Contents

Reports & Opinions

Dialogue

Features

Reviews

Copy Dates

C Vu 17.1: January 7th 2005

C Vu 17.2: March 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

I really do enjoy being part of the open source movement. It really does make the mind boggle at the speed of development of software. I've been tracking Novell's Mono application for quite a while now and since the 1.0 release (which was roughly 8 months ago), things have gone on in leaps and bounds. Probably the most interesting part has been the development of System.Windows.Forms.

In the original version, Mono relied on Winelib to provide the SWF parts. Unfortunately, Winelib and Wine moved all the time, which meant that rather than concentrating on SWF, the Novell team and those around the world which contribute to the source would be trying to hit a moving target. In the end, it was decided that there would be a ground up effort to implement SWF natively. That was roughly 4 months ago. It is (at the time of writing) somewhere close to being 88% complete. To anyone, that is a big achievement and probably, without the sort of community which is found in the open source movement, the completion would be closer to 60%.

Of course, I could be wrong, but I can only call what I see. It certainly has been a rollercoaster of a ride and has been a lot of fun.

The Student Code Critique

The SCC is a critical part of C Vu. It is one of the ways for you to all participate in the magazine. However, the numbers seem to have dropped and dropped to such a point that in this issue, there is one entrant. While it does make my judging for the book a lot easier, it is worrying the level of members who will take about 20 minutes to enter. Please get involved – not only do you make it more interesting, but you're also helping to educate others.

Ever Had One of Those Moments?

I'm sure that you've all had one of those moments, those gloriously inspired moments when you know *exactly* how to fix that piece of code which has been bugging you for days now and better than that, by making a couple of changes to said code, you can fix a number of other problems *plus* make it run faster and take less time to compile? It's wonderful isn't it.

I had one of those a while back when I was hand optimising some code within a program I help develop. By replacing a lot of inefficient code with something like

```
QString combo[] = { "Text", "Link", "External Link",
                    "External Web-Link" };
size_t comboArray = sizeof(combo)/sizeof(*combo);
for(uint prop = 0; prop < comboArray; ++prop)
  ComboBox1->insertItem(tr(combo[prop]));
```

it made the program a lot tighter and quicker. There was a problem with it which wasn't apparent under the test conditions (aka on my machine with different EU locales set). `tr()` is the Qt translator – it is a very powerful piece of kit, but unfortunately with it set inside of the `insertItem` method, the translator wasn't called. This did bite into the efficiency (as shown by various calls to memory and CPU profilers) and the second version was to replace

```
QString combo[] = { "Text", "Link", "External Link",
                    "External Web-Link" };
```

with

```
QString combo[] = { tr("Text"), tr("Link"),
     tr("External Link"), tr("External Web-Link") };
```

which oddly enough worked. At first, I thought the problem was in the `comboArray` line, but then that really didn't make sense – all that line does is give the size of the created array. That could only mean that the `insertItem` method couldn't take the `tr` conversion step. Swine! I had altered a substantial amount of code to use my original method and then find it doesn't work as well as anticipated (though it did work).

This did lead me to suspect that perhaps my testing and programming methodology was incorrect. Up to now, I had been a single programmer, working on a project which really, not many people would use and if they did, well, the problems would not be that huge to work around. In other words, I'm not being lazy per se, just not being as considerate as perhaps I should; software, after all, is a global commodity.

What I finally concluded was that I should not have made so many changes, had one and sent that out as a simple test case and worked on the results of that. Okay, grepping through the code wasn't that big a task, neither were the changes, but it was time I could have spent doing things I enjoy – like having a relaxing pint of some foaming nut brown ale and reading my collection of Dr Who books (hey, even I have to rest sometimes!)

Leading on from that, I decided to do some more code – this time, replacing normal code with template code to try and speed things up – if not from the user's point of view, then definitely from the system's point of view. This time, I started small...

```
/* While this version is simpler to read and the
   final binary is around 4k smaller than the
   template version, gcc 3.4 with a few
   optimisation tools being run show this to be
   slightly less efficient in terms of processor
   time. */

#include <qapplication.h>
#include <qslider.h>
#include <qlcdnumber.h>

int main(int argc, char* argv[]) {
  QApplication myapp(argc, argv);

  QWidget* mywidget = new QWidget();
  mywidget->setGeometry(400, 300, 170, 110);

  QSlider* myslider = new QSlider(0, 9, 1, 1,
                    QSlider::Horizontal, mywidget);
  myslider->setGeometry(10, 10, 150, 30);

  QLCDNumber* mylcdnum = new QLCDNumber(1, mywidget);
  mylcdnum->setGeometry(60, 50, 50, 50);
  mylcdnum->display(1); // display initial value

  // connect slider and number display
  QObject::connect(myslider, SIGNAL(sliderMoved(int)),
                mylcdnum, SLOT(display(int)));

  myapp.setMainWidget(mywidget);
  mywidget->show();
  return myapp.exec();
}
```

Okay, not exactly rocket science in terms of code (*and as you've all been reading the Qt series over the past couple of issues, you can tell me what it does*). However, some of the methods are very similar and how they work even more so.

Now, I could have written a simple wrapper, but instead came up with this

```
// Qslider v2 – template version.
// qslider-template.cpp

#include <qapplication.h>
#include <qslider.h>
#include <qlcdnumber.h>
```

```
#include "memory.h"

template <typename N, typename T>
void setGeometry(N m, T *x) {
  m->setGeometry(x[0], x[1], x[2], x[3]);
}

int main(int argc, char* argv[]) {
  QApplication myapp(argc, argv);
  QWidget *mywidget(allocate_memory<QWidget>());
  testAlloc(mywidget);

  int geom[4];
  geom[0] = 400; geom[1] = 300; geom[2] = 170;
  geom[3] = 100;
  setGeometry(mywidget, geom);

  geom[0] = 0; geom[1] = 9;
  geom[2] = geom[3] = 1;
  QSlider *myslider(allocate_memory<QSlider>(geom,
                  QSlider::Horizontal, mywidget));
  testAlloc(myslider);

  geom[0] = geom[1] = 10; geom[2] = 150;
  geom[3] = 30;
  setGeometry(myslider, geom);

  QLCDNumber *mylcdnum(
        allocate_memory<QLCDNumber>(1, mywidget));
  testAlloc(mylcdnum);

  geom[0] = 60; geom[1] = geom[2] = geom[3] = 50;
  setGeometry(mylcdnum, geom);

  mylcdnum->display(1); // display initial value

  QObject::connect(myslider, SIGNAL(sliderMoved(int)),
                mylcdnum, SLOT(display(int)));

  myapp.setMainWidget(mywidget);
  mywidget->show();
  return myapp.exec();
}

// memory.cpp
#include <qapplication.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <new>
#include <cstdlib>
using std::nothrow;

template <typename N>
N *allocate_memory() {
  return new(std::nothrow) N;
}
template <typename N, typename M>
N *allocate_memory(int val, M *&m) {
  return new(std::nothrow) N(val, m);
}

template <typename N, typename M, typename O,
          typename T>
N *allocate_memory(T *t, M m, O o) {
  return new(std::nothrow) N(t[0], t[1], t[2], t[3],
                             m , o);
}

template <typename N>
void testAlloc(N &w) {
  if(!w)
    exit(EXIT_FAILURE);
}
```

What is the advantage over the original version? Well, for a start in `memory.cpp` I have a very simple, yet very effective memory handling routine – given it was only a test bed, it is probably not win any prizes for the best and tightest code around, but the important thing was for what I threw at it, the code worked and worked well (the profilers I use showed roughly a 10% improvement over the original).

Was there really a point to the exercise though? The code never did make the release version after all. Yes. Yes there was. It is a proof of concept that demonstrates that it is entirely possible not only make the code tighter and more importantly, more secure (there is a planned network of the application so everything has to be as secure as possible).

Sad Times

Unfortunately, I have to report that our production editor of many moons has decided to very reluctantly move on to pastures new. Pippa has been possibly one of the best production editors I've had the pleasure to work with. Not only has she been patient, but has that rare quality of knowing about the subject matter in hand.

We will all miss her and wish her well. C Vu and Overload are now after a new production editor. If you would like further details, please contact John Merrells (`publications@accu.org`).

And So...

Well, this is the final edition of C Vu for 2004. All that remains for me to say is that from all of the ACCU Committee and C Vu + Overload production staff, may we all wish you a warm and merry yuletide and that 2005 be a fantastic year for you all. See you in 2005!

Paul F. Johnson

View From the Chair

Ewan Milne <chair@accu.org>

As we approach the end of 2004, it is time to look back on a past year and recognise just a few of the great contributions made by C Vu and Overload authors, as we announce the winners of the 2003 ACCU Authors' Competition. Err, 2003? Yes, I'm afraid that over-optimistic planning and project overruns can affect the best of us, so that the intended announcement of these winners at this year's conference has obviously been missed by, well, some months. Announcing a release date before having a plan in place – who would ever think of doing that?

But a dedicated team of judges have now spent several autumn evenings poring over every article published in C Vu and Overload in 2003, and we have finally reached the end of our deliberations. To say that the judging was made difficult by the universally high quality of articles that are published in both magazines might sound like an awards ceremony cliché, but it is true nonetheless. We have mentioned some of the articles that just missed the top spots, but in truth there were many other strong contenders. The committee would like to thank everyone who has contributed to the magazines. So without further ado, the winners are...

Best C Vu Article

An Introduction to Optimising Programs by Roger Orr

Honourable mentions: 10 Things You Always Wanted To Know About Assert (But Were Afraid To Ask) by Garry Lancaster, the Professionalism in Programming series by Pete Goodliffe

Best C Vu Article by a New Author

When Worlds Collide #1 - Embedded Systems and General Purpose Computers by Mark Easterbrook

Honourable mentions: Brackets Off! by Thomas Guest, Maintaining Context for Exceptions by Rob Hughes

Best Overload Article

A Return Type That Doesn't Like Being Ignored by Jon Jagger

Honourable mentions: Singleton - the Anti-Pattern by Mark Radford, Ruminations on Knowledge in Software Development by Allan Kelly

Best Overload Article by a New Author

Choosing Template Parameters by Raoul Gough

Honourable mentions: Stream-Based Parsing in C++ by Frank Antonsen, `EXPR_TYPE` - An Implementation of `typeof` Using Current Standard C++ by Anthony Williams, Exported Templates by Jean-Marc Bourget, Labouring: An Analogy by Seb Rose

[reports concluded at foot of next page]

Dialogue

Student Code Critique Competition 31

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

Have you ever come across a tricky bug at work that took you the whole day to find, or an exercise at school that didn't work the way you expected to? Those could be good opportunities not only to share it with other members, but to receive feedback from them. After all, this belongs to the Dialogue section, so who better than you to take part?

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 30 Entry

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);
```

[continued from previous page]

Membership Report

David Hodge <membership@accu.org>

With the main renewal period over the membership stands at about 895.

From my point of view the rolling membership system is going well. The joining envelope now contains just one issue instead of the back issues for the whole year, so is easier to manage. A new member usually gets their journals, handbook and welcome letter posted within 48 hours of joining. If you have a UK bank account and would like to save £5.00 on your next year's subscription by paying by standing order, just email me.

Standards Report

Lois Goldthwaite
<lois@loisgoldthwaite.com>

One of the discussions at the meeting of the C++ standard committee in Redmond, Washington, in October could have a big effect on the future of C++ as one of the world's most important programming languages. This was a presentation by Andrei Alexandrescu, who has been a popular speaker at several ACCU conferences.

In a paper with several co-authors (the UK's Kevlin Henney being one of them), Andrei is proposing that C++ define a memory model which will serve as a reliable basis for multithreaded programming. The C++ standard is written in terms of the operations of an abstract machine. A conforming implementation need not copy or emulate the structure of this abstract machine, so long as it produces the same observable behaviour resulting from a well-formed program with proper inputs. The observable behaviour of the abstract machine is "its sequence of reads and writes to volatile data and calls to library I/O functions."

As regards non-volatile data, compilers are free to reorder reads and writes as much as they like, so long as conforming observable behaviour is the result. Apart from compiler optimisations, processors and operating systems may contain their own logic which affects when changes to memory locations become visible.

Many multi-threading libraries are explicitly or implicitly based on the idea of a single main thread of control which assigns time-slices to different execution contexts. As multi-processor and multi-core computers become more common - and there are desktop systems right now with two or more processors - truly parallel executing threads will become the norm. What will it take to ensure that two threads looking at the same memory location will definitely see the same value there?

The paper, which you can find at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf`, attempts to set out the rationale for why the C++ committee should undertake the effort to define a more robust memory model for the future, clearly specifying the interactions between threads and memory. Future tasks, outlined by Andrei in his presentation to the committee, are to define a small set of primitive operations necessary to support multi-threading, on top of which a threading library can be built.

The main questions addressed by a memory model include:

- Atomicity: Which memory operations have indivisible effects?
- Visibility: Under what conditions will the effects of a write action by one thread be seen by a read by another thread?
- Ordering: Under what conditions are sequences of memory operations by one or more threads guaranteed to be visible in the same order by other threads?

These factors cannot be guaranteed by a threads library alone; they are fundamental to the meaning of any code as simple as `a=1; b=2;`. Some of the basic theoretical work has already been done during the rethink of the memory model for Java programming; the authors hope to reuse some of that work in the C++ project.

This effort is not expected to result in any changes to C++ syntax, but it may well change some of the underlying semantics – if only by tightening up the rules on how optimisation can alter the flow of control written into source code.

If you would like to participate in this important work, and especially if you have expertise in this area, please write to `standards@accu.org` for information on how to join the UK C++ panel.

Letter from America

Reg Charney
<charney@charneyday.com>

Here is the report on the U.S. chapter.

- Membership/attendance at the local Silicon Valley meeting has been fairly stable – 10-20 people at each meeting.
- Due to cost considerations, we have stopped printing the ACCent news letter. It did not seem to be generating enough interest/attendance.
- Most membership dues must be going through the ACCU main web site. I have had only one member renew by cheque in the last two months.
- I am trying to revamp our U.S. web site and am in negotiations with some people to give us the basis for some unique content. It will entail a major rework of our site.
- We have a healthy bank balance.

```
        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;
  }
```

From Neil Youngman <ny@youngman.org.uk>

This piece of code is a bit of a problem. On a first reading it's hard to tell what it's trying to achieve. I can see that there's some sort of date related functionality from the definition of variable `Day2`, but it really doesn't make it obvious what it's doing with those dates.

First things first, I guess, start with the compiler error and then try to deal with the other issues. This is at some level obvious, i.e. the `else` is mismatched, but which `if` statement does it go with?

It looks as if it probably matches

```
if((mtgrec.substr(3,1) == "0")
   || (mtgrec.substr(3,1) == "1"))
```

but with erratic indentation and without any clear idea of intent, that's not really certain.

To go much deeper I need some idea of meaning. The function is called `IS_IT_A_DDR`. DDR to me is a type of memory, which doesn't help. The parameter names don't offer much of a clue either. The first parameter is called `mtgrec`. I guess that would be 2 components `mtg` and `rec`. `mtg` could be mortgage or meeting and `rec` could be record. Given the date related details I would guess that meeting record is the most likely meaning.

The first `if` statement inside the `for` loop looks for a 3 letter day of the week at the start, the second for a '0' or a '1' and the 3rd for any of the characters in "BCLMOPSTW*" anywhere from the 8th character on. the value of `ddrrc` will be set according to which of these it finds. As I seem to be no closer to deducing the purpose of this code, so I guess I'd better consider the many stylistic abominations.

First off naming. As pointed out in the question the naming is poor. I have been unable to determine what the code is intended to achieve. There should of course be comments to assist with maintenance, but there aren't. Even with comments, clear use of names is invaluable in understanding the detail of a program. Here I have neither.

Looking at the function definition `ALL_CAPS` is a common stylistic convention to denote a macro or a constant. I can't see why it is used here. The choice of names we have already criticised. The first parameter seems to be read only and should therefore be const. The last parameter seems to be a return code, indicating the result of the function. It seems to me that this should be the function's return value instead of the function returning void.

Looking at the variables both `Day2` and `Daytoken` should both be constants and some sort of collection structure, e.g. an array or a vector, seems more appropriate than a string. This makes clear that they are a group of separate, if related values, not a single value. The variable `badday` seems entirely redundant, as I can't see that it's value can be anything other than 7 if the loop runs to completion. Of course that makes the last `else` clause entirely redundant.

Next we come to a size check. This is fairly straightforward, but involves a magic number "8". Generally hard coded constants should be declared somewhere central with a name, both to minimise the number of places where you they have to be changed, should a change be needed and to make the code more readable.

That brings us to the values assigned to the variable `ddrrc`. The 8 in the size check could be related to the code we see, which clearly expects at least 8 characters. The numbers going into `ddrrc` carry no meaning whatsoever. These should definitely be defined as constants somewhere. I would probably declare an enumeration and make the function return a value of the enumeration type.

There are also some efficiency concerns. The first 3 characters of `mtgrec` are potentially extracted up to 7 times as we iterate through the loop. The obvious solution to this is to extract them just once, with a statement like

```
const string mtgDay = mtgrec.substr(0, 3);
```

but I suspect that this would be missing an opportunity to improve the design further, by introducing a proper structure to be used in place of a string. The string appears to be a collection of structured data and using a string for the data hides that structure. Defining a proper class (or struct) for that data would bring that structure out as well as being more efficient than using `string::substr()` to extract the components.

The Winner of SCC 30

The editor's choice is:

Neil Youngman

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

Francis' Commentary

My first reaction to the student's question is 'Are you surprised that the code has an error?' I would rapidly follow it up with 'If the corrected code passes the compiler, would you trust it?' I think that the only acceptable answer to both these questions is 'no'.

My next question is 'What should you do about it?' I would try to guide the student into 'Redesign the code and factor the separate actions into their own informatively named functions.' If performance becomes an issue after doing that, it is time to consider helping the compiler with the `inline` keyword.

Most of the reorganisation I want to do is concerned with the implementation so I will move that out into the unnamed namespace.

Before I do any of that, I take strong objection to both the function name (spelt in all uppercase) and the function return type. Any function whose name asks a question should return a `bool`. However it seems to me that the function does not answer a simple binary question but asks something else for which there answer is a choice of five things. I have no idea what DDR means in this context (I am pretty sure it does not refer to 'Dance Dance Revolution', 'Developers Diversified Reality' nor to some form of SDRAM), nor what the classification stored in an out-parameter (`ddrrc`) means so I will have to use some placeholders. These placeholders should be replaced with meaningful names. `enum`s are designed to deal with this kind of issue. As the enumeration constants need to be available in multiple translation units, it needs to be declared in a header file. Let me start with that:

```
#ifndef DDR_DECLARATIONS_H
#define DDR_DECLARATIONS_H
#include <string>

enum ddr_classification{
  ok, too_short, bad_day, bad_flag,
  bad_symbol, err5
};
ddr_classification classify(
      std::string const & mtgrec,
      std::string & outrec);
inline
void IS_IT_A_DDR(std::string const & mtrec,
                 std::string & temprec,
                 int & ddrrc) {
  ddrrc = classify(mtrec, temprec)+1;
}
#endif
```

I have provided a simple forwarding function to provide temporary support for the ill-named function so that no immediate changes need to be made to code elsewhere. I would expect that to be rapidly replaced. I have made `ok` take a zero value so that in future it will be possible to use the return value of `classify()` for a rapid check of validity. Note that I use fully qualified names in the header file and that I have added a `const` qualifier to the first parameter.

There is also the question of those two string variables; they aren't variable nor are the local (though they could be static). Such items belong in the associated unnamed namespace.

Here is the start for the implementation file:

```
#include "ddr_declarations.h"
using std::string;
namespace {
  string const daynames[] = {"Sun", "Mon",
                             "Tue", "Wed", "Thu",
                             "Fri", "Sat"};
  char * const daynumber = "0123456";
  char * const symbols = "BCLMOPSTW*";
}
```

I would favour a better name for symbols but without knowing the context that is the best I can do. Now let me try to write a halfway sensible definition for `classify()`.

