phrase or sentence, and there is at least one whole paragraph repeated word-for-word. All this detracts from the book, which is unfortunate since Scott has many good things to say.

The book is divided into 5 parts. The first two parts cover the principles you should work to and practices you should be doing to say that you are doing Agile Modelling as Scott defines it. The values of Agile Modelling are the four values of eXtreme Programming, plus a fifth (Humility), and the principles and practices are then derived

from applying these values to a modelling perspective. For example, the value of Courage leads one to Discard Temporary Models, and the values of Humility and Communication lead one to realise that Everyone Can Learn From Everyone Else, and that you should Model With Others. This description of the principles and practices forms the real meat of the book. Most (perhaps all) of what Scott says here is sensible advice that should be followed by anyone pursuing an Agile approach to software development.

The remaining parts describe the how Agile Modelling fits into XP and the Universal Process, with a discussion on introducing Agile Modelling into your process. This also includes a discussion of when Agile Modelling is not a good fit; a check list of things you must be doing to say you are Agile Modelling; and a list of things which you must not be doing if you want to say you are Agile Modelling. The book finishes off with an appendix listing a host of modelling techniques to consider when the need arises; Scott is quite clear that you need to Apply the Right Artefact, and having a wide range to choose from makes this easier since you are not stretching a model beyond what it can easily cover.

If you are interested in modelling, and want to know how it fits into Agile projects, or you are looking to make your current process more Agile by reducing unnecessary modelling work, then this book is well worth a read; I just wish it was easier to read. Recommended.



Agile Modeling by Scott Ambler (0-471-20282-7), John Wiley & Sons Ltd, 384pp @ $34.99/£22-96
reviewed by Jon Steven White

User Stories Applied is an excellent guide to writing User Stories and understanding how they can be best incorporated into the development lifecycle. The book is clearly written by an author who has not only an obvious wealth of experience in agile development, but also the ability to provide information to the reader in a simple effective manner.

In the first part of the book, the author provides a good overview of user stories, including detail on writing stories, gathering stories through user role modelling, writing stories when you do not have access to real end users, and testing user stories. Each chapter concludes with a clear summary, followed by an outline of exactly what the developer and customer are responsible for, clearing up any ambiguity.

The second part of the book covers estimating and planning, whilst the third part covers frequently discussed topics, including excellent chapters on bad user story application and using stories with Scrum. Again, these sections are very well written and offer both good explanation and practical advice.

The fourth part of User Stories Applied describes a comprehensive example, bringing together all of the earlier material. This works very well, giving the user extra confidence in the material, and a chance to revisit the concepts again.

Overall, I think that Mike Cohn has produced a great book in User Stories Applied, directly tackling an area that is often condensed and confused elsewhere. Requirements gathering is more important than ever today, and I would not hesitate in recommending this book because I am confident that the guidance it provides will help to produce better software.

Extreme Programming Adventures in C# by Ron Jeffries (0-7356-1949-2), Microsoft Press*, 518pp @ $39.99/£27-99
reviewed by Anthony Williams

I thoroughly enjoyed reading this book. It is neither a guide to XP, nor a tutorial for C#; rather it is a description of Ron's efforts to produce a working program that provides real customer value whilst learning a new language. The program in question is an XML notepad, with the aim of making it easier for Ron to edit his website, and Ron guides us through it in the humorous manner common to all his writing. Ron being Ron, the project is undertaken in an eXtreme Programming style, though the limitations of the book project mean that he has not employed all the practices "as written"; he is his own customer, and he doesn't always manage to find a pair, for example. As you follow Ron through the project, with the aid of the lessons he pulls out, you get a better understanding of the way he develops software, and are given an opportunity to judge how it compares to what you would have done. You might also learn a little C# along the way, as Ron explains each new language feature when he first uses it, though this is not the key focus of the book.

The project is not just one big success story; Ron shares his mistakes with us so that we may learn from them. The book is interspersed with "lessons", where Ron reflects on the preceding section and tries to identify important points, either things that he felt worked well, or the mistakes he made, and what he thinks could be done to try and avoid similar mistakes. Also throughout the book are sentences marked "sb", for "sound bite". These are short phrases which summarise a point, like "It's Chet's fault" (don't focus on finding who is at fault when things go wrong, rather focus on fixing the problem), or "You Aren't Gonna Need It" (focus on what needs doing now, rather than what you think you will need for later). It is these lessons and sound bites which provide the "message" of the book – Ron's belief that incremental development, done test-first, with simple design, continuous refactoring and a focus on producing value for the customer is an effective method of producing high quality software.

If you like Ron's other writing you will love this book. If you have never read Ron's work before and are interested in learning a little about how he applies the principles of XP (and maybe a little C#), it is worth reading; you might even enjoy it. Highly Recommended.