```
ddr_classification classify(
      std::string const & mtgrec,
      std::string & outrec){
  if(mtgrec.size() < 8) return too_short;
  int daynum(dayname_to_int(mtgetrec));
  if(daynum > 6) return bad_day;
  if(not valid_flag(mtgrec.substr(3, 1)))
           return bad_flag;
  if(valid_symbol(mtgrec.substr(7, 1))){
    outrec = daynumber[daynum]
                  + mtgrec.substr(1);
    return ok;
  }
  else return bad_symbol;
}
```

Now notice that if the original code's `for`-loop ever exited that the student's `badday` variable must be equal to 7. There just isn't any other way through that nest of `if`'s. That was far from obvious in the original. Separating out the various conditions and only continuing if everything is still checking out leads to much clearer code. It also much better models the way would handle the problem ourselves. First check that the first three characters represent an abbreviation for a day, next check that the fourth symbol is acceptable then check that the eighth one is OK. Human beings might only notice that there were too few symbols at that last stage, though it is easier to check it first from a program perspective.

Now let me go back and add the requisite helper functions (which will go in the unnamed namespace).

```
int dayname_to_int(string const & mtgetrec){
  for(int i(0); i != 7; ++i){
    if(mtgrec.substr(0, 3) == daynames[i]{
      return i;
    }
  }
  return 7;
}
```

Yes, I know there is a magic number lurking in there, but I am running short of time if David is to get this in time to use it.

```
bool valid_flag(char flag){
  if(flag == '0') return true;
  if(flag == '1') return true;
  return false;
}
```

```
bool valid_symbol(char symbol){
  for(int i(0); i != strlen(symbols); ++i){
    if(symbol == symbols[i])
      return true;
  }
  return false;
}
```

Now I think that this code represents the intention of that provided by the student. It would have been much easier had the student specified what the code was intended to do.

Notice that the coding error was a direct consequence of an inappropriate view of how to code the problem. The student was willing to use all kinds of tools he had found in the Standard C++ Library but the tool he really needed was his own brain. Sadly once the code was working too many instructors would accept it.

Remember that this is part of the Dialogue section of C Vu so you have an implicit invitation to critique my solution as well as add any other useful information you have about the actual problem.

Student Code Critique 31

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

Here is the code I have using the equation to drop the lowest number from the grades. The problem is, if I change up number 3 and number 4, I get a different answer. I used the numbers 80, 84, 60, 100 and 90. Putting the numbers in like that, I get 88 but, if I mix up the 100 and 60 then I get a grade of 81. Can anyone tell me why it is not finding the lowest number and just dropping it when I tell it to (- lowest)?

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
int test1, test2, test3, test4,test5,average,
                            averagescore,divide;

cout <<"This program will gather five test
                                  scores and\n";
cout <<"drop the lowest score, giving you the
                                     average\n";
cout <<"\n";
cout <<"Please enter Five Test scores\n";
cin >> test1>>test2>>test3>>test4>>test5;

int lowest = test1;
       // test 1 is the lowest number unless
if (test2 < test1)lowest = test2;
       // test 2 is smaller than test 1 unless
if (test3 < test2)lowest = test3;
       // test 3 is smaller than test 2 unless
if (test4 < test3)lowest = test4;
       // test 4 is smaller than test 3 unless
if (test5 < test4)lowest = test5;
       // test 5 is smaller than test 4.

average = (test1+test2+test3+test4+test5);
       // all test scores averaged together
averagescore = average - lowest;
       // average score minus the lowest grade
divide = averagescore /4;
       // average grade is then divided by 4
cout << divide<< endl;
       // final grade after division
return 0;
}
```

Besides the question asked by the student, this code gives you a chance to discuss topics such as extensibility, design and style. Please address as many issues as you consider necessary, without omitting the answer to the original question.

Francis' Scribbles

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

Redmond Meetings

I spent the last two weeks of October in Redmond attending three Standards meetings; WG21 (C++), ECMA's TC3/TG5 (C++/CLI) and then WG14 (C).

WG21 is working very hard on developing the next release of the C++ Standard (scheduled for some time about 2008/9). There are a great many good ideas for improving or developing C++. Quite a number of these are originating in the UK.

The Most Important Paper

However, to my mind, the most important paper N1680 (available at `www.open-std.org/jtc1/sc22/wg21/docs/papers/`) was not a direct proposal but a discussion document. It is co-authored by A. Alexandrescu, H. Boehm, K. Henney, D. Lea, B. Pugh, and entitled *'Memory Model for Multithreaded C++'*. The paper makes it clear that providing proper support for efficient multi-threading is more than just adding a few libraries. It is not necessary to change the syntax of the language to provide correct underpinnings for multi-threading but some changes to the semantics are essential.

Trying to implement multi-threading on top of a single thread abstract machine simply cripples modern multi CPU, multi-core hardware. In fact even such primitive developments as Intel's hyper-threading has to be turned off for many programs to avoid complete chaos. It isn't just the assumptions that programmers make, but those that are made by the compilers.

If we had to design a suitable abstract machine from scratch we would have a task that we would almost certainly get wrong the first time (Java did, and its designers were neither naïve nor ignorant). However we can capitalise on the experience of others and we have suitable expertise within the C++ community. What I have some doubts about is whether we actually have the will. Many people seem to think that the solutions that are already available through such libraries as the Boost one are good enough. I do not think they are and I think that once you spend time understanding the above paper you will agree. However the argument that what we have is 'good enough' is a very seductive one because, if accepted, it allows us to go away and do those other simpler things that we understand and want in C++.

How urgent is it to provide a suitable memory model for multi-threaded C++? Well C++ will have many fruitful years left even if we do not change the abstract machine. But those years will be numbered. Already AMD has sampled a dual-core CPU (i.e. two independent processors on a single chip). Intel have announced that future versions of their Pentium line will be multi-core and Sun Microsystems are soon to release an 8-core processor. The upshot is that we are only a very few years away (perhaps as small as two) from entry-level machines having, effectively, multiple processors.

Note that this development has been clearly coming for several years though the present competition between AMD and Intel has probably brought it forward a couple of years. Every attempt at increasing performance by increasing CPU speed involves increased heat production. The current speeds are now pushing the limit of what can be handled by air-cooling. When chip designers start muttering about having to go to water-cooling you know we will have problems, particularly with low-end hardware.

The only viable option is to increase computing power by having more processors doing the work. From the hardware designer's perspective this is not a big problem. The problem comes from the software end. The big number-crunchers have been using multiple processors and big array processors for years. But getting effective use from such hardware has meant developing specific development tools for the hardware in an area where software tools are expected to cost orders of magnitude more than those we customarily use for desk-top software development.

Now simple mathematics comes into play. Place C++ head-to-head with a language such as Java on a single CPU machine and all the traditional arguments of compiled versus interpreted code, manually managed memory resources versus garbage collection come into play. In its own specialist areas each language will be able to capitalise on its strengths and make a case for continued use.

Move to a multi-processor environment and any language, which can be used to make efficient use of all the computational resources available, will outperform one that is tied to a more primitive model. A lock that stops all other threads is a potential disaster on a multi-processor machine. Suddenly half your power is lost on a primitive dual core machine, much more on the more advanced hardware that is on the horizon.

The upshot is that either C++ evolves to make good use of the coming hardware or it will lose out to newer languages. Such issues as support for garbage collection are of minor importance.

Now the interesting thing is that if we get the fundamental memory model right library designers will be able to provide efficient components for the application level programmer. The latter will have to learn a bit about multi-threading (actually. probably less than what you have to learn today) but will get good use of the available hardware. If we do not get the underlying model correct, no amount of excellent library design will compensate.

C++ Evolution

WG21 deliberately make all the development papers available for general reading because they hope that others will review them and add their insights and knowledge in order to improve them.

Just because a paper makes a proposal does not mean that it will be in the next version of C++. Indeed a great many good ideas will fall by the wayside. One certain way for an idea to fail is if it is not actively pursued. WG21 have more than enough to do without taking on work from others. If you see a proposal that you would like to see followed through, at the very least provide the authors with moral support and encouragement. They need to know that others think the effort worthwhile.

If you are in a position to do more such as contributing your knowledge and experience of prior art, or volunteering to try the idea out by modifying a compiler whose source code you have access to then please come forward.

One of the hidden costs in pursuing any proposal is the fact that other good proposals will die. We just do not have the resources to follow every good idea. And even if we did, doing so would not be good for C++. I know some people think that there should be a nice coherent development plan for C++. Great idea but it just does not happen that way. What gets done is what people are willing to do.

If you want to add a major feature to C++ like full support for functional programming (just to choose an area where there is some interest but no existing proposals so I will not be hurting anyone's feelings) you will have a much better chance for success by getting together a group of people who will do the work (and that includes coming to meetings). Sorry if you do not like that message but it is the practicality of the situation.

If You Want, You Must Do

An example of how that works is the 'Embedded C' TR that was produced by WG14. Those that wanted support for such things as DSPs got together, came to WG14 meetings and did the work.

Others have sat on the sidelines and criticised their doing the work for a whole range of reasons. Such criticism is unfair, inappropriate and, in my opinion, unprofessional. The group believed they needed support at least to the level of an ISO Technical Report, they put up the resources and no little cost to themselves and did the work. With more support and more expert eyes the result might have been better and more comprehensive, however I do not know that. What I do know is that those who simply tried to vote the work away on the basis that it should never have been done demean themselves and those they represent.

Shamefully, for reasons that I cannot explain here, the UK has been the worst offender over the last few years. Steps are being taken to correct the position but it would never have happened had the broad UK C community participated rather than sitting on the sidelines expecting others to represent them and their needs.

By contrast the UK C++ community has been one of the most active contributors to the future development of C++. We may not be able to send many people to WG21 meetings but our BSI Panel meetings are well attended and bubble with ideas and enthusiasm. Panel members have been putting in time and personal resources to help make C++ better. We do our best to ensure that C++ becomes better, and better meets the needs of its users. Most participants have a broad programming base and actively program in several languages.

ECMA v ISO

You will have noticed that one of the three meetings that I attended in Redmond was an ECMA meeting concerned with the development of what is called C++/CLI. In other words a 'dialect' (Herb Sutter likes to call it a set of bindings, but most of us think that is too simple a view) of C++ to be used with the ISO Standard to which .NET conforms.

I hear a good deal of criticism of the ECMA process. I agree with much of it but there is one thing of which I recently became conscious and that is that the radical difference in the natures of ECMA and ISO do legitimately lead to a different process.

ISO Committees are composed of National Body delegations. Those delegations are supposed to represent their national interests. Indirectly those will include a substantial element of their nation's commercial interests. However note that this is indirect. ECMA Committees are composed of direct commercial representation.

It is perfectly natural that a decision made in TG5 should be dominated by the commercial interests of the participating companies. Such things as currently planned shipping dates are important. The process will be designed to take much more direct consideration and input from the participating companies.

ISO committees naturally have a wider community to serve. Where it becomes interesting is when we have a situation such as that of WG21 that is colocated with a strong, technically competent NB Committee (J16). Like ECMA, J16 is largely based on corporate membership. Like ECMA, J16 members are focused on the needs of the companies they represent rather than a broader community. Often the tension between WG21 and J16 goes unnoticed but sometimes it pokes above the surface.

I am greatly in favour of technical work being done together without too much commercial influence but it would be a mistake to ignore that the latter exists and that in the case of both WG14 and WG21 the (overwhelming) majority of attending experts are actually from J11 and J16.

Sometimes areas (such as what new work should be done) are definitely the domain of the ISO Committee and not a National Body. Yet sometimes the way that proposals get discussed means that the dominant position of experts from a single country can distort the outcome.

A case in point is a proposal from a UK expert that WG21 should produce a TR on a Statistics Library. The UK had only two people at the Redmond meeting. I had my hands full with the work of the Evolution Group. Most attending NBs had only one or two representatives. The result was that when the WG21 Library workgroup considered the UK proposal for work on a Statistics Library the discussion was dominated (overwhelmingly) by the J16 experts. A number of strong voices had good commercial reasons for not wanting to add a Statistics Library even as a TR. Note that I am not criticising those J16 experts, they were doing their job; the job that their employers expect and pay them to do. It just is not the same job as that which should be done by a National delegation to WG21. Indeed the official US Delegation often has a markedly different as a national delegation to that of the members of J16 as members of J16.

I think we need to address the issues that this raises. But before we do, we need to be willing to provide (or acquire from other NBs) the resources to do the work. When it comes to the crunch, it is those who do the work who will determine what happens.

It is no use sitting on the sidelines and whingeing about things we do not like, we have to get in there and get involved.

While much of the above is from a UK perspective, much of it would be valid from the perspective of other National Bodies.

Problem 18

Some programmers seem to hate to use more names in their programs than they absolutely have to. Your challenge is to write a program in C++ that outputs the first `n` members of the Fibonacci series where the value of `n` is provided by user input at runtime.

That is easy for most readers. But there is a limitation, any variable, function, type or namespace that you declare must be called `i`. You are allowed to use anything you like from the Standard (such as `main`, `std::cin` and `std::cout`).

The requirement that it be written in C++ is because I do not think it can be done in C, and there maybe some other languages in which the problem is trivial.

In case you think it is impossible, I have a solution (which took me about ten minutes to develop – but I have the advantage of knowing the key to coding the problem) that is fifteen lines of code with not more than one statement per line.

Commentary on 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;
}
```

This is a version of an old problem, which is to write a program that outputs itself. In considering answers I am interested in more than just a program, I want to see the mental process by which the author arrived at it.

The first problem with the above code is ensuring that `(a * b)` compiles. There are two possibilities, either `a` is a type and `b` is a pointer to that type or `a` and `b` are both global objects with an `operator *` that can be called.

If `a` is a type then the output must be generated by some other code that will be run by executing the program. That basically requires that there is a global object with has a constructor or a destructor that somehow causes the output.

If `a` is not a type then both `a` and `b` must be global objects (because they must be declared somewhere.) There must also be a suitable `operator *`.

Once you recognise that the output must be generated by some code that runs outside `main()` you are left with a number of options. Here is one that I wrote earlier:

```
#include <iostream>
struct work {
  work() {
    std::cout << "int main(){\n"
              << "  a * b;\n"
              << "}\n";
  }
} x;
int a, b;
int main() {
  a * b;
}
```

The submissions I have had so far have gone for more complicated solutions. That does not make them bad but as a general rule good programs achieve their objective with a minimum of complexity.

I forgot to give a closing date so I am not going to select a winner until the end of November (which will be too late for inclusion in this issue of C Vu.) For obvious reasons, it will be too late to enter by the time you read this.

Cryptic Clues For Numbers

Last issue's clue seems to have provided rather a different problem to readers. Several of you wrote to me with the number itself but were struggling to produce an alternative clue. Here is the clue again:

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

The first three words give 040 (love is conventionally used in cryptic clues as 'o' (as a letter) or '0' (as a number) because of its use in such things as tennis scores. Now, in C that means 32 (octal 40). The second sentences provide confirmation because that is the value of the fifth bit in a byte (hey, we are C or C++ programmers and so count from zero).

Here is another clue to keep your brain cells working over the holiday season:

Deuce, it sounds like they came for tea twice. (4 digits)

There is plenty of potential for alternative clues. Something that happened on November 15, 1971 might be of use as clue material.

Francis Glassborow

Features

Blue Fountain Systems – An Open Source Company

Paul F. Johnson <editor@accu.org>

A company which produces Open Source Software (OSS) is nothing new. There are plenty in the US and Europe that have been trading very nicely for quite a number of years (RedHat and SuSE spring to mind). However, these are companies who write the distributions. Can a company exist which produces OSS, and if it can, how does it survive?

Recently, I was informed of a company based in Liverpool which does just that. As Liverpool is my home city and only about 45 minutes away, I arranged for a meeting and on the 3rd of November, in rather dull weather, ventured forth to visit them in the world famous India Buildings, very close to the River Mersey and a stone's throw away from the town centre.

I was met by a very friendly environment, with a team of six programmers and one receptionist. I was presently introduced to one of the directors, Aidan McGuire. Over a coffee and quite a lot of laughter, we settled down to conduct the interview for C Vu. Okay, he did find out a lot about my involvement with OSS, programming and technical background, C Vu and lots of other things first, but I wouldn't have expected very much else.

From past experience, interviewing company directors can be a tricky affair. Most don't (or can't) reveal very much. In true Open Source tradition though, Aidan offered more than I expected when he answered with a candidness which was really refreshing!

How long has Blue Fountain Systems been around and what is your primary business? How can an open source company make money? Is it funded on the back of writing bespoke code?

1991, though it became incorporated in 1996. Our primary business is as a solution company. A client stipulates what they want and not only do we write the software, but install the hardware, maintain the both software and hardware, provide training – in fact, everything you would expect.

As a business plan, we offer very reasonably priced maintenance contracts (ranging between £100 and £750 per month), as well as working with the likes of Southport Council to provide a free WiFi network for the town centre (it would be funded by a nominal fee from local businesses).

[*At this point, we chatted about the problem of older buildings and I used the example of the conference building we used this year – only to discover that Aidan had also been there and gave a talk in one of the Python sessions – we do move in a small world!*]

We are effectively using an IBM style "utility" model.

We do not work on the typical IPR model. It is something which surprises a lot of companies in the same line as us, as we freely and openly give away the source code, IPR is not a big issue and being an OSS company, we are transparent in that if we make a mistake, we can hold up our hands, admit to the mistake and fix the problem. OSS also gives us an advantage in that we can involve other developers and code without having to go through the expense on non-disclosure agreements. We do not write anything closed source.

We also work with our competitors. We are part of the Zope group and as such, while we may be in competition with other companies, we are working to the same common aim. This really confuses traditional companies! We are in competition, but we all meet quite regularly with a common aim.

You have described Liverpool as being "the open source capital of the UK". How did you come to that description?

That was from a PR company! That said, businesses and Liverpool Council are starting to come to see the advantages of using the Open Source model over the traditional way of purchasing software and licences.

Obviously, you have the two Universities a stroll from your offices as well as a good supply of talent from colleges. As they are brought up on the world of closed source (largely), how much of a culture shock is it for them to move to open source?

It is a culture shock and actually quite hard for them. However, they adapt quickly. We are working with Liverpool John Moores and the University of Liverpool on a mentoring programme. This means that students will see both sides of the coin.

Do you source most of your employees locally or do you advertise nationally and internationally?

Both – via the internet (we use JobCV as an agency). We also employ people on word of mouth as we find it is one of the more reliable ways of finding staff.

We are a company with offices in Belgium and a small office in London as well as possibly a new office opening in China due to their increased uptake of OSS. Additionally, we have people who work from home who are dotted around the UK and other countries.

What licence model do you use (GPL etc)?

LGPL

How do you feel when you have some of the big closed source producers denouncing open source as a flash in the pan and largely unsustainable (as has been recently seen in the technical press)?

I personally don't think that they understand the business model – and that equally applies to some Open Source companies

Development environment

Do you have an in-house development environment or have you settled on something like kdevelop / anjuta? Do you have a preferred development language?

We use Python / PostgreSQL / Zope for all development – unless there is a specific requirement from a client for another. Even if they do, we usually find that our trio of set technologies will accomplish the task and they are happy to use them when we demonstrate what they are capable of.

Does the company have a preferred widget set or is it a home grown one?

Most of our work is web based but for GUI projects we use wxWidgets (wxPython).

As lots of non-software companies expect to pay, pay and pay some more for commercial software, how, in your opinion, have they responded to the surge in OSS and the ability to get something they can tailor to their needs (or have you do) and still be able to see the source?

We are still fighting against scepticism and "early adopter" syndromes. Undoubtedly, the SMEs can see the big advantage in not only the total cost of ownership in using OSS, but there is still a lot of resistance given the relative newness of OSS and the domination of the big players in the commercial world.

If you combine that with the Liverpool being home to some very large companies, it is an uphill struggle, but one we are winning on.

Do you employ any form of extreme programming (or similar) and how effective do you feel it is?

We have examined many different development methodologies. I can't say we employ any specific one although our development methods do utilise methods of XP (e.g. rapid feedback, embrace change) and others. At various points we will try new ideas and embrace them if they work for us or throw them away if they don't.

What do you look for in a new employee? At the ACCU, we actively promote best practice when it comes to new employment as well as giving out a lot of advice on what to and what not to expect.

We look for quite a lot of qualities other than being a good programmer! As we have to go to the customer for their service contracts, the employee has to have not only customer relations skills but also be technically proficient to speak to them at the correct level.

Roughly, what proportion of local talent do you have to "shipped in" talent?

Currently, it's about 50/50. We do hope to be expanding soon and when we do, the local number will increase.

Future prospects

As you know, software is a fickle beast, though OSS has been increasing in adoption and use over the past years on not just the Linux

[concluded at foot of next page]

Writing Custom Widgets in Qt

Jasmin Blanchette

In the fourth installment of our series on cross-platform GUI programming with the Qt C++ toolkit, we are going to write a custom widget using Qt. The widget in question is a "scribble" widget (see Figure 1) – that is, the drawing area of a simple paint program. The user can draw by moving the mouse pointer while holding down the left mouse button.

Writing a custom widget using Qt isn't much different from writing an application's main window (C Vu Volume 16 No 3) or a dialog (C Vu Volume 16 No 4). It also involves deriving from a Qt base class, reimplementing some virtual functions, and connecting signals to slots. The main difference is that we also need to handle low-level events (also called "messages") such as paint events and mouse events to give the widget its look and feel.

The Scribble Class Definition

We'll start by looking at the definition of the Scribble class:

```
#ifndef SCRIBBLE_H
#define SCRIBBLE_H
#include <qimage.h>
#include <qwidget.h>

class Scribble : public QWidget {
public:
  Scribble(QWidget *parent = 0);
  QSize sizeHint() const;
  void setPixmap(const QPixmap &pixmap);
  QPixmap pixmap() const { return m_pixmap; }
  void setPenColor(const QColor &color);
  QColor penColor() const { return m_color; }
protected:
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  void paintEvent(QPaintEvent *event);
private:
  QColor m_color;
  QPixmap m_pixmap;
  QPoint m_prevPos;
};
#endif
```

The `Scribble` class inherits from `QWidget`, the base class for all widgets and windows. `Scribble` provides public access functions, three protected event handles, and some private variables.

The `m_color` data member holds the current pen colour. The `m_pixmap` member holds the image that the user is drawing. The `m_prevPos` member will be explained later; just ignore it for the moment.

The protected event handles are virtual functions inherited from `QWidget` that are called whenever the widget receives an event. Events are sent by the window system whenever some condition occurs. For example, if the user presses a key while the widget has the keyboard focus, the window system dispatches a "key press" event that the widget can handle by reimplementing `QWidget::keyPressEvent()`. The `Scribble` widget is interested in three kinds of event: "mouse press", "mouse move" and "paint" events.

The Scribble Class Implementation

We will now go through the implementation of the `Scribble` class, starting with the constructor:

```
Scribble::Scribble(QWidget *parent)
    : QWidget(parent) {
  m_color = black;
  m_pixmap.resize(480, 320);
  m_pixmap.fill(0xFFFFFF);
  setWFlags(WStaticContents);
}
```

The constructor takes a parent widget and passes it on to the base class constructor. If parent is a null pointer, the widget is a window in its own right; otherwise the widget is displayed within the parent's area.

In the constructor body we initialize the `m_color` and `m_pixmap` data members to default values. The pen colour is set to black; the pixmap is initialized to size 480 × 320 and filled with white (`0xFFFFFF`). Finally we set the `WStaticContents` flag on the widget, telling Qt that the widget's content doesn't scale when the widget is resized, but rather it stays rooted in the top-left corner. This simple trick lets Qt optimize drawing and reduce flicker drastically.

```
QSize Scribble::sizeHint() const {
  return m_pixmap.size();
}
```

The `sizeHint()` function is reimplemented from `QWidget`. It should return the ideal size of a widget. Layout managers take this into account when assigning screen positions to widgets. Here we return the size of the pixmap (480 × 320 by default) as the ideal size for the widget.

```
void Scribble::setPixmap(const QPixmap
                                    &pixmap) {
  m_pixmap = pixmap;
  update();
  updateGeometry();
}
```

The `setPixmap()` function sets the pixmap which the user can draw on. Notice that we call `update()` and `updateGeometry()` in addition to assigning the new pixmap to `m_pixmap`. The call to `update()` tells Qt to repaint the widget, ensuring that the new pixmap is shown straight away. The call to `updateGeometry()` tells the layout manager responsible for this widget (if any) that the `sizeHint()` might have changed.

[continued from previous page]

(and other free OS) platform, but also the Windows platform. Can you see this continuing for (say) the next 10 years and what effect will it have on company business plans?

Open Source is increasing at an almost exponential rate and should assure us of a good future.

Our main concern is the speed at which UK Plc is adopting open source. If we compare the UK to our European neighbours, we see them moving over and adopting Open Source more and more. The commercial edge is being lost to companies with far lower overheads due to their adoption of OSS. If we decide to go with the proprietary system and everyone else doesn't, then UK Plc is not going to be very healthy and it will probably take ages for us to claw our way back.

Hardware is forever changing with the push currently for movement to x64/IA64 and above. What have you got in place currently to ensure current products will still work in (say) 3 – 5 years from now?

All our work is done within Python so we are shielded from the joys of such things.

Have you seen any significant turn down or reluctance to using OSS since SCO's unfortunate FUD over their IP in Linux and subsequent suits against RedHat, Autozone, IBM and Novell? And where do you see that ending up?

We haven't seen a down turn and really, it is up to SCO to prove their claim. Even if SCO win, it may slow things down, but certainly won't stop it. Open Source is here and it's here to stay

I must thank both Aidan for being so friendly and open with his answers and Ian Cottee (the company's technical manager) for the more technical answers to some of my questions.

As you can see, it is not only possible for an OSS company to exist, but in this case, it is a company which is expanding and succeeding despite the reluctance of some to accept the change.

Blue Fountain Systems can be contacted on 0870 0202 111, `http://www.bluefountain.com` or `info@bluefountain.com`

Paul F Johnson

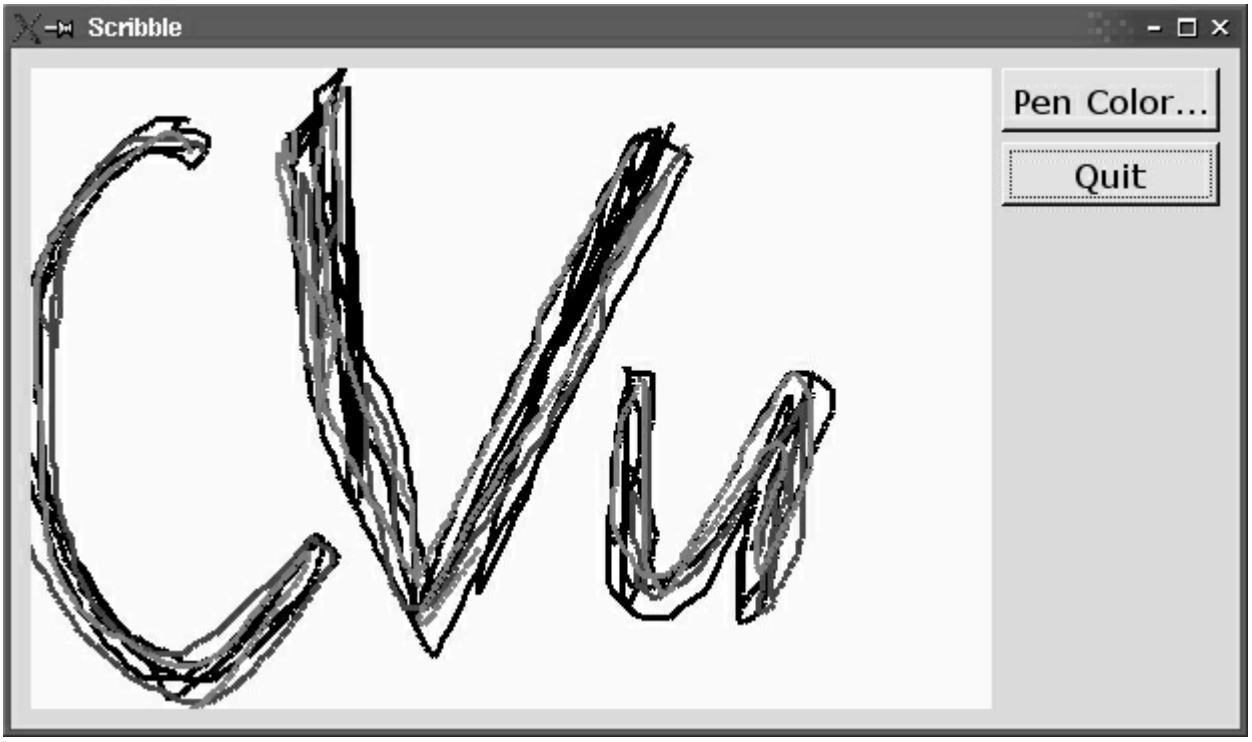


Figure 1: The Scribble Widget

```
void Scribble::setPenColor(const QColor
                                    &color) {
  m_color = color;
}
```

The `setPenColor()` function sets the current pen colour. This time we don't need to call `update()` because the operation doesn't affect the screen rendering of the widget (it only affects pixels that the user will draw in the future). We don't need to call `updateGeometry()` either because `m_color` isn't used when computing the size hint.

```
void Scribble::mousePressEvent(QMouseEvent
                                    *event) {
  if(event->button() == LeftButton)
    m_prevPos = event->pos();
}
```

The `mousePressEvent()` function is called whenever the user presses a mouse button while the mouse pointer is located on the widget. The `event` parameter gives additional information, such as the button that was pressed (`button()`) and the screen position of the mouse cursor when the button was pressed (`pos()`). If the user pressed the left button, we store the mouse position in `m_prevPos` for later use.

```
void Scribble::mouseMoveEvent(QMouseEvent
                                    *event) {
  if(event->state() & LeftButton) {
    QPainter painter(&m_pixmap);
    painter.setPen(QPen(m_color, 3));
    painter.drawLine(m_prevPos, event->pos());

    QRect rect(m_prevPos, event->pos());
    rect = rect.normalize();
    update(rect.x() - 1, rect.y() - 1,
           rect.width() + 2,
           rect.height() + 2);

    m_prevPos = event->pos();
  }
}
```

The `mouseMoveEvent()` function is called continuously when the user moves the mouse pointer while holding down a mouse button. The typical sequence of events is one "mouse press" event when the user presses a button, then a series of "mouse move" events that describe the path taken by the mouse pointer, and finally a "mouse release" event when the user releases the button.

We check if the left button is one of the buttons that are currently pressed. If this is the case we update `m_pixmap` and repaint the widget using `update()`.

We create a `QPainter` to draw on the pixmap. We set the pen to have the correct colour (`m_color`) and a thickness of 3 pixels. Then we draw a line from the previous mouse position (`m_prevPos`) to the new mouse position (`event->pos()`).

`QPainter` is the entrance door to Qt's paint engine. It provides functions to draw all sorts of geometric shapes (rectangles, circles, pie sections, Bezier curves, etc.) and supports transformations such as rotating and scaling. A `QPainter` object can be used to draw on a pixmap, a widget, a vector diagram or a printer.

Once we're done updating the pixmap we must update the on-screen version. The reductionist approach would be to call `update()` with no argument and be done with it; this would tell Qt to redraw the entire widget area, a somewhat expensive operation. Instead we compute the bounding rectangle for the line segment we just drew and pass it to `update()`.

At the end of the function, we update `m_prevPos` so that the next "mouse move" event will prolong the line segment we just drew.

```
void Scribble::paintEvent(QPaintEvent *event) {
  QPainter painter(this);
  painter.drawPixmap(0, 0, m_pixmap);
}
```

The `paintEvent()` function is called whenever the widget must be repainted. This can occur if the widget was temporarily obscured by another window and then made visible again, or as a result of calling `update()`. Here we simply draw the pixmap onto the widget.

At this point you might wonder why we bother drawing on a pixmap then transfer the pixmap onto the widget. Couldn't we draw directly on the widget instead, eliminating the need for `m_pixmap`? The answer is no. This is because we can't rely on the window system to keep a copy of the widget's pixels if the window is obscured or minimized. A well-behaved widget must implement `paintEvent()` and be able to redraw itself entirely at any moment.

The Application's Main Window

We are done implementing the custom widget. To make it useful, we need a window around it, with a "Pen Color..." button and a "Quit" button. Here's the class definition:

```
#ifndef WINDOW_H
#define WINDOW_H

#include <qwidget.h>
class Scribble;

class Window : public QWidget {
  Q_OBJECT
public:
  Window(QWidget *parent = 0);
private slots:
  void choosePenColor();
private:
  Scribble *m_scribble;
};

#endif
```

We can call the class `Window` because it will be the only window in the application. The class has one slot, `choosePenColor()`, which pops up a colour dialog.

```
Window::Window(QWidget *parent)
    : QWidget(parent) {
  m_scribble = new Scribble(this);
  m_scribble->setSizePolicy(
               QSizePolicy::Expanding,
               QSizePolicy::Expanding);

  QPushButton *penColorButton =
       new QPushButton(tr("Pen Color..."),
                       this);
  QPushButton *quitButton =
       new QPushButton(tr("Quit"), this);

  connect(penColorButton, SIGNAL(clicked()),
          this, SLOT(choosePenColor()));
  connect(quitButton, SIGNAL(clicked()),
          this, SLOT(close()));

  QGridLayout *layout = new QGridLayout(this);
```

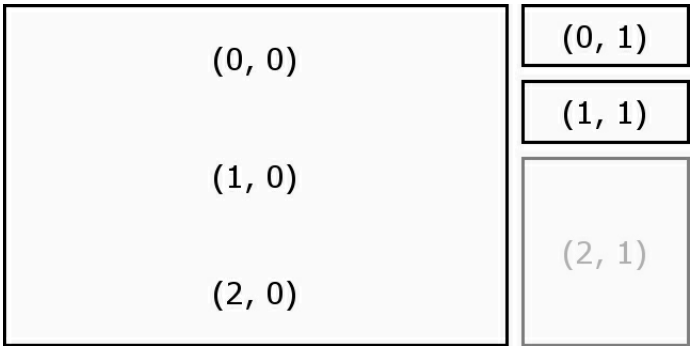


Figure 2: Grid Layout of Child Widgets

```
  layout->setMargin(10);
  layout->setSpacing(5);

  layout->addMultiCellWidget(m_scribble, 0, 2,
                             0, 0);
  layout->addWidget(penColorButton, 0, 1);
  layout->addWidget(quitButton, 1, 1);

  setCaption(tr("Scribble"));
}
```

In the constructor we create three child widgets (the scribble area and two push buttons), connect the "Pen Color..." button to the `choosePenColor()` slot, connect the "Quit" button to the window's `close()` slot, and put the child widgets in a grid layout. Figure 2 shows how the child widgets are laid out in the grid cells.

```
void Window::choosePenColor() {
  QColor color =
       QColorDialog::getColor(
                  m_scribble->penColor(), this);
  if(color.isValid())
    m_scribble->setPenColor(color);
}
```

When the user clicks "Pen Color...", we pop up a `QColorDialog` that allows the user to select a pen colour. We pass the old pen colour to the dialog as the initial value.

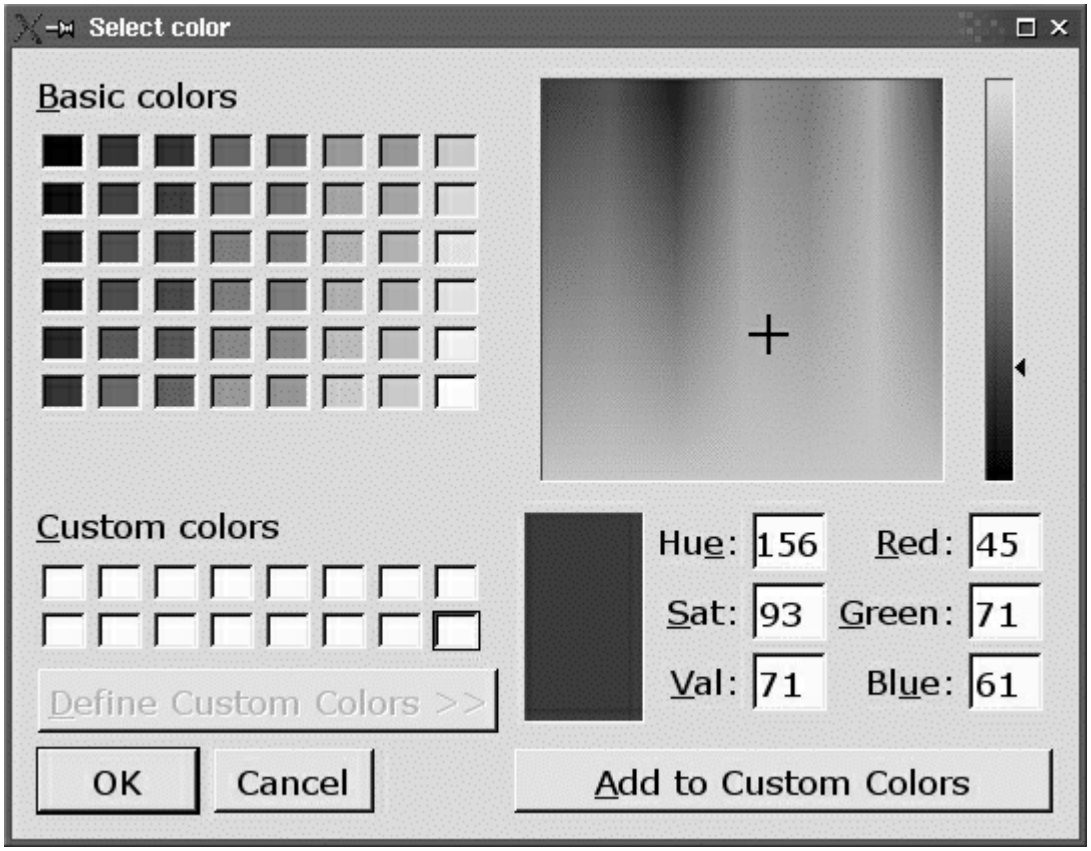


Figure 3: The Colour Dialog

This is all the code we need in `Window`. To complete the application, we need a `main()` function:

```
int main(int argc,
         char *argv[]){
  QApplication
       app(argc, argv);
  Window win;
  app.setMainWidget(
                  &win);
  win.show();
  return app.exec();
}
```

That's it! One of Qt's striking features is how easy it is to create custom widgets. In fact all of Qt's built-in widgets (e.g. `QPushButton` and `QColorDialog`) are implemented using the techniques described in this article. While with other toolkits writing custom widgets is considered an advanced topic, in Qt it is so easy that it is taught straight away to beginners as an introduction to the Qt way of thinking.

Jasmin Blanchette

Memory for a Short Sequence of Assignment Statements

Derek M. Jones <derek@knosof.co.uk>

The process of comprehending source code often involves reading some statements on a line by line basis. Some of the information read only needs to be remembered for a short period of time, while other information needs to be remembered over a longer period.

This article reports on an experimental study, carried out during the 2004 ACCU conference, that investigates the consequences of a limited capacity short term memory on subjects' performance in some of the tasks needed to comprehend short sequences of code. The source code used contained two commonly occurring constructs, assignment statements and `if` statements. Subjects' ability to recall the numeric values assigned to particular identifiers and to correctly deduce which arm of an `if` statement is executed were used as measures of their performance.

It is hoped that this study will provide information on the impact different kinds of identifier character sequences have on the cognitive resources needed during program comprehension.

Few developers appreciate how short the *short* in short term memory actually is. It only has the capacity to hold information on a few statements at most. It is hoped that the results of this study will bring home to developers the consequences of short term memory limitations on their code comprehension performance.

Also, advantage was taken of the `if` statements used in the experiment to try and duplicate the pattern of subject performance seen in some studies of human reasoning. The results seen in some of these studies suggest that the ordering of operands in a pair of relational expressions has an impact on people's performance in evaluating it.

This article is split into two parts, the first (this one) provides general background on the study. Part two discusses the results of the assignment problem and discusses the `if` statement results.

Characteristics of Human Memory

Models of human memory often divide it into two basic systems, short term memory (while the term *working memory* is sometimes used, this really refers to a collection of short term memory subsystems – see Figure 1) and long term memory. This two subsystem model is something of an idealization in that there is not a sharp boundary between short and long term memory; there is a gradual transition between them.

The phonological loop, which can hold approximately 2 seconds worth of sound, is the primary short term memory system of interest in this study. The information that can be held by the phonological loop is a sound-bite corresponding to 7±2 digits [11] spoken in English (the variation is highly correlated with differences in the rate at which people speak; faster speakers can remember more) and 9.9 digits spoken in Chinese.

While some of the characteristics of human memory (e.g. forgetting) are often criticized, they can provide useful functionality. It would make sense for human memory to be optimized for the information recall demands that frequently occur in everyday life and various studies [1] appear to confirm this evolutionary priority. For instance, forgetting is not necessarily the result of a poorly designed memory system. Studies [1] have found an exponential decay in the likelihood that information will be needed after a given period of time from when it was first encountered and that the rate at which information is *lost* from memory also has an exponential form...

People who can readily remember and later accurately recall information report that their conscious thoughts are repeatedly interrupted by 'unforgotten' information [13].

It could be claimed that the underlying problem is one of using of a computing platform (i.e. the human brain/mind) for a purpose for which it was not designed.

Information recall performance has been found to be affected by the extent to which the to-be-remembered information has associations with a person's existing network of memories.

Human Reasoning

A commonly used model of the human mind is that of a very powerful computer with the reasoning faculties based on mathematical logical. George Boole (after whom the term *boolean* is named) titled his book [3] *"An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities"*. However, the results of many studies are not consistent with this model of human reasoning.

Studies of various kinds of reasoning involving logical statements have discovered patterns in subjects' performance that are believed to be characteristic of how people solve reasoning problems. If the performance of subjects deducing the behaviour of source code `if` statements also exhibit patterns (e.g. differing numbers of errors made for different representations of the same logical condition), it may be possible to use this information to reduce the number of errors made by developers in comprehending source code. This topic is covered in detail in part 2 of this article, which discusses subject `if` statements evaluation performance.

Experimental Setup

The experiment was run by your author during one 45 minute session of the 2004 ACCU conference held in Oxford, UK. Approximately 300 people attended the conference, 40 (13%) of whom took part in the experiment. Subjects were given a brief introduction to the experiment, during which they filled out background information about themselves, and they then spent 20 minutes working through the problems. All subjects volunteered their time and were anonymous.

The Problem to be Solved

Each problem seen by subjects was intended to involve memory processes that operate over a time frame of approximately 30 seconds. It was expected that the characteristics of short term memory would have a significant impact on subjects' performance within this time frame.

To obtain statistically reliable data answers to a large number of related problems would be needed. Therefore it had to be possible to create a number of variations on the same underlying problem. Problems also had to be created that were not too easy or too difficult. If all subjects answered all questions correctly, or all incorrectly, no useful information would be obtained.

By using various rules of thumb (e.g. short term memory can contain two seconds worth of sound), simplifying assumptions (discussed below), and practising on himself, your author settled on a problem that involved recalling information about three assignment statements and selecting the appropriate arm of an `if` statement.

There are several reasons for using two kinds of statements in the code read by subjects:

- both kinds of statements occur frequently in source and using them together allowed the questions asked of subjects to reflect the kind of questions they have to answer when comprehending source code. The study thus has some claim to being *ecologically valid* (i.e. the behaviour in the experimental situation is characteristic of a real life environment)
- experience from another (unpublished) experiment found that when performing a single task some subjects became very focused on improving their performance by looking for, and using, patterns in the questions. It was hoped that by forcing subjects to switch between two tasks this unintended focusing behaviour would not be significant.

The following is an example of one of the problems seen by subjects. One side of a sheet of paper contained three assignment statements while the second side of the same sheet contained the `if` statements and a table to hold the recalled information. A series of X's were written on the second side to ensure that subjects could not see through to identifiers and values appearing

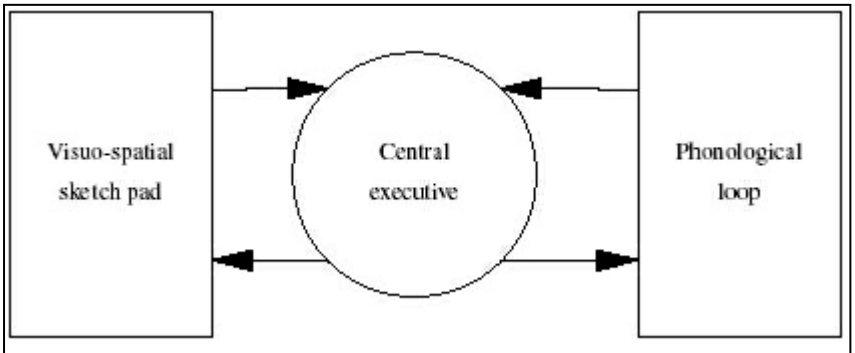


Figure 1: Model of working memory. The phonological loop can hold approximately 2 seconds worth of sound, while the visuo-spatial sketch pad holds a visual image that degrades within about 1.5 seconds. From Baddeley [2].

on the other side of the sheet. Each subject received a stapled set of sheets containing the instructions and 32 problems (one per sheet of paper).

```
——————— first side of sheet starts here ———————
prevented = 58;
liberation = 83;
conception = 94;
———————- second side of sheet starts here ———————
if ((e > a) && (u < a))
if (u > e)
.............
else
.............