Games Programming

Andrew Rollings and Ernest Adams on Game Design by (1-5927-3001-9), Mew Riders, 617pp @ $49.99/£38-99
reviewed by Alan Lenton

I have to admit that I approached this book with more than a little cynicism. As a game designer I often get asked to comment on books about game design, and frankly most of them are crap. This time though I was pleasantly surprised – a well written book whose authors clearly know what they are talking about.

This is not a book for those who 'want to get into the business'; it is strictly about the art and science of game design. The book will appeal to those in the computer games business who are already games designers, or who are aiming to move sideways into games design. It will also appeal to anyone who just wants to find out how it is done.

The book is in two sections. The first part covers general issues that crop up in games design, while the second half of the book is a systematic look each of the different game genres.

What weaknesses there are tend to show up in this second half, which is a little uneven, since the authors don't (understandably) have first hand experience of each genre. There is also something of a tendency to try to jam all the genres into the same framework.

The weakest part of the book is the chapter on multi-player games where there is a failure to realise that game technical and social management tools have to be part of the design – they can't be bolted on afterwards – boring, I know, but important.

But these are not mega problems and they don't detract from the overall usefulness of the book. I've been designing persistent world multi-player games (and one commercial single player game) for nearly 20 years, and there was much in the book that I found helped articulate and systematise my experience.

Very useful indeed.

Linux Game Programming by Mark Collins et al. (0 7615 3255 2), Prima*, 330pp + CD @ $39.99/£29-99
reviewed by Paul F. Johnson

Where to start with this?

Code that is broken, but compiles; code that is broken and does not compile; insecure network advice and code; using libraries that will not work on quite a large number of Linuxes, and code samples for sound which do not work.

The average Linux distribution has just about everything a user needs; however, the one final area where there is completely inadequate number of packages is in the games domain. Without games, the appeal is diminished; which is a pity given the strength of Linux now.

Unfortunately, this book will not help. It is neither in depth enough or clear enough in how to write a game. There is nothing on game timing, the planning or other game essentials.

A number of the websites listed on the back of the book do not exist, neither does the website given inside the book for the network socket library. All of these diminish the value of the book.

To add insult to injury, the CD supplied is dire – it is written using the MS end of line, which means under Linux, there is no line wrapping.

This is a very poor book. Do not buy it. Not Recommended.

Programming Linux Games by Loki Software (1 886411 49 2), No Starch, 424pp @ $39.95/£29-99
reviewed by Paul F. Johnson

This is a very good book that has one downfall; it uses code snippets rather than proper

code examples. It feels more of a "proof of concept" book.

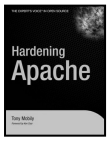
The other problem is that the book needs some minor updates – the main one being the coverage of the OpenAL code. While not a major problem (it's quite easy to fix the code by looking at just the header), having code that does not just compile out of the box can cause problems for those new into the games programming arena.

Most of the main aspects of game programming are covered – scripting, SDL, sound (compressed and uncompress – encouragingly, it demonstrates how to use vorbis files) and event handling. Game timing is the only part that perhaps is not covered quite as much as it should be. While keeping a baseline of update every 1/30th of a second will work, it is perhaps not the best way of ensuring everything keeps moving.

I did enjoy the games engine code and description. I have read many books dedicated to the subject and to be honest, this holds its own to them.

Had there been someway of getting hold of updates with the OpenAL code fixed, this would have gained a highly recommended. Unfortunately, it does not so, only achieves a recommended.

The Web



Hardening Apache by Tony Mobily (1-59059-378-2), Apress, 270pp @ $29.99/£18-50
reviewed by Richard Lee

The aim of this book is to provide a starting point for anyone needing to secure an Apache server. Each chapter deals with a different security issue before pointing the reader at a few good sites for further information. The author assumes a Unix-derived operating system but half of the book still remains useful to Windows.

The first chapter illustrates how this book differs from others. There is brief introduction to digital signatures and encryption before explaining how to verify the download has not been tampered with. Instead of just providing commands to install Apache, the author immediately delves into testing for problems including steps to remove vulnerabilities.

While the first chapter may follow a cookbook approach, the book aims to be more than just a simple set of recipes. It also explains how the server may get compromised, to look for suspicious behaviour in logs and web sites to visit to keep up-to-date with emerging security issues.

Given the responsibility of setting up an Apache based server, should you buy this book? It boils down to whether you can find all the information you need from the Internet or if you prefer a little helping hand along the way.

Web Development with Apache and Perl by Theo Petersen (1 930110 06 5), Manning, 410pp @ $44.95
reviewed by Joe McCool

Petersen makes a strong case for using Perl on Web development. Perl text handling capabilities are legendary. It is easy to learn. The richness of features derives from its maturity. It is widespread; most systems administrators have access to it and have some sort of notion on its workings.