            remember  would refer back  not seen
suspend       ____         ____          ____
prevented     ____         ____          ____
liberation    ____         ____          ____
conception    ____         ____          ____
```

The instructions given to subjects followed that commonly used in memory related experiments. Subjects see the material to be remembered, then perform an unrelated task (chosen to last long enough for the contents of short term memory to have degraded), and are then asked to recall the previously seen information.

The sequence *"remember->unrelated task->recall"* has an obvious parallel in source code comprehension; i.e. *"sequence of assignments->conditional test->use of identifiers previously assigned to"*.

In practice software developers do not make a remember/not remember decision, there is always the opportunity to refer back to previously read information. The selection remember/*would refer back* more accurately reflects the decision made by software developers.

The following written instructions were given to subjects:

This is not a race and there are no prizes for providing answers to all questions. Please work at a rate you might go at while reading source code.

The task consists of remembering the value of three different variables and recalling these values later. The variables and their corresponding values appear on one side of the sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. *Read the variables and the values assigned to them as you might when carefully reading lines of code in a function definition.*
2. *Turn the sheet of paper over. Please do NOT look at the assignment statements you have just read again, i.e. once a page has been turned it stays turned.*
3. *Assuming that the condition specified in the first if-statement is true, which arm of the nested if-statement will be executed? Treat the paper as if it were a screen, i.e. it cannot be written on. Mark the arm you think will be executed with a cross or a tick.*
4. *You are now asked to recall the value of the variables read on the previous page. There is an additional variable listed that did not appear in the original list.*
   - *if you remember the value of a variable write the value down next to the corresponding variable,*
   - *if you feel that, in a real life code comprehension situation, you would reread the original assignment, tick the "would refer back" column of the corresponding variable,*
   - *if you don't recall having seen the variable in the list appearing on the previous page, tick the "not seen" column of the corresponding variable.*

If you do complete all the questions do NOT go back and correct any of your previous answers.

The Set of Possible Questions

It was hoped that at least 32 people (on the day 40) would volunteer to take part in the experiment and it was estimated that each subject would be able to answer 32 problems (on the day 22.7) in 20-30 minutes (on the day 20 minutes). Based on these estimates the experiment would produce 1024 (on the day 884) answered problems.

Given the 8 different ways of ordering the operands and operators appearing in the chosen form of the `if` statement conditional expression and the 4 different questions that can be asked, it is possible to create 32 different `if` statement problems.

It was decided to use four sets of identifiers in the assignment problems, with each set containing four different identifiers. The possible values assigned to these identifiers were drawn from a set of four possible two digit integer literals (the rationale is discussed below). Given 16 possible identifiers and 4 possible numeric values (8 had been intended, but a bug in the generation script meant that only 4 were ever used), it is possible to generate 80,640 different sets of 3 assignments (the same identifier or value only being allowed to occur once in any set of assignments).

However, if all identifiers within a given set are considered to be equivalent and all two digit values are considered equivalent, then there are only 4 different sets of assignments (a set containing single digit constants had also been planned, which would have created 8 different sets of assignments).

Combining 32 different `if` statement problems with 4 different sets of assignment problems creates a total of 128 different problems (256 had been intended). Given 1024 answers then there would be 8 answers for each different problem (assuming subjects answered all problems).

The problems and associated page layout were automatically generated using a C program and various awk scripts to generate troff, which in turn generated postscript. The identifier and constant used in each assignment statement was randomly chosen from the appropriate set and the order of the assignment statements (for each problem) was also randomized. The (corrected) source code is available on the experiments web page.

Selecting Identifiers and Integer Constants

Studies have found that people's performance in processing character sequences can vary between different kinds of sequences. For instance, frequently used character sequences (i.e. words) are recognized faster and are more readily recalled than rare ones, also many performance characteristics are slower and more error prone for non-words compared to words, recognizing known subsequences (e.g. ibmchairs) within a longer character sequence allows it to be divided up into a smaller number of larger chunks (i.e. such recognition reduces information content and requires less storage resources).

Some of the factors affecting people's performance in recalling recently read information include:

- the encoding used for the information. For instance, a sequence having the same form as a word in a language known by a person can be encoded in a sequence of sounds that is shorter than the sequence of sounds representing the individual characters,
- the extent to which people are able to maintain the information in short term memory. This will depend on the short term memory resources consumed by the encoded information and other calls on short term memory resources between when the information is originally encoded and when it needs to be recalled,
- the extent to which the information is already stored in longer term memory subsystems. For instance, this information may exist because a character sequence has been encountered before, or its sound pattern matches (or rhythms with) that of a known word. It is also possible that a persons brain happens to store a given character sequence into a longer term memory subsystem, when it is encountered.

The identifier attributes varied in this study were the amount of short term memory storage required to hold their spoken form (the number of syllables was used as an approximate indicator of storage requirements; the effects of phonological complexity were ignored), and they were either a word (i.e. they were established in long term memory) or a sequence of unrelated characters. The identifiers thus belonged to one of four possible sets of character sequences.

Identifier Character Sequences

A variety of different kinds of character sequences are used to represent identifiers in source code. Some are recognisable words or phrases, some abbreviated forms of words or phrases, while others have no obvious association with any known language (e.g. they may be acronyms that are unknown to the reader). It is to be expected that subjects' memories of an identifier will be sound based, rather than vision based. For instance, a character sequence representing a known word is likely to be remembered as the spoken form of that word, while a sequence of unrelated characters might be remembered as the spoken form of each individual character.

Subjects are likely to have read many distinct character sequences every day for most of their lives. Many of these character sequences will have been stored in every subject's long term memory and be readily available for recall. Creating a character sequence that only evokes a response from a subject's short term memory is likely to be impossible. Whatever character sequence is chosen, it is likely that there will be some form of association with the contents of a subject's long term memory. The best that can be achieved is to use a set of character sequences, for identifiers, that all result in the contents of long term memory having the same impact on performance for all subjects.

Experience shows that developers sometimes read source code so quickly that visually similar, but different, identifiers are treated as being the same identifier. To reduce the possibility of this occurring during the experiment an attempt was made to use visually distinct character sequences (this involved arranging for ascending e.g. t, and descending e.g. p, characters to occur at different relative locations in a sequence).

All words used in the study had a frequency of occurrence of between 1 per 18 million words and 1 per 24 million words (word frequency counts were based on the British National Corpus). The Collins Advanced Learners English Dictionary was used for syllable counts.

The four sets of identifiers used in assignment statements were:

1 a single character whose spoken form contained a single syllable. The least frequently used letters in written English are *wybvkxjqz*. For reasons lost in the mists of time, the letters *wxyz* rather than the overall less frequent (and not sequential) *xjqz* were used,
2 an English word whose spoken form contained one syllable (i.e.van, guy, tip, mud),
3 three characters whose spoken form is likely to contain three syllables (i.e. vcq, qmt, bfj, rpl). That is the characters did not represent an English word. Google was used to reduce the possibility that the character sequence did not denote an acronym that was likely to be contained in subjects long term memory (e.g. IBM). Google returned a page count (at the start of 2004) of between 10,000 and 34,000 matches for the character sequences used (most other such sequences each returned over 100,000 matched pages).
4 an English word whose spoken form contained three syllables (i.e. conception, suspend, prevented, liberation).

In the rest of the article the term *short identifier* denotes an identifier whose spoken form is short (i.e. it contains a single syllable) and the term *long identifier* denotes an identifier whose spoken form is long (i.e. it contains three syllables). In practice the only reliable method of finding out the duration of the spoken form of word is to average the time taken by various people to say the word repetitively.

The character sequences first selected did not appear to share any common sounds that might result in increased interference between them when held together in short term memory[2].

`if` Statement Identifiers

It was intended that the only cause of interference between the identifiers used in the two forms of statements should be contention for short term memory resources. For this reason the identifiers chosen for the two kinds of statements were distinct, both in terms of visible appearance and sounding different.

The most frequently used letters in written English are *etaoinsrhldcu*. For reasons lost in the mists of time, the single letters *aeu* rather than overall more frequent (and not all vowel) *eta* were used,.

Selecting Integer Constants

Measurements of the frequency of integer constants in various contexts have found that some values occur more frequently than others. Measurements of source code have found various patterns between numeric values and the frequency with which they appear in the visible source code (see Figure 2).

The following integer constants were chosen (the digit 7 was not used in any value because its spoken form has two syllables):

- single digit numbers. The values 5, 6, 8, and 9 were chosen because they all have approximately the same frequency of occurrence in source code and other contexts, and have a spoken form containing a single syllable. However, due to a bug in the script generation program no single digit numbers were used in this study,
- two digit numbers. These have the advantage over three digit numbers in that they are all likely to be encoded using a single spoken form (many three digit numbers have many possible spoken forms e.g. 869 might be spoken as eight-six-nine or eight hundred and sixty nine).

Threats to Validity

Experience shows that software developers are continually on the lookout for ways to reduce the effort needed to solve the problems they are faced with. Because each of the problems seen by subjects in this study has the same structure it is possible that some subjects will have detected what they believe to be a pattern in the problems and will then attempt to use this information to improve their performance. Possible patterns appearing across problems include:

- a bug in the problem generation script meant that the identifier that did not appear in the list of assignment statements always appeared first in the list of to be recalled information. At least one subject noticed this pattern (he raised it during discussions after he completed the experiment),
- the number of identifiers used was a very small subset of those that could have been used. This meant that the first character of each character sequence was unique to that identifier (i.e. there was only one identifier starting with any given letter of the alphabet). At least one subject noticed this (in discussions after completing the study he said that he had saved time by only encoding the first few letters of the longer identifiers),
- the ordering of the identifiers in the assignment statements and in the to be filled in list of recalled information was the same. It is not known if any subjects noticed this pattern and used it to improve their performance.

While the kind of problems used commonly occur during program comprehension, the mode of working (i.e. paper and pencil) does not. Source code is invariably read within an editor and viewing is controlled via a keyboard or mouse. Referring back to previously seen information (e.g. assignment statements) requires pressing keys (or using a mouse). Having located the sought information more hand movements (i.e. key pressing or mouse movements) are needed to return to the original context. In this study subjects were only required to tick a box to indicate that they *would refer back* to locate the information. The cognitive effort needed to tick a box is likely to be less than would be needed to actually refer back. Studies have found that subjects make cost/benefit decisions when deciding whether to use the existing contents of memory (which may be unreliable) or to invest effort in relocating information in the physical world. It is possible that in some cases subjects ticked the *would refer back* option when in a real life situation they would have used the contents of their memory rather than expending the effort to actually refer back.

A previous experiment (unpublished), involving a source comprehension task that only contained conditionals, found that some subjects' solution strategies changed during the course of answering questions.

Initially these subjects obtained their answers by applying the traditional algebraic strategies usually associated with solving logic problems. However, developers' familiarity with problem solving is not confined to source code comprehension and is often applied to the problem of minimizing the effort they need to expend on the task.

In the case of having the solve a sequence of conditional problems some subjects switched to a pattern matching strategy. That is, they looked for (and claimed to have found) patterns in the questions that enabled them to quickly provide what they believed to be the correct answer (i.e. the answer

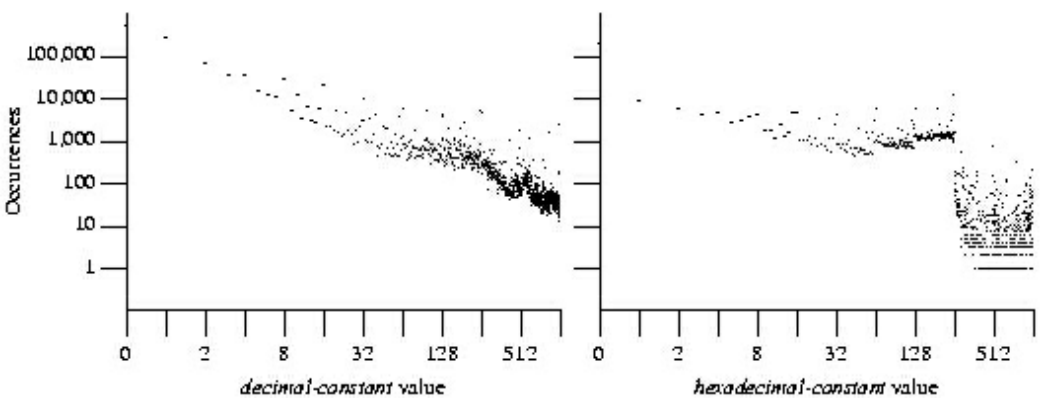


Figure 2: Occurrences, in the visible form of various applications written in C. of integer constants with different values.

1 Your author would not claim to any special knowledge on how common sounds (*phonological similarity* is the technically correct term and proposals have been made for measuring it) might be measured or which sounds might interfere with each other

to a question was based on matching it to a pattern having a known answer). It is possible that the intervening assignment problem did not provide sufficient cognitive demand (i.e. distraction) that in some cases subjects gave answers to the `if` statement problem based on patterns they believed to exist in the sequence of problems they saw.

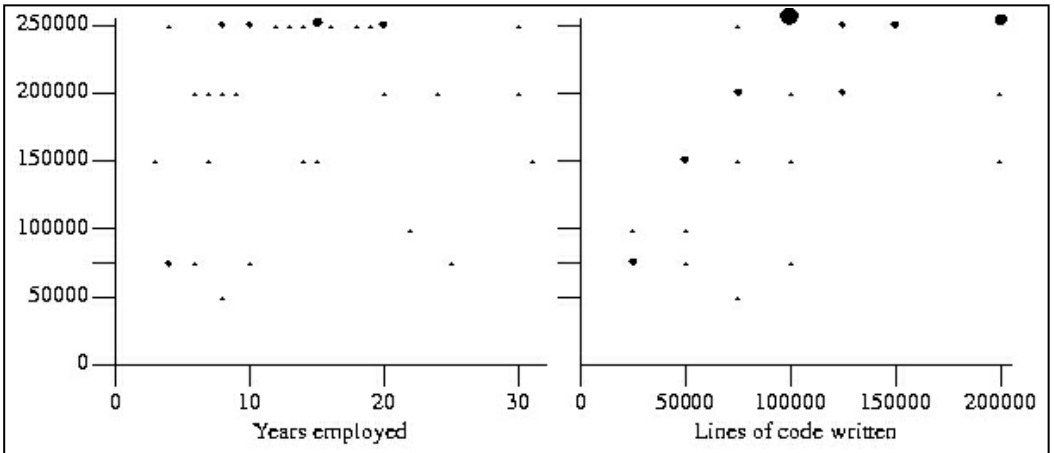


Figure 3: Developer Experience. The plot on the left depicts number of lines of code read against number of years of professional experience. The plot on the right depicts number of lines of code read against number of lines of code written, for each subject. The size of the circle indicates the number of subjects specifying the given values. In cases where subjects listed a range of values (i.e. 50,000-75,000) the median of that range was used.

Results

Subject Experience

Traditionally, developer experience is measured in number of years of employment performing some software related activity. However, the quantity of source code (measured in lines) read and written by a developer (developer interaction with source code overwhelmingly occurs in its written, rather than spoken, form) is likely to be a more accurate measure of source code experience than time spent in employment. Interaction with source code is rarely a social activity (a social situation occurs during code reviews) and the time spent on these activities may be small enough to ignore. The problem with this measure is that it is very difficult to obtain reliable estimates of the amount of source read and written by developers. This issue was also addressed in a study performed at a previous ACCU conference. While it was hoped that some of the problems encountered in that study were solved in the current study, the results (see Figure 3) suggest that the upper range of possible answers is still insufficient to cover the amount of code that subjects believe they have read.

Plotting the number of lines read against number of lines written gives a ratio of approximately 2.5 lines read per line written. Your author's experience suggests this ratio ought to be greater than 25.

One possible reason for this difference is that the questions asked (e.g. *How many lines of code would you estimate you have {read|written}, in total, over your career?*) are open to various interpretations. For instance, does reading previously read code count towards the total number of lines read (previously read lines that a developer has forgotten about might be thought to result in more learning than lines reread after a time delay of a few minutes), and how should changes that modify part of an existing line be counted?

It has to be accepted that reliable estimates of lines read/written are not likely to be available until developer behaviour is closely monitored (e.g. eye movements and key presses) over an extended period of time.

A plot of problems answered against experience (Figure 4) does not show any correlation between the two quantities. The number of subjects in each quadrant is approximately the same.

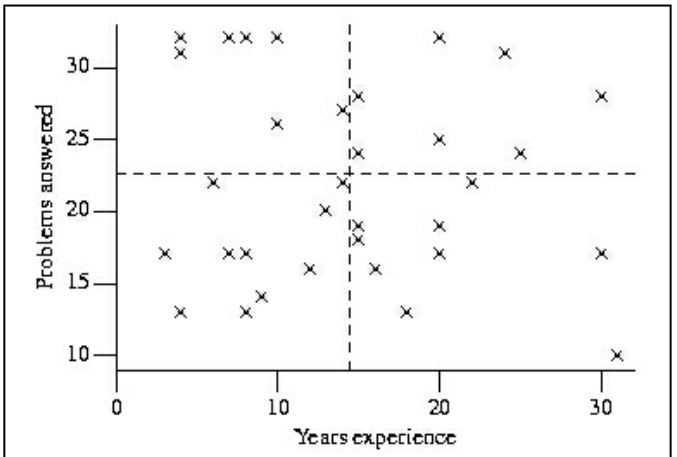


Figure 4: Plot of the number of problems answered against the number of years of professional experience of the subject. Dashed lines represent the mean number of problems answered (22.3) and the mean number of years of experience (14.5). The problems answered/years experience pairs (22, 6), (32, 8), and (19, 15) occurred for two subjects each.

Assignments

The following discussion breaks the results down by individual subject and by kind of identifier used in the assignment statements. The raw results for each subject are available on the studies web page. While there is enough raw data to perform detailed statistical analysis, none is performed. There are enough threats to validity to render the conclusions from any such detailed analysis spurious. However, it is hoped that some general conclusions can be drawn from the results obtained.

A total of 844 sets of assignment statements were remembered/recalled giving a total of 2,547 answers to individual assignments. The answer given to 43 assignment statement questions was an `x` in the *remember* column (only for some of the initial problems answered by a few subjects). This response was treated as indicating that the subject believed they knew the answer. However, since no value was specified it was not possible to verify the accuracy of the response. Therefore answers having this form were ignored (they were not counted in any category).

The number of incorrect *not seen* answers decreased from 13% (averaged over answers from all subjects) for the first eight problems to 7% for the ninth and subsequent problems. It is inevitable that some subjects will have noticed that the correct answers was always the first identifier in the response list. However, not all subjects noticed this pattern (i.e. they continued to give incorrect answers; in some cases a greater percentage of incorrect answers).

Individual Subject Performance

There is a great deal of variation in subject performance. Correct recall performance varied between 0 and 96%, instances where subjects would refer back varied between 0 and 94%, while incorrect answers varied between 1 and 40% of all answers given by any subject. This extreme variation suggests that the experimental design aim of creating problems whose solution stretched the limits of subjects' short term memory capacity was achieved. Had the problems required more or less short term memory capacity then it is likely that the variations in subjects' performance would have been narrower (i.e. subjects would have be likely to have provided a fewer or a greater number of incorrect or *would refer back* answers).

A *would refer back* response does not imply that a problem has exceeded a subject's short term memory capacity. It could imply that the subject is a very cautious individual, or that they were distracted by other thoughts while answering a particular problem.

If subject performance was consistent for all problems answered, it would be expected that averaged results for the first few problems answered would be the same as for the last few problems. Figure 5 plots *would refer back* and incorrect answer performance for the first eight and for the ninth and all subsequent problems answered. The lack of clustering of the bullets with the crosses means there was little correlation between the two sets of results. There are approximately twice as many dots below the crosses as there are above, which suggests that an individual's performance improved as more problems were answered.

One possible reason for an increase in performance is because answering problems enabled subjects to learn something that was beneficial in answering subsequent questions (e.g. the first identifier in the list of assignment questions was always the one that did not appear in the previous

assignment statements). One possible reason for a decrease in performance is that subjects became fatigued through having to answer so many questions that constantly stretched the capacity limits of their short term memory.

It might be thought that a subject answering a greater number of questions would be more likely to give incorrect or *would refer back* answers. Figure 6 shows that this is not the case. Fitting a least squares line through the data shows that both the percentage of incorrect and *would refer back* answers decreased as more questions were answered. As pointed out earlier it is possible that some subjects were able to detect and use patterns in the presentation of the problems to improve their performance. This improvement in performance could take the form of an increase in the number of problems answered as well as an increase in the number of correct answers.

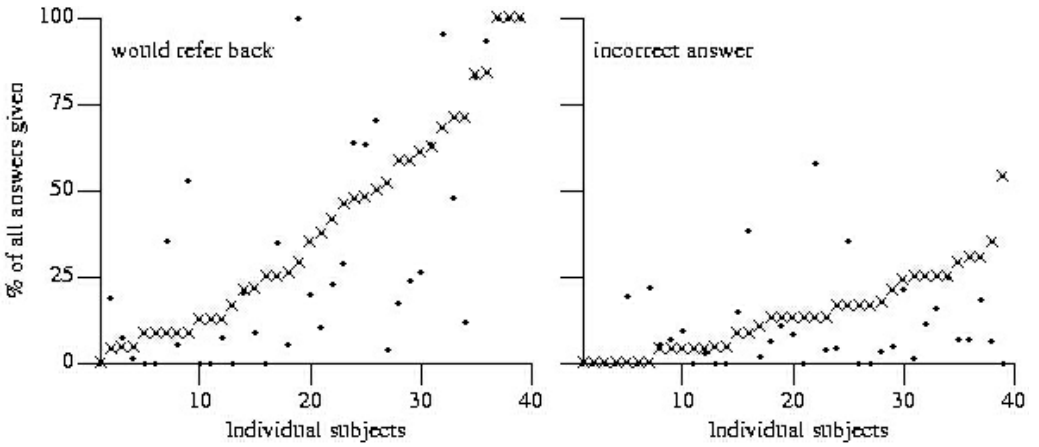


Figure 5: Left graph is the percentage of *would refer back* answers for the first 8 problems (crosses) and for the ninth and subsequent problems (bullet) answered (subjects are ordered by increasing *would refer back* response rates). Right graph is the percentage of incorrect answers with subjects being ordered by increasing percentage of incorrect.

Different Kinds of Identifiers

The analysis of individual subject results suggests that their performance improved as more problems were answered. The analysis of the results for different kinds of identifiers takes this behaviour into consideration by dividing the results in two; those from the first eight problems answered and those from the ninth and all subsequent problem answers.

There are a number of surprises in the results (Figure 7) (at least for your author):

1 for the first eight problems the pattern of answers for the identifiers composed of three unrelated letters does not follow that of the identifiers composed of three syllable words.

One explanation for the three unrelated letter behaviour is that these letter sequences are likely to be completely unknown to subjects (they were selected on this basis). When asked to recall information about previously seen assignment statements subjects were initially unable to make use of any longer term memory associations as a recall aid, and so opted for the *would refer back* option.

As more problems were answered, and subjects encountered more instances of the three unrelated letter sequences used, it is possible that some information about these letter sequences became stored in longer term memory subsystems and subjects were able to make use of this new existing knowledge.

2 for the first eight problems subject performance is best for short identifiers. For the ninth and subsequent problems the results showed what might be called a *word superiority effect* (i.e. a greater number of correct answers). This suggests that after some practice the contents of a person's longer term memory (i.e. their experience in using words) has a greater impact on performance than limits on their short term memory capacity.

The extent to which solving the `if` statement problem may have resulted in a degrading of the contents of short term memory (i.e. assignment statement information) is discussed in part two of this article.

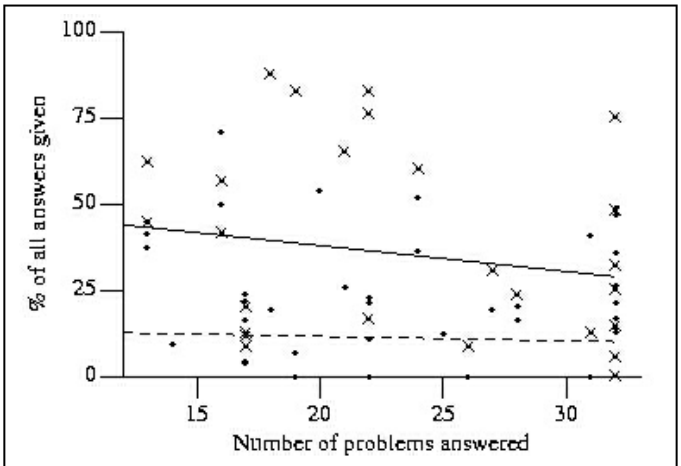


Figure 6: The percentage of would refer back answers (crosses, least squares line unbroken) and incorrect answers (bullet, least squares line dashed) plotted against the number of problems answered by each subject.

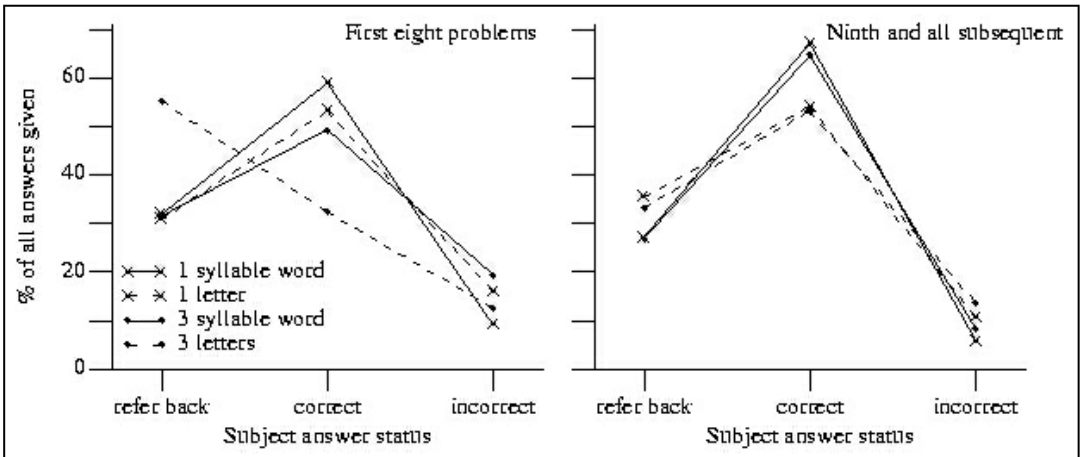


Figure 7: The percentage of would refer back, correct and incorrect answers for each kind of identifier, averaged over all subjects. The left graph is based on answers to the first eight problems, while the right graph uses the answers from the ninth and subsequent problems answered.

Kinds of Recall Errors

If the repetitive process of remembering assignment information caused the numeric values seen to be stored in longer term memory, then it would be expected that the set of values recalled in error would converge to the set of values seen during the experiment. The results (Table 1 on next page) show a small increase (between the first eight, and the ninth and subsequent answers) in the number of incorrect answers given that appear somewhere in the list of assignment statements that a subject saw for a given problem (fifth row). There is a larger increase in the number of incorrect answers given that come from the set of all values seen during the experiment (last row).

[concluded at foot of next page]

3 Subjects were not asked to provide a guess for those cases where they *would refer back*, so it is not possible to measure the accuracy of any information they might have believed they had on a given assignment statement.

Wx – A Live Port

Part 2: Connecting the User Interface to Code

Jonathan Selby <jon@xaxero.com>

The process of connecting the user interface to code is very similar to MFC. The event table is as follows:

MFC

```
ON_COMMAND(ID_VIEW_SOUNDINGS, OnViewSoundings)
ON_UPDATE_COMMAND_UI(ID_VIEW_SOUNDINGS,
OnUpdateViewSoundings)
```

wxWidgets

```
EVT_MENU(ID_SOUNDINGS,
WXWindPlotView::OnViewSoundings)
EVT_UPDATE_UI(ID_SOUNDINGS,
WXWindPlotView::OnUpdateViewSoundings)
```

For clarity I put the message/event handlers in the class that will actually be handling it – this is the way the class wizard built the application in the first place but this is not the a requirement. Class wizard builds the event table with a few mouse clicks. Under wxWidgets you have to code it manually, however there is very little work to do.

As with MFC the wxWidgets framework handles the UI changing in idle time.

Here is the implementation of the code:

```
void WXWindPlotView::OnViewSoundings() {
  WXWindPlotDoc* doc
            = (WXWindPlotDoc*)GetDocument();

  if(!Soundings->DepthFilePresent) {
    wxMessageBox("Ocean depth features require
                 Registration and the Xaxero
                 CD ROM.\nVisit www.xaxero.com
                 for details.");
    return;
  }

  if(bSound)
    bSound=FALSE;
  else
    bSound=TRUE;
  doc->UpdateAllViews(NULL,NULL );
}

void WXWindPlotView::OnUpdateViewSoundings(
                      wxUpdateUIEvent& event) {
  event.Check(bSound);
}
```

Almost identical to MFC – however one little pitfall to be careful of. If you want to set a check in an item make sure you have set it to checkable in the wxDesigner properties panel or you will get assertion errors in debug.

ARRAY Macros

Like MFC, wx allows an array of classes with its own dynamic array allocation. Here I am trying to define an array of email addresses.

[continued from previous page]

Discussion

Based on both years of employment and the claimed number of lines of code read/written the subjects taking part in the experiment have a significant amount of software development experience.

The number of years of software development experience is likely to have a high correlation with a subject's age. While cognitive performance has been found to decrease with age, age does not appear to have been a factor affecting the number of questions answered in this experiment (however, most subjects are likely to be younger than the age at which studies find a significant age decrease in performance; 50s and over).

The aim of creating a problem that would require approximately 30 seconds to answer was not met. The average time taken to answer problems was 67 seconds, over twice that intended in the experimental design. It is possible that a subject's short term memory resources were completely consumed by solving the `if` statement problem.

	first eight	ninth and subsequent	total
total recall errors	126	158	284
both digits incorrect	64 (51%)	104 (66%)	168
only first digit incorrect	34 (27%)	27 (17%)	61
only last digit incorrect	28 (22%)	27 (17%)	55
answer given in list	56 (44%)	76 (48%)	132
first digit in list	28 (22%)	23 (15%)	51
last digit in list	20 (16%)	18 (11%)	38
answer given in set	61 (48%)	91 (58%)	152

Table 1: Number of various kinds of recall errors made by subjects when answering the assignment problem. The percentage is calculated using the total at the top of the corresponding column. The phrase *in list* refers to the constant values appearing in the list of assignment statements read immediately prior to the `if` statement. The phrase *in set* refers to the set of all possible constant values appearing in assignment statements. The first digit is the most significant digit.

Given the experience of the subjects participating in this experiment any learning affects that occurred are likely to be caused by patterns in the presentation of the problems (e.g. particular identifiers always appearing in a given order). Known patterns include:

- using a relatively small, compared to the number of problems seen by a subject, set of identifiers. The results show that when answering the initial problems recall performance was significantly better for short identifiers. The change in performance characteristics, as subjects answered more problems, could have been caused by subjects learning the limited number of different identifiers used in the experiment, or it could have been caused by something else being learned. Repeating the experiment using a greater number of different identifiers will help answer this question,
- using a relatively small, compared to the number of problems seen by a subject, set of constant values. The issues here are the same as those for using a small set of identifiers,
- listing the identifiers in the same order in the recall list as they appeared in the assignment list. Subjects could have used this information to answer problems without remembering any identifier information. While identifiers sometimes need to be recalled in the same order in which they are read in the source, this is not always the case. Repeating the experiment using different relative orderings will remove this possible threat to validity,
- having the first identifier in the recall list as the identifier that did not appear in the assignment list. While the problem appears to be difficult enough without this identifier, its presence provides a mechanism for estimating the amount of guessing made by subjects in their answers.

More results are discussed in the second part of this article.

Derek M Jones

Further Reading

For a readable introduction to human memory see *Essentials of Human Memory* by Alan D. Baddeley. A more advanced introduction is given in *Learning and Memory* by John R. Anderson. An excellent introduction to many of the cognitive issues that software developers encounter is given in *Thinking, Problem Solving, Cognition* by Richard E. Mayer.

Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiment and the ACCU for making a conference slot available in which to run it.

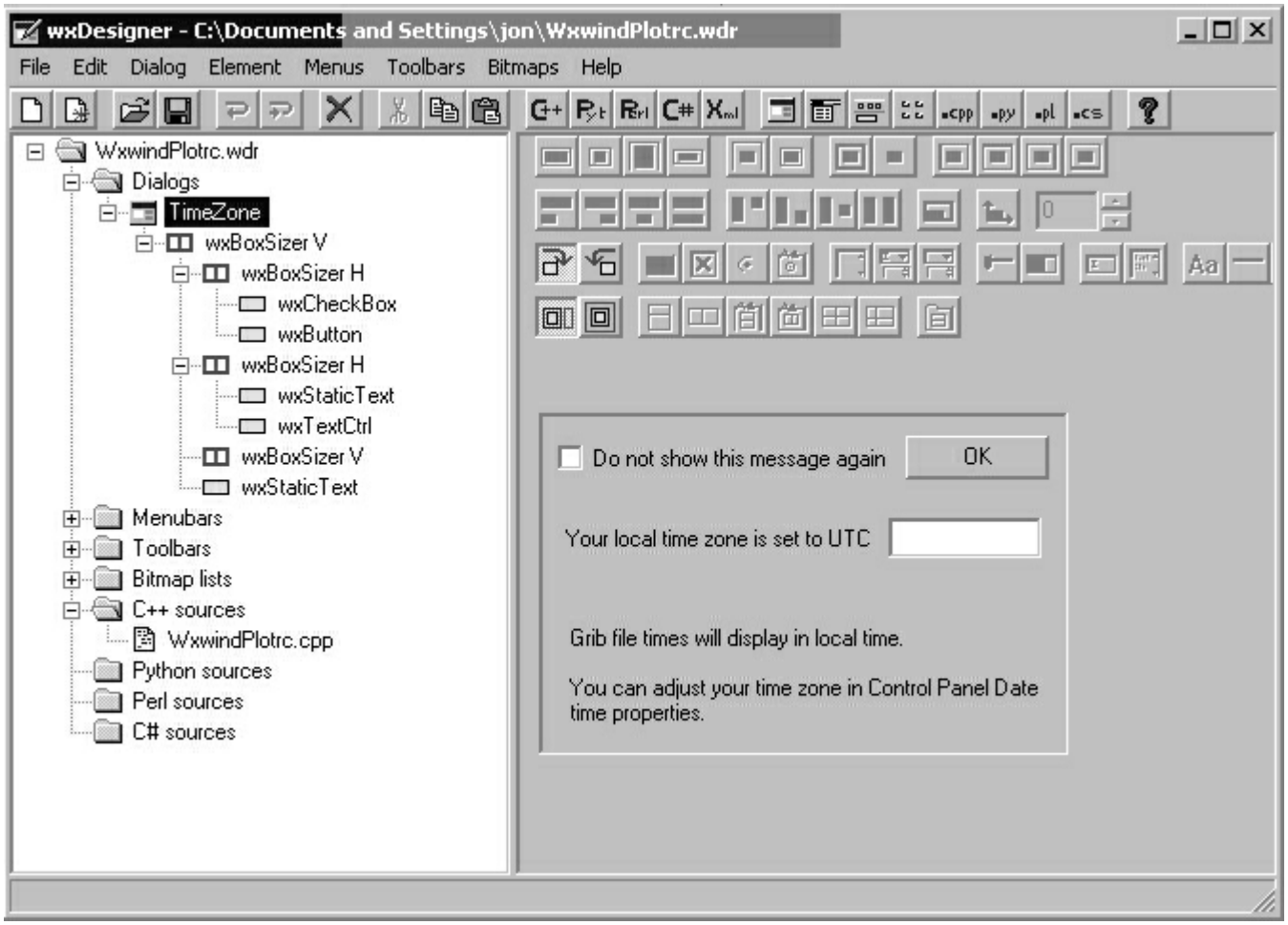


Figure 1: wxDesigner

Prior to the class that defines an email message we define an array container for the addresses:

```
WX_DECLARE_OBJARRAY(wxSMTPAddress,
                    arr_Recipients);

class Message {
  ...
  arr_Recipients
              Recipients;
};

#include <wx/arrimpl.cpp>
// this is a magic
// incantation which must
// be done!

WX_DEFINE_OBJARRAY(
         arr_ToRecipients);
```

I left the comment in the include statement. I had link errors when the `WX_DEFINE_OBJARRAY` was left off and so after reading the `wxArray` section of the documentation fully it all started to make sense. More important – I went to a clean compile and link.

Creating a dialog.

Using wxDesigner (see Figure 1) you lay out your dialog.

This simple example shows how 3 layers of vertical sizers encapsulate a box. The top two layers are horizontal sizers with adjacent controls. The hierarchy is a little tricky at first but when you have the hang of it, design goes really fast.

As you create the controls you will be giving them resource names similar to MFC. Now comes the tricky bit – to generate the code.

For the whole project I am using one `wdr` file containing all my dialogs, tool bars, etc. When you press the C++ button the code is written to `wxwindplotrc.cpp` that has the low level hard to read stuff that does the actual painting of the resources.

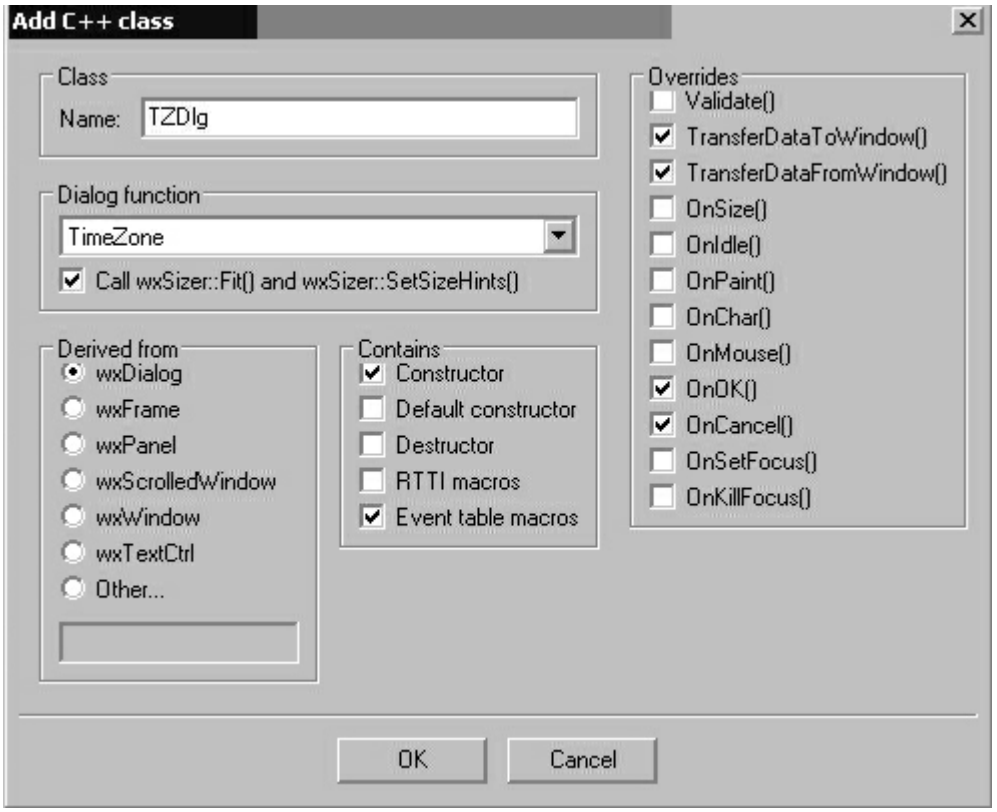


Figure 2: The Add A Class Dialog

We need to generate a dialog implementation class now. We press the `.cpp` button to add a C source. Give it a name (remember to add the suffix `.cpp`)

This will generate an empty C++ container.

We now need to implement a class for our dialog. Easy enough. Press the Class button (see Figure 2).

We have added the handlers on the right – everything we wanted our dialog to do. We respond to either the OK or the cancel messages and we have handlers to move data to the dialog and out of it.

That is all there is to it – a shell is created that will compile and run. It will not do anything yet though.

We have two ids that we wish to manipulate from the code.

- A check box: `ID_TZCHECK`
- An edit box: `ID_TZDISP`

We need to add inline functions to read and write from these ids.

Press the Get button on the source code editor to add the get functions:

Select the dialog you wish to use and press on the field you want to allow reading and writing from.

The program will select a method name. You can alter this if you like. The Add getter button places it in your code. Now we have:

```
// WDR: method declarations for MyDialog
-> wxCheckBox* GetTzcheck() { return
      (wxCheckBoxl*) FindWindow(ID_TZCHECK); }
-> wxTextCtrl* GetTzdisp()  { return
        (wxTextCtrl*) FindWindow(ID_TZDISP); }
```

So far wxDesigner has been doing all the work for us. Now we have to roll up our sleeves and start writing code.

Before we leave the header file we need to add variables to hold our values. A `wxString` – `DispUTC` and `BOOL Check`.

In the code we need to connect this to actions.

Look at the constructor – the first line should connect the dialog resource name to code. The first line of the constructor is generated as:

```
MyDialogFunc(this, TRUE);
```

Double-check this is what you want. Normally a meaningful name is generated as specified. This will be in the constructor of the generated source file.

Next we go to the following functions and flesh them out.

```
bool MyDialog::TransferDataToWindow() {
  // wxDesigner has added two getters, used to
  // set the values on startup and retrieve them
  // when closing the dialog (next method).
  GetTzdisp()->SetValue(DispUTC);
  GetTzcheck()->SetValue(Check);
  return TRUE;
}

bool MyDialog::TransferDataFromWindow() {
  DispUTC = GetTzdisp()->GetValue();
  Check = GetTzcheck()->GetValue();
  return TRUE;
}
```

There we are – a fully working dialog.

You can include a bunch of dialogs in one chunk of source code – useful for keeping wizard and notebook pages together.

Invoking the Dialog

In your code include the `tzdlg.h` file and a calling subroutine in the header of the calling program and create a call in the body.

Invoke the dialog as follows:

```
TZDlg dialog(GetMainFrame(), -1,
             wxT("Time Zone Display"));
... Initialization
dialog.ShowModal();
```

As easy as that !

Jonathan Selby

Resources

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

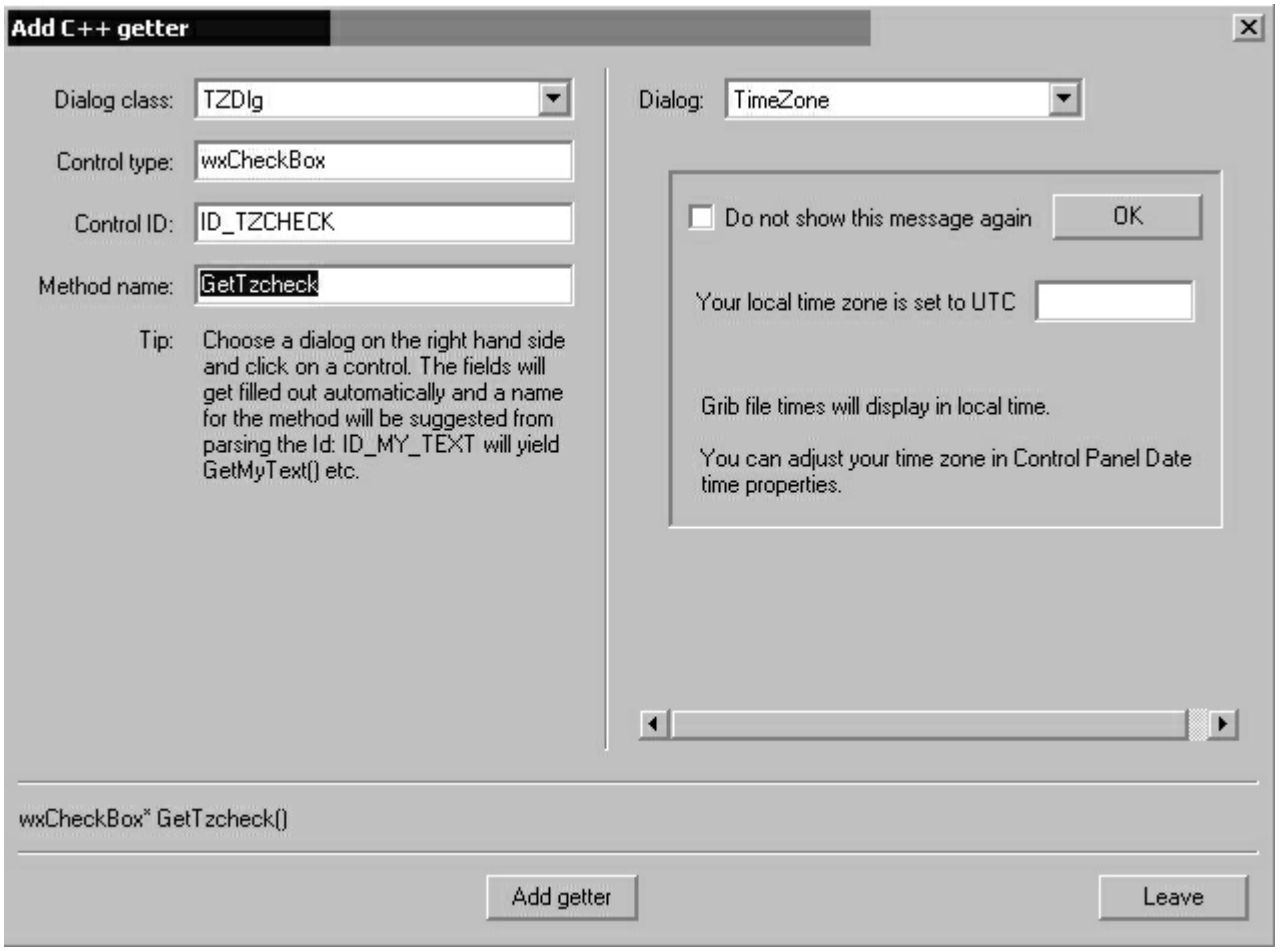


Figure 3: The Add C++ Getter Dialog

An Introduction to Programming with GTK+ and Glade in ISO C and ISO C++ - Part 3

Roger Leigh <rleigh@debian.org>

GTK+ and GObject

In the previous sections, the user interface was constructed entirely by hand, or automatically using `libglade`. The callback functions called in response to signals were simple C functions. While this mechanism is simple, understandable and works well, as a project gets larger the source will become more difficult to understand and manage. A better way of organising the source is required.

One very common way of reducing this complexity is *object-orientation*. The GTK+ library is already made up of many different objects. By using the same object mechanism (`Gobject`), the ogcalc code can be made more understandable and maintainable

The ogcalc program consists of a `GtkWindow` which contains a number of other `GtkWidget`s and some signal handler functions. If our program was a class (`Ogcalc`) which derived from `GtkWindow`, the widgets the window contains would be member variables and the signal handlers would be member functions (methods). The user of the class wouldn't be required to have knowledge of these details, they just create a new `Ogcalc` object and show it. By using objects one also gains *reusability*. Previously only one instance of the object at a time was possible, and `main()` had explicit knowledge of the creation and workings of the interface.

This example bears many similarities with the C++ Glade example (next edition). Some of the features offered by C++ may be taken advantage of using plain C and `GObject`.

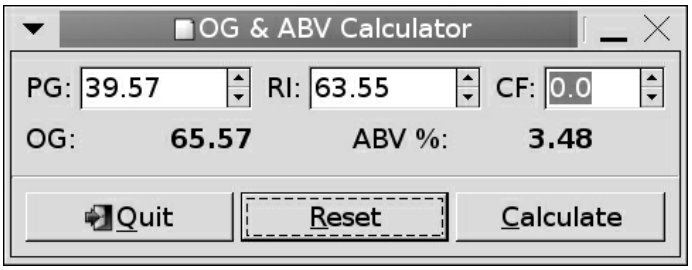


Figure 1: `C/gobject/ogcalc` in action.

The listings for the code are given at the end of the article (next two pages).

To build the source, do the following:

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

Analysis

The bulk of the code is the same as in previous sections, and so describing what the code does will not be repeated here. The `Ogcalc` class is defined in `C/gobject/ogcalc.h`. This header declares the object and class structures and some macros common to all GObject-based objects and classes. The macros and internals of `GObject` are out of the scope of this document, but suffice it to say that this boilerplate is required, and is identical for all `GObject` classes bar the class and object names.

The object structure (`_Ogcalc`) has the object it derives from as the first member. This is very important, since it allows casting between types in the inheritance hierarchy, since all of the object structures start at an offset of 0 from the start address of the object. The other members may be in any order. In this case it contains the Glade XML interface object and the widgets required to be manipulated after object and interface construction. The class structure (`_OgcalcClass`) is identical to that of the derived class (`GtkWindowClass`). For more complex classes, this might contain virtual function pointers. It has many similarities to a C++ `vtable`. Finally, the header defines the public member functions of the class.

The implementation of this class is found in `C/gobject/ogcalc.c`. The major difference to previous examples is the class registration and the extra functions for object construction, initialisation and notification of destruction. The body of the methods to reset and calculate are identical to previous examples.

`ogcalc_get_type()` is used to get the the typeid (`GType`) of the class. As a side effect, it also triggers registration of the class with the GType type system. Remember, Gtype is a *dynamic* type system. Unlike languages like C++, where the types of all classes are known at compile-time, the majority of all the types used with GTK+ are registered on demand, except for the primitive data types and the base class `GObject` which are registered as *fundamental* types. As a result, in addition to being able to specify constructors and destructors for the object (or *initialisers* and *finalisers* in Gtype parlance), it is also possible to have initialisation and finalisation functions for both the *class* and *base*. For example, the class initialiser could be used to fix up the vtable for overriding virtual functions in derived classes. In addition, there is also an `instance_init` function, which is used in this example to initialise the class. It's similar to the constructor, but is called after object construction.

All these functions are specified in a `GTypeInfo` structure which is passed to `g_type_register_static()` to register the new type.

`ogcalc_class_init()` is the class initialisation function. This has no C++ equivalent, since this is taken care of by the compiler. In this case it is used to override the `finalize()` virtual function in the `GObjectClass` base class. This is used to specify a virtual destructor (it's not specified in the `GTypeInfo` because the destructor cannot be run until after an instance is created, and so has no place in object construction). With C++, the vtable would be fixed up automatically; here, it must be done manually. Pure virtual functions and default implementations are also possible, as with C++.

`ogcalc_init()` is the object initialisation function (C++ constructor). This does a similar job to the `main()` function in previous examples, namely contructing the interface (using Glade) and setting up the few object properties and signal handlers that could not be done automatically with Glade. In this example, a second argument is passed to `glade_xml_new()`; in this case, there is no need to create the window, since our `Ogcalc` object *is a* window, and so only the interface rooted from `ogcalc_main_vbox` is loaded.

`ogcalc_finalize()` is the object finalisation function (C++ destructor). It's used to free resources allocated by the object, in this case the GladeXML interface description. `g_object_unref()` is used to decrease the reference count on a `GObject`. When the reference count reaches zero, the destructor is run and then the object is destroyed. There is also a `dispose()` function called prior to `finalize()`, which may be called multiple times. Its purpose is to safely free resources when there are cyclic references between objects, but this is not required in this simple case.

An important difference with earlier examples is that instead of connecting the window `destroy` signal to `gtk_main_quit()` to end the application by ending the GTK+ main loop, the `delete` signal is connected to `ogcalc_on_delete_event()` instead. This is because the default action of the `delete` event is to trigger a `destroy` event. The object should not be destroyed, so by handling the `delete` signal and returning `TRUE`, destruction is prevented. Both the "Quit" button and the `delete` event end up calling `gtk_widget_hide()` to hide the widget rather than `gtk_main_quit()` as before.

Lastly, `C/gobject/ogcalc-main.c` defines a minimal `main()`. The sole purpose of this function is to create an instance of `Ogcalc`, show it, and then destroy it. Notice how simple and understandable this has become now that building the UI is where it belongs – in the object construction process. The users of `Ogcalc` need no knowledge of its internal workings, which is the advantage of encapsulating complexity in classes.

By connecting the `hide` signal of the `Ogcalc` object to `gtk_main_quit()` the GTK+ event loop is ended when the user presses "Quit" or closes the window. By not doing this directly in the class it is possible to have as many instances of it as one likes in the same program, and control over termination is entirely in the hands of the user of the class – where it should be.

Roger Leigh

Listing 1: `C/gobject/ogcalc.h`