He also makes a strong case for mod_perl, where the perl code can be built into the web servers directly. mod_perl is stable,. enjoys ongoing development and is well documented. Where Perl is already in house, mod_perl adds considerable leverage.

Considerable attention is given to the installation of mod_perl, where the conventional CPAN installation methods of Perl fall slightly short.

The whole of part 3, 110 pages are devoted to example web sites. These include a store front, office applications, systems administration, build your own portal and a little bit on credit card processing (hardly adequate).

Part 4, 80 pages, is devoted to site management, both content and performance. Here the discussion on development life cycles and phased testing is but a glance at a complex and dangerous subject. Most of us join teams with this already in place or learn precariously through practice.

Security is now a horrendous issue on the web and Petersen's treatment might not be sufficient (9 pages on user authentication and 3 on management).

My main reservation is that Petersen is so taken with the Open Source world that he is inclined to waste a lot of space preaching to the converted. The first few chapters are taken up with a discussion of the ubiquitous nature of Apache and its close cousins: perl, mysql, cgi etc. Most sites thinking of web server applications will already be up to speed on these. A lot of the material on the use of CPAN and installing MySql are already covered in other, less specialised, books.

Thankfully, he does not even pretend to offer an introduction to the Perl language itself. Familiarity with that is assumed.

Despite this, Petersen's book is useful and I am happy to recommend it. Most experienced readers can afford to skip the first few chapters. It is good value for money and well worth the shelf space, but it will probably need accompaniment with a few other texts to get readers up to speed.

Web Privacy with P3P by Lorrie Faith Cranor (0-596-00371-4), O'Reilly, 321pp @ $39.95/£28-50
reviewed by Tim Pushman

This book covers the P3P Project (the Platform for Privacy Preferences), from its inception and development through to a discussion of the current state of the proposal. Further chapters also provide an overview of related protocols and tools, such as APPEL. The author of the book is one of the co-authors of the specification and so has a good understanding of the issues involved in creating the standard.

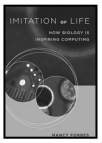
The book is arranged in three parts: background and history, enabling a web site, and software and tools. At the end are appendices covering some odds and ends.

P3P is a protocol to allow web sites to inform their users of what kind of privacy they can expect on the site, how their data will be collected and used, and what recourse the user has if she believes the data is being misused. In short, it is a Privacy Policy as one would find on a site such as Amazon, but with the added twist that it can be installed in a machine-readable format and directly interpreted by a P3P enabled browser. And the machine-readable format is, of course, XML. The second part of the book gives a detailed explanation of how to create a P3P policy, both by hand or by using a policy editor. There are many levels of complexity in a privacy policy and the author does a good job of explaining the various possibilities, from the simplest (we collect no data) through to the most complex, as would be needed by a large commercial organisation.

The question is: do people really care about their privacy online? Probably not as much as they should do. P3P is an attempt to make protecting our privacy as transparent as possible. We should be able to specify what information about ourselves we want to make available to a web site or organisation and then let the software take care of it for us. There are many places that software can be P3P enabled, browsers being an obvious example, but also web proxies, installation programs, registration programs and so on. Unfortunately there seems to be very little available in the real world.

When reviewing the book I had expected to find more on the code side, and was a bit disappointed to realise that the book covers only the protocol, albeit with a large chunk of XML. As far as discussing the P3P protocol goes, the book is excellent reading, if occasionally rather dry. The author clearly knows the technology and explains it clearly. Whether any of it matters is another thing entirely, but if you are in the business of P3P enabling your company's web site, then this book is recommended.

General Programming



Imitation of Life by Nancy Forbes (0-262-06241-0), MIT, 171pp @ $25.95/£16-95
reviewed by Francis Glassborow

The sub-title of this book is 'How Biology is Inspiring Computing'. I think that only tells half the story because by the time you have finished reading this book you will realise that computing is also inspiring biology.

This book is a comprehensive overview of the ways that biology and computing are interacting. Every one of the ten chapters provides food for thought. Some such as chapter 4 on artificial life also provide enough data so that you can find interesting work on the Web. I found a good place to start was `http://www.his.atr.jp/~ray/tierra/`. I won't spoil it for you by saying more than 'have a look at that site and follow the links'.

Chapter 8 is titled 'Computer Immune Systems' and covers some of the lessons from biology that can be applied to dealing with computer viruses and the like.

The author manages to focus on providing information rather than regurgitating the hype of enthusiasts for a specific area. For example she steers straight down the middle on the subject of DNA computing. She provides enough information to inform the reader as to what this is and what has so far been done but avoids the wild speculation of some popularisers.

The book is readable and short enough so that you will not need to spend you Christmas holidays reading it. If you want to think about where computing is going and some possibilities being currently explored in the laboratory this is a book worth taking the time to read.