```
#include <gtk/gtk.h>
#include <glade/glade.h>
/* The following macros are GObject boilerplate. */

/* Return the GType of the Ogcalc class. */
#define OGCALC_TYPE (ogcalc_get_type())

/* Cast an object to type Ogcalc.  The object must
   be of type Ogcalc, or derived from Ogcalc for
   this to work.
   This is similar to a C++ dynamic_cast<>. */
#define OGCALC(obj) \
  (G_TYPE_CHECK_INSTANCE_CAST ((obj), \
                      OGCALC_TYPE, Ogcalc))

/* Cast a derived class to an OgcalcClass. */
#define OGCALC_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_CAST ((klass), \
                      OGCALC_TYPE, OgcalcClass))

/* Check if an object is an Ogcalc. */
#define IS_OGCALC(obj) \
  (G_TYPE_CHECK_TYPE ((obj), OGCALC_TYPE))

/* Check if a class is an OgcalcClass. */
#define IS_OGCALC_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_TYPE ((klass), \
                            OGCALC_TYPE))

/* Get the OgcalcClass class. */
#define OGCALC_GET_CLASS(obj) \
  (G_TYPE_INSTANCE_GET_CLASS ((obj), \
                      OGCALC_TYPE, OgcalcClass))

/* The Ogcalc object instance type. */
typedef struct _Ogcalc Ogcalc;
/* The Ogcalc class type. */
typedef struct _OgcalcClass OgcalcClass;

/* The definition of Ogcalc. */
struct _Ogcalc {
  GtkWindow parent;
    /* The object derives from GtkWindow. */
  GladeXML *xml;    /* The XML interface. */
  /* Widgets contained within the window. */
  GtkSpinButton *pg_val;
  GtkSpinButton *ri_val;
  GtkSpinButton *cf_val;
  GtkLabel *og_result;
  GtkLabel *abv_result;
  GtkButton* quit_button;
  GtkButton* reset_button;
  GtkButton* calculate_button;
};

struct _OgcalcClass {
  /* The class derives from GtkWindowClass. */
  GtkWindowClass parent;
  /* No other class properties are required (e.g.
     virtual functions). */
};

/* The following functions are described in ogcalc.c */
GType ogcalc_get_type(void);
Ogcalc * ogcalc_new(void);
gboolean ogcalc_on_delete_event(Ogcalc *ogcalc,
                                GdkEvent *event,
                                gpointer data);
void ogcalc_reset(Ogcalc *ogcalc, gpointer data);
void ogcalc_calculate(Ogcalc *ogcalc,
                      gpointer data);
```

Listing 2: `C/gobject/ogcalc.c`

```
#include "ogcalc.h"

static void ogcalc_class_init(OgcalcClass *klass);
static void ogcalc_init(GTypeInstance *instance,
                        gpointer g_class);
static void ogcalc_finalize(Ogcalc *self);

/* Get the GType of Ogcalc.  This has the side
   effect of registering Ogcalc as a new GType if it
   has not already been registered. */
GType ogcalc_get_type(void) {
  static GType type = 0;
  if(type == 0) {
    /* GTypeInfo describes a GType.  In this case,
       we only specify the size of the class and
       object instance types, along with an
       initialisation function.  We could have also
       specified both class and object
       constructors and destructors here as well. */
    static const GTypeInfo info = {
      sizeof (OgcalcClass),
      NULL,
      NULL,
      (GClassInitFunc) ogcalc_class_init,
      NULL,
      NULL,
      sizeof(Ogcalc),
      0,
      (GInstanceInitFunc) ogcalc_init
    };
    /* Actually register the type using the above
       type information.  We specify the type we are
       deriving from, the class name and type
       information. */
    type = g_type_register_static(GTK_TYPE_WINDOW,
                                  "Ogcalc", &info,
                                  (GTypeFlags) 0);
  }
  return type;
}

/* This is the class initialisation function.  It
   has no comparable C++ equivalent, since this is
   done by the compiler. */
static void ogcalc_class_init(OgcalcClass *klass) {
  GObjectClass *gobject_class
                        = G_OBJECT_CLASS (klass);
  /* Override the virtual finalize method in the
     GObject class vtable (which is contained in
     OgcalcClass). */
  gobject_class->finalize
           = (GObjectFinalizeFunc) ogcalc_finalize;
}

/* This is the object initialisation function.  It
   is comparable to a C++ constructor.  Note the
   similarity between "self" and the C++ "this"
   pointer. */
static void ogcalc_init(GTypeInstance *instance,
                        gpointer g_class) {
  Ogcalc *self = (Ogcalc *) instance;
  /* Set the window title */
  gtk_window_set_title(GTK_WINDOW (self),
                       "OG & ABV Calculator");
  /* Don't permit resizing */
  gtk_window_set_resizable(GTK_WINDOW (self), FALSE);
  /* Connect the window close button ("destroy-
     event") to a callback. */
  g_signal_connect(G_OBJECT (self), "delete-event",
                   G_CALLBACK (ogcalc_on_delete_event),
                   NULL);
```

```
  /* Load the interface description. */
  self->xml = glade_xml_new("ogcalc.glade",
                            "ogcalc_main_vbox", NULL);

  /* Get the widgets. */
  self->pg_val = GTK_SPIN_BUTTON
    (glade_xml_get_widget (self->xml, "pg_entry"));
  self->ri_val = GTK_SPIN_BUTTON
    (glade_xml_get_widget (self->xml, "ri_entry"));
  self->cf_val = GTK_SPIN_BUTTON
    (glade_xml_get_widget (self->xml, "cf_entry"));
  self->og_result = GTK_LABEL
    (glade_xml_get_widget (self->xml, "og_result"));
  self->abv_result = GTK_LABEL
    (glade_xml_get_widget (self->xml, "abv_result"));
  self->quit_button = GTK_BUTTON
    (glade_xml_get_widget (self->xml, "quit_button"));
  self->reset_button = GTK_BUTTON
    (glade_xml_get_widget (self->xml, "reset_button"));
  self->calculate_button = GTK_BUTTON
    (glade_xml_get_widget (self->xml,
                           "calculate_button"));

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

  g_signal_connect_swapped
    (G_OBJECT (self->cf_val), "activate",
     G_CALLBACK (gtk_window_activate_default),
     (gpointer) self);
  g_signal_connect_swapped
    (G_OBJECT (self->calculate_button), "clicked",
     G_CALLBACK (ogcalc_calculate),
     (gpointer) self);
  g_signal_connect_swapped
    (G_OBJECT (self->reset_button), "clicked",
     G_CALLBACK (ogcalc_reset),
     (gpointer) self);
  g_signal_connect_swapped
    (G_OBJECT (self->quit_button), "clicked",
     G_CALLBACK (gtk_widget_hide),
     (gpointer) self);

  /* Get the interface root and pack it into our
     window. */
  gtk_container_add
    (GTK_CONTAINER (self), glade_xml_get_widget(
               self->xml, "ogcalc_main_vbox"));

  /* Ensure calculate is the default.  The Glade
     default was lost since it wasn't in a window
     when the default was set. */
  gtk_widget_grab_default
    (GTK_WIDGET (self->calculate_button));
}

/* This is the object initialisation function.  It is
  comparable to a C++ destructor.  Note the similarity
  between "self" and the C++ "this" pointer. */
static void ogcalc_finalize(Ogcalc *self) {
  /* Free the Glade XML interface description. */
  g_object_unref(G_OBJECT(self->xml));
}

/* Create a new instance of the Ogcalc class (i.e.
   an object) and pass it back by reference. */
Ogcalc * ogcalc_new(void) {
  return (Ogcalc *) g_object_new(OGCALC_TYPE, NULL);
}

/* This function is called when the window is about
   to be destroyed (e.g. if the close button on the
   window was clicked).  It is not a destructor. */
gboolean ogcalc_on_delete_event(Ogcalc *ogcalc,
          GdkEvent *event, gpointer user_data) {
  gtk_widget_hide(GTK_WIDGET (ogcalc));
  /* We return true because the object should not be
     automatically destroyed. */
  return TRUE;
}

/* Reset the interface. */
void ogcalc_reset(Ogcalc *ogcalc, gpointer data) {
  gtk_spin_button_set_value(ogcalc->pg_val, 0.0);
  gtk_spin_button_set_value(ogcalc->ri_val, 0.0);
  gtk_spin_button_set_value(ogcalc->cf_val, 0.0);
  gtk_label_set_text(ogcalc->og_result, "");
  gtk_label_set_text(ogcalc->abv_result, "");
}

/* Perform the calculation. */
void ogcalc_calculate(Ogcalc *ogcalc, gpointer data) {
  gdouble pg, ri, cf, og, abv;
  gchar *og_string;
  gchar *abv_string;
  pg = gtk_spin_button_get_value (ogcalc->pg_val);
  ri = gtk_spin_button_get_value (ogcalc->ri_val);
  cf = gtk_spin_button_get_value (ogcalc->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(ogcalc->og_result, og_string);
  gtk_label_set_markup(ogcalc->abv_result, abv_string);
  g_free(og_string);
  g_free(abv_string);
}
```

Listing 3: C/gobject/ogcalc-main.c

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

/* This main function merely instantiates the ogcalc
   class and displays its main window. */
int main(int argc, char *argv[]) {
  /* Initialise GTK+. */
  gtk_init(&argc, &argv);
  /* Create an Ogcalc object. */
  Ogcalc *ogcalc = ogcalc_new();
  /* When the widget is hidden, quit the GTK+ main
     loop. */
  g_signal_connect(G_OBJECT (ogcalc), "hide",
                   G_CALLBACK (gtk_main_quit), NULL);

  /* Show the object. */
  gtk_widget_show(GTK_WIDGET (ogcalc));

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

  /* Clean up. */
  gtk_widget_destroy(GTK_WIDGET (ogcalc));
  return 0;
}
```

What's in a Namespace?

Paul Grenyer

In my experience most C++ developers have heard about namespaces. Most of them understand what namespaces are for and the problems they solve. Some even make use of them!

Namespaces can be used for more than preventing name clashes. In this article I will visit the mechanics of namespaces and anonymous namespaces and explain how they are used to solve some of the problems associated with linking C++ programs. Then I will move on to explain how they can also be used to provide context.

What are Namespaces?

The C++ standard has the following description of namespaces:

> 7.3.0.1 A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

This tells you what a namespace is, but not what one is used for. Consider the following example:

You are writing a COM object that is going to be used to split an input file into a number of output files. For maximum ease of testability and performance you write the actual file processing code in standard C++ and wrap it in a Façade [Façade] called `FileSplitter`. A COM object can then be written to wrap the file processing `FileSplitter` class. The COM object provides an interface that forwards to the file processing `FileSplitter` class.

The COM object client has no knowledge that the COM object is actually just a wrapper, and has no need to know. As far as the client is concerned the COM object is the file splitter. Therefore the obvious name for the COM object class is also `FileSplitter` (with `IFileSplitter` the obvious name for the interface).

Having two classes with the same fully qualified name in a C++ program is not permitted. The solution is to introduce namespaces. From Microsoft Visual C++ 7.0 onwards, all COM classes are placed in the `ATL` namespace (in earlier versions COM classes were required to be in the global namespace). However, although I will use the`ATL` namespace in this article; and putting COM objects in the `ATL` namespace is a Microsoft convention, using the name of a technology for a namespace is not usually good practice as it does not provide the right sort of context. For example it would not be sensible or useful to group together all abstract base classes or all classes that implement a recognized pattern.

Due to the limited scope of this example the name for the namespace containing the file processing `FileSplitter` class is not clear. However, an appropriate name might be something like `Process`, as the class performs the actual processing of files within the program:

```
// filesplitter.h

namespace Process {
  class FileSplitter {
    ...
  };
}
```

The file processing `FileSplitter` class can then be used by fully qualifying its name in the COM class:

```
// filesplitter_com.h

#include "filesplitter.h"

namespace ATL {
  class FileSplitter : public IFileSplitter {
    ...
  private:
    Process::FileSplitter impl_;
  };
};
```

Both classes can now happily coexist in the same program, despite the fact that they have the same name, as they are both in different namespaces. The namespaces also help to make maintenance easier by providing local context for each class.

What are Anonymous Namespaces?

The C++ standard has the following to say about anonymous (or unnamed) namespaces:

> 7.3.1.1 An unnamed-namespace-definition behaves as if it were replaced by
>
> ```
> namespace unique{ /* empty body */ }
> using namespace unique;
> namespace unique { namespace-body }
> ```
>
> where all occurrences of *`unique`* in a translation unit are replaced by the same identifier and this identifier differs from all other identifiers in the entire program. (Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.)

This is an even less useful description than the one for regular namespaces. Bjarne Stroustrup has the following to say about unnamed namespaces in The C++ Programming Langue [TCPPPL]:

> It is often useful to wrap a set of declarations in a namespace simply to protect against the possibility of name clashes. That is, aim to preserve locality of code rather than to present an interface for users...
> In this case we can simply leave the namespaces without a name...
> Clearly, there has to be some way of accessing members of an unnamed namespace from the outside. Consequently, an unnamed namespace has an implied using-directive...
> ...In particular unnamed namespaces are different in different translation units. As desired, there is no way of naming a member of an unnamed namespace from another translation unit.

This gets much closer to what an anonymous namespace is for, but is still not as clear as it could be. Mark Radford was kind enough to supply me with the following description and examples of the use of anonymous namespaces:

> Designers of C++ programs often encounter a need for some declarations to have Translation Unit (TU) scope. For example, consider the encapsulation of database access using SQL: it may well make sense for the SQL strings to be encapsulated within the TU in which the database access is implemented.
> Having declared identifiers for string constants within a TU, the designer has another issue to resolve: what if the same identifier is used in another TU? Without support from the C++ language, it may not be possible to guarantee avoiding this situation, without telling other people what identifiers have been used. Putting it another way: without language support, such encapsulated identifiers are not really encapsulated.
> In early C++ the solution was one inherited from C: declare identifiers as static to give them internal linkage. However, already this has the drawback of overloading the keyword `static`. Also, as the language evolved and templates were added, it became apparent there was another drawback: identifiers with internal linkage could not be template arguments.
> To resolve the above issues, a more C++-centric solution was devised - the unnamed namespace, or as it tends to be called in more common parlance, the "anonymous namespace". Identifiers declared in the anonymous namespace have external linkage (and can be used as template arguments), but are accessible only in the TU in which they are declared. The mechanism by which this is achieved is implementation dependent, but a popular approach is the use of a scheme where the compiler mangles the identifier name with that of the TU.

The descriptions in the C++ standard and The C++ Programming Language, together with the comments from Mark Radford cover the importance and uses of anonymous namespaces well. Not only do they prevent name clashes, they also provide context. The reader of a source file (`.cpp`) knows that anything located within the anonymous namespace is only intended for use within that translation unit and no other.

Namespaces Provide Context

Namespaces do not only provide solutions to the problems associated with linking C++ programs. They can also provide context that helps a developer

determine a class's, a function's or a variable's position and purpose within a program just from looking at a single source file.

Before describing the mechanics of namespaces in The C++ Programming language, Stroustrup has the following to say about them:

> A namespace is a mechanism for logical grouping. That is, if some declarations logically belong together according to some criteria, they can be put in a common namespace to express the fact….

So, namespaces are also about grouping related elements of a program together. The file splitter example above can be expanded to demonstrate this. Suppose your program not only splits files, but can also merge them.

Again, the standard C++ processing code should be wrapped in a façade class (`FileMerger`) and should be separate from the COM class. The processing class processes files and therefore belongs in the `Process` namespace along with `Process::FileSplitter`.

```
// filesplitter.h

namespace Process {
  class FileSplitter {
    ...
  };
}

// filemerger.h

namespace Process  {
  class FileMerger {
    ...
  };
}
```

The `FileMerger` COM class, of course, goes into the `ATL` namespace with `ATL::FileSplitter`.

Elements that are grouped together by a namespace share a context. Equally, when you look at a single class, function or variable declaration you know what context it is in from its namespace.

For example, you could open any source or header file from the example above and be looking at a `FileSplitter` or `FileMerger` class and know immediately whether it was a file processing class or a COM class, just from its namespace. This is a significant maintenance advantage as you would not have to go searching through other source and header files trying to determine the context of the file you had just opened.

There are, of course, other ways of providing this context. Some, such as directory structure, complement the use of namespaces very well, but is a subject beyond the scope of this article.

Appending `File` to the front of `FileSplitter` and `FileMerger` suggests that there can be other types of splitters and mergers within the context of the program. Otherwise, they may as well just be called `Splitter` and `Merger`. In the example presented so far that would be perfectly reasonable.

However, now consider that as well as splitting complete files, record by record, the records themselves are split in some way. The logical name for a class that splits a record is `RecordSplitter`. This introduces a new context and should really introduce a new namespace:

```
// recordsplitter.h
namespace Process {
  namespace Record {
    class Splitter {
      ...
    };
  }
}
```

If a record merging class is introduced into the program that too would go into the `Record` namespace. The file processes should also be placed in a nested namespace:

```
// filesplitter.h

namespace Process {
  namespace File {
    class Splitter {
      ...
    };
  }
}

// filemerger.h

namespace Process {
  namespace File {
    class Merger {
      ...
    };
  }
}
```

This technique can of course be taken too far and is probably overkill for this example, but I hope it shows the concept of namespaces providing context.

What about anonymous namespaces? Do they provide context too? Absolutely! Anonymous namespaces provide context within a translation unit. As stated above, they tell you that the contents of the anonymous namespace are only intended for use in the current translation unit.

Consider the following example. You have a lookup table of postcodes that are to be loaded from a database:

```
// Postcode.cpp

#include "lookup\postcode.h"

namespace PostcodeTools {
  namespace {
    const std::string postcodeSql
        = "SELECT postcode FROM postcodes";
  }

  void Postcode::Load() {
    dbConn_->Execute( ... );
    ...
  }

  ...
}
```

There are a number of things that can be done with the `postcodeSql` string. It could be a local variable inside the `load` function, but it may be something that changes if the database table moves or is renamed for some reason. Therefore it should be as prominent as possible to make finding it easy. This would suggest it should be brought out to namespace scope so that it is near the top of the file. This opens up the possibility of a name clash (although const variables actually have internal linkage) as other translation units containing the namespace `PostcodeTools`, could also have a `postcodeSql` member also.

The obvious solution is to place `postcodeSql` in an anonymous namespace as shown. Even though members of an anonymous namespace have external linkage, they cannot clash with names declared in other translation units. The anonymous namespace also tells you that the `postcodeSql` string is only intended for use within the source file.

In this article I have examined the use of namespaces and anonymous namespaces and the context provided by them. I hope I have made a good case for their usage and that I have encouraged readers to use namespaces more widely and for context as well as for preventing name clashes.

Paul Grenyer

References

[Façade] Alan Shalloway, James J. Trott, *Design Patterns Explained: A New Perspective on Object-oriented Design*, ISBN: 0201715945

[TCPPL] Bjarne Stroustrup, *The C++ Programming Language, Special Edition*, ISBN: 0201700735

Acknowledgments

Thank you to Adrian Fagg, Mark Radford, Phil Bass and Alan Griffiths.

An Introduction to Objective-C

Part 3 – An Example Using Foundation

D.A. Thomas

The best way to get the feel of a programming language is to have a look at actual code. This demonstration program I've written consists of three Objective-C source files:

Listing 1 shows `main.m`, which reads file names from the command line and prints the unique lexical tokens found in each file. Tokens are strings of printable characters separated by whitespace and punctuation marks.

Listing 2 shows `StringTokenizer.h`, which declares the public interface of the class `StringTokenizer`. Private methods, being part of the implementation, typically have no place in this file.

Listing 3 shows `StringTokenizer.m`, which contains the implementation of the `StringTokenizer` class.

Listing 1: `main.m`

```
#import <Foundation/Foundation.h>
#import "StringTokenizer.h"

// Prints unique tokens found in files
// supplied as arguments.

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

  // Create a pool of items to be garbage-collected
  NSAutoreleasePool *pool
                = [[NSAutoreleasePool alloc] init];
  if(argc > 1) {
    int i;
    for(i = 1; i < argc; ++i) {
      // Read contents of file into string
      NSString *path
             = [NSString stringWithCString:argv[i]];
      NSString *myString
         = [NSString stringWithContentsOfFile:path];
      if(myString == nil) {
        fprintf(stderr, "File %s not found\n",
                [path cString]);
        // The system will clean up anyway when we
        // exit,  but we do this for form's sake
        [pool release];
        return 1;
      }

      // Create our tokenizer
      StringTokenizer *tokenizer
            = [[[StringTokenizer alloc]
                initWithString:myString
                andDelimiters:@" ,.!?;:\t\r\n"]
               autorelease];

      // Create a set with room for 100 items to
      // hold unique tokens
      NSMutableSet *theSet
               = [NSMutableSet setWithCapacity:100];

      // Get the first token
      NSString *token = [tokenizer nextToken];
      while(token != nil) {
        // This will fail if there is an identical
        // token there already
        [theSet addObject:token];
        // Get more tokens
        token = [tokenizer nextToken];
      }

      // Print out unique tokens in the set in case-
      // insensitive alphabetical order
      NSArray *tokens = [[theSet allObjects]
                sortedArrayUsingSelector:@selector(
                caseInsensitiveCompare:)];

      printf("Unique tokens in %s:\n",
             [path cString]);

      int j;
      for(j = 0; j < [theSet count]; ++j)
        printf("\t%d %s\n", j+1,
               [[tokens objectAtIndex:j] cString]);
    }
  }
  else
    fprintf(stderr,
      "Usage: StringTokenizer file1 file2 ...\n");

  // Trigger autorelease of allocated memory
  [pool release];
  return 0;
}
```

Listing 2: `StringTokenizer.h`

```
// Minimal tokenizer class, useful for
// demonstration purposes only

#import <Foundation/Foundation.h>

@interface StringTokenizer : NSObject {
  NSString *data;
  NSCharacterSet *delimiters;
  size_t position, dataSize;
}

// Default initializer - object contains no data
// and delimiters string set to space, tab, newline
// and return
- (id)init;

// Initialise object with data string;
// delimiters string is set to space, tab
// newline and return.
- (id)initWithString:(const NSString *)aString;

// Initialise object with data string to tokenise
// and a set of delimiters to ignore
- (id)initWithString:(const NSString *)aString
andDelimiters:(NSString *)delims;

// Assign the object a new data string to
// tokenise
- (void)setData:(const NSString *)aString;

// Assign the object a new set of delimiters to work
// with
- (void)setDelimiters:(NSString *)delims;

// Return the next token from the data string or nil
// if none exists
- (NSString *)nextToken;

@end
```

Listing 3: `StringTokenizer.m`

```
#import "StringTokenizer.h"

// Default delimiters are whitespace
#define DEFAULT_DELIMITERS @" \t\n\r"
```

```
// Create a category to forward-declare
// private method in order to avoid
// compiler warnings about undeclared methods.
@interface StringTokenizer (Private)
- (void)skipDelimiters;
@end

@implementation StringTokenizer

- (id)init
{
  return [self initWithString:nil
          andDelimiters:DEFAULT_DELIMITERS];
}

- (id)initWithString:(const NSString *)aString
{
  return [self initWithString:aString
          andDelimiters:DEFAULT_DELIMITERS];
}

// This is the designated initialiser, which
// is called by all the other initialisers and
// does all the work
- (id)initWithString:(const NSString *)aString
       andDelimiters:(NSString *)delims
{
  if(self = [super init]) {
    position = 0;
    data = [aString retain];

    // Cache length of data string
    dataSize = [data length];
    delimiters = [[NSCharacterSet
          characterSetWithCharactersInString:delims]
          retain];
  }
  return self;
}

- (void)setData:(const NSString *)aString
{
  if(aString != data) {
    [data release];
    position = 0; // We are starting from scratch
    data = [aString retain];
    dataSize = [data length];
  }
}

- (void)setDelimiters:(NSString *)delims
{
  [delimiters release];
  delimiters = [[NSCharacterSet
        characterSetWithCharactersInString:delims]
        retain];
}

- (NSString *)nextToken
{
  if(data == nil || position >= dataSize)
    return nil;

  [self skipDelimiters];
  if(position >= dataSize)
    return nil;

  size_t oldPosition = position;
                     // Save current position

  BOOL nonDelim = YES;
                  // Assume that the next
                  // character is a non-delimiter
  while(position < dataSize && nonDelim) {
    // Test for a match in the delimiter string of
    // the character at the current position in the
    // data string; if no match is found, increment
    // position and proceed.
    if(![delimiters characterIsMember:
         [data characterAtIndex:position]])
      position++;
    else
      nonDelim = NO;
  }

  // Create a string containing the token and return
  // it. Type NSRange is a struct containing two
  // members: location and length
  NSRange range = {oldPosition, position-oldPosition};
  return [data substringWithRange:range];
}

- (void)skipDelimiters
{
  BOOL nonDelim = NO;
         // Non-delimiter character not yet found

  while (position < dataSize && !nonDelim) {
    // Test for a match in the delimter string of
    // the character at the current position in the
    // data string; if a match is found, increment
    // position and proceed.
    if([delimiters characterIsMember:
        [data characterAtIndex:position]])
      position++;
    else
      nonDelim = YES;
  }
}

// Invoked automatically when object is released
- (void)dealloc
{
  // Release memory allocated for our instance
  // variables
  [data release];
  [delimiters release];
  [super dealloc];
}

@end
```

Notes

The preprocessor directive `#import` is generally used in Objective-C; it differs from `#include` in that it ensures that a header file is included only once even if it does not contain guard macros.

Foundation's memory management involves semi-automated reference counting. Pointers to all the objects allocated in the main function will be added to the autorelease pool, and when this pool is released, the method `-dealloc` is called on all the objects before the memory they occupy is freed.

Memory allocation for objects is usually provided by the method `+alloc` in the root class, `NSObject`. `-retain` increases the reference count by one, while `-release` decrements it. `-autorelease` adds the receiver to the autorelease pool.

`NSString` is Foundation's basic string-handling class. `NSString` objects hold Unicode strings that cannot be changed once created; objects of its subclass `NSMutableString` allow their contents to be edited.

`StringTokenizer` is a class I have written to extract tokens from a string. It has limited functionality but is sufficient for the purposes of this demo. Tokens are extracted by calling `nextToken` repeatedly until `nil` is returned. An instance of `StringTokenizer` is created by calling the class method `alloc` to allocate storage for the object and

[concluded on next page]

Automatically-Generated Nightmares

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

A student sent me the source code for a mini-project of his. It came to about 100 printed pages of Java code, and he had a problem with it. I started looking through the code, but it was very difficult: there were hardly any comments, and most variable names implied that the code was something to do with a graphical interface but it was hard to see exactly what was going on, and anyway the problem in question had nothing to do with the interface. I searched in vain for the part of the code that actually did something other than manipulating interface objects, and wondered how on earth he can write all of this without getting completely muddled. I thought "am I such a lousy programmer that I can only deal with small and manageable code while the new students can write reams and reams like this?" and "how many hours a week is a programmer expected to work these days to come up with all of this?" and "perhaps I'd better admit my utter lack of productivity now and find a different job". When I finally figured out roughly how it all worked, I asked the student a question about part of the code that I thought was more suspect than the rest, and his reply was "Oh, I don't know about that; the GUI wizard program wrote it all for me." Ah! So THAT'S why it was so big and complicated. I shouldn't have worried so much. Or should I?

There are plenty of "GUI wizards" and other tools that will generate code for you, and that code will usually contain "TODO" comments to show where you should add your own logic.

Unfortunately it seems that most programmers, after writing what they are prompted to write, delete the "TODO" comment and do not add another in its place, which means that anyone else who wants to browse or debug the code will first have to spend considerable time unravelling the automatically generated code to try and figure out where the user-written code is actually located. This must add up to an awful lot of wasted programmer time in the industry. While some tools give you code that is relatively easy to follow when finished, others generate such large amounts of hard-coded graphical widget handling that you'd be pushed to find anything else.

I can understand why programmers want to delete "TODO" comments, especially in the light of tools that flag them up as "things not done", but I think it would be better if, instead of deleting the whole comment, they simply delete the word "TODO" when they've done it. That would leave a comment that gives some description of what is happening and also serves to highlight where the user-written code is to be found; as the automatically generated parts usually have few if any comments, any comments at all would make your code stand out. Even a complete beginner who is not skilled at writing comments can adopt this method, and it would make things a great deal easier for anyone who has to check their work.

What would be even nicer is if more of the wizards and other rapid application development tools could promote a clearer separation of concerns between your code and their code. Object-oriented languages like Java support this naturally (think of encapsulation and all the other buzzwords you know) but it's not being used as well as it could be. Why do the tools promote obfuscation, turning Java into a language of a much lower level than it was designed to be?

Perhaps it is because they want you to spend a lot of money on other tools to help you maintain the code. In effect, they are creating a language of their own which gets compiled into Java and then has to be de-compiled by their tools before it makes sense. Java is going the same way as HTML: it was originally intended to be written and read by humans, but now most of it is generated automatically and can't easily be made sense of without highly complex tools (and even then you're not guaranteed to see the programmer's intentions). To see what I mean, take a straightforward text editor and try to make sense of the source code in an average Eclipse plugin. (Yes I know some of them are better than others.)

The problem is that the tools are not always available. What if you want to review some code when you don't happen to be sitting at your most powerful computer? When you're not at any computer? When the tool doesn't co-operate with your special disability access software, or its layout is too complex for the size of the display you're using? When there are license restrictions that get in the way of your using it in the circumstances at hand? When you don't even know what tool you're supposed to be using because someone has handed you the code without telling you? It's understandable that you need the right tools when compiling or testing code, but it's rather more restrictive if you have to arrange your life around them just to look.

Moreover, what if the way that you think best when you review code does not tie in with the way the tool pushes you into thinking? Sometimes tool designers push you into a particular way of thinking for a good reason, but at other times they're just being short-sighted. It reminds me of Green's Cognitive Dimensions of Notations theory; ordinary text editors and printouts, when applied to code that is designed for them, tend to be good at allowing "viability" of the notation, avoiding the need for "premature commitment", and so on, whereas other tools don't always do so well. While they may make some developers more effective, others become less effective, and that's bad news when you're collaborating.

If you are a user of a code-generation tool, perhaps it would be a good idea if you put your own code in a completely different file, and get the automatically-generated stubs to call it. This should make it easier when the code needs reviewing, or when you want to reuse parts of it in other projects. It also makes it easier if for some reason you want to re-run the tool, or to run some other tool (perhaps for a newer kind of interface) without throwing away all of your work. If you can't do such separation, then at least keep clear comments about which parts of the code were written by hand, otherwise you may be in trouble later.

Unfortunately, for the project in question, we ended up starting over, because I thought that this would be less effort than trying to figure things out from the forest of automatically-generated code that was camouflaging the parts of interest. It shouldn't have been that way.

Silas S Brown

[continued from previous page]

then the object is initialised by the instance method `initWithString:andDelimiter:`, which does the same kind of work as a constructor in C++ and Java.

The Objective-C keyword `nil` refers to a null object. It differs from the macro `NULL` in that it is perfectly legal and safe to send messages to `nil`.

`NSMutableSet` is a subclass of `NSSet`, from which it differs by allowing objects to be inserted and deleted after it has been initialised. Instances of `NSSet` and `NSMutableSet` are unordered collections of values, where each value occurs at most once.

Objects of class `NSArray` are immutable ordered collections of objects. The line:

```
NSArray *tokens = [[theSet allObjects]
                sortedArrayUsingSelector:
                @selector(caseInsensitiveCompare:)];
```

deserves some comment. First the message `allObjects` sent to the set causes it to return an `NSArray` of its contents in arbitrary order; this `NSArray` object then receives the message `sortedArrayUsingSelector:` with the selector of `NSString`'s `caseInsensitiveCompare:` method as argument. (The compiler directive `@selector` turns a method name into a selector.) The `NSArray` object's method `sortedArrayUsingSelector` then returns a new copy of itself with the `NSString` objects in case-insenstitive ascending collating order.

The `StringTokenizer` class contains an instance variable called `delimiters` of type `NSCharacterSet`. Its method, `characterIsMember:` is called for each character of the string in turn; if the character is found in the `delimiters` character set, it is skipped, and any non-delimiter or unbroken sequence thereof is recognised as a token and returned.

D. A. Thomas

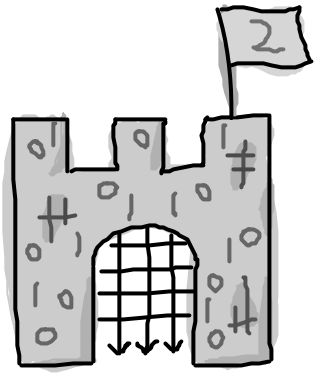
Professionalism in Programming #29

An Insecurity Complex (Part Two)

Pete Goodliffe <pete@cthree.org>

The more you seek security, the less of it you have.

Brian Tracy



Last time we opened an ugly can of worms by investigating the seedy world of software security. We learnt the nature of security problems and discovered why it's depressingly hard to secure our code. This article concludes our tour by investigating specific code vulnerabilities and working out how to prevent them in the programs we write.

Feeling Vulnerable

To learn how to write secure code and defeat our adversaries let's look at the security nuts-and-bolts. These are some specific types of code vulnerability. Each is a hole that can be compromised by an attacker.

Insecure Design

This is the most fundamental flaw, and consequently the hardest to fix. If you don't consider security at the architectural level then you will be committing security sins everywhere: sending unencrypted data over public networks, storing it on easily accessible media, and running software services that have known security flaws.

You could write a simple system, and rely on your host environment for security, but then your application will only be as secure as that system. For example, a Java program can be no more secure than the JVM it's running on.

Absolutely every system component must be considered for security concerns. A computer system is only as safe as its least secure part.

Buffer Overrun

Many applications are public-facing, running an open network port or handling input from a web browser or GUI interface. All of this input must be parsed and acted on. If you're not careful, these are prime sites for security failure.

Parsing is often done using the standard C library function `sscanf` (although this exploit is far from a C-only problem). You might see code like this:

```
void parse_user_input(const char *input) {
  /* first parse the input string */
  int  my_number;
  char my_string[100];
  sscanf(input, "%d %s", &number, my_string);
  ... now use it ...
}
```

The problem is simple (and obvious). A badly formed input string could cause mayhem. Any string over 100 characters long will overrun the `my_string` buffer, and smear arbitrary data across invalid memory addresses.

The results of this can vary in severity. Sometimes the program will carry on unaffected; you've been very, very lucky. Sometimes the program continues, but its behaviour is subtly altered – this can be hard to spot and confusing to debug. Sometimes the program will crash as a consequence, perhaps taking other critical system components down with it. But the worst case is when the spilt data gets written somewhere in the CPU's execution path. This isn't actually hard to do, and allows an attacker to execute arbitrary code on your machine, potentially even gaining complete access to it.

More Terms

There are a few important pieces of security terminology. Understanding them will help us to reason about security problems.

Flaw A flaw is an unintended problem in an application. It is a program bug. Not all flaws are security problems.

Vulnerability A vulnerability exists when a flaw opens the possibility for a program to be insecure.

Exploit This is an automated tool (or a manual method) that employs a program vulnerability to force unintended – and insecure – behaviour.

Overrun is easiest to exploit when the buffer is located on the execution stack, as in the example above. Here it's possible to direct CPU behaviour by overwriting the stack-stored return address of a function call. However, buffer overrun exploits can abuse heap-based buffers too.

Embedded Query Strings

This breed of attack can be used to crash systems, execute arbitrary code, or fish for unauthorised data. Like buffer overrun it relies on a failure to parse input, but rather than burst buffer boundaries these attacks exploit what the program subsequently does with the unfiltered input.

In C programs *format string attacks* are a common example of the problem. A great culprit is the `printf` function (and its variants), being used as follows:

```
void parse_user_input(const char *input) {
  printf(input);
}
```

The input string is used as `printf`'s format string parameter, and a malicious user could provide an input string containing format tokens (like `%s` and `%x` for example). This can be used to print data from the stack or even from locations in memory, depending on the exact form of the `printf` call. An attacker can also write arbitrary data to memory locations using a similar ploy (exploiting the `%n` format token).

Solutions to this problem aren't hard to find. Simply writing `printf("%s", input)` instead of `printf(input)` will avoid the problem, by ensuring that `input` is not interpreted as a format string.

There are many other contexts where an embedded query can be inserted maliciously into program input. SQL database query statements can be surreptitiously fed into database applications to force them to perform arbitrary database lookups for an attacker.

Another variant is commonly exhibited by lax web-based applications. Consider an online bulletin board system providing forums where users post messages to be read by any other web browser. If an attacker posts a comment containing hidden Javascript code, this will be executed by all browsers rendering the page – without their users realising. This is known as a cross site scripting exploit, due to the way the attack works 'across' the system; from an attacker's input, through the web application, finally manifesting on a victim's browser.

Race Conditions

It is possible to exploit systems which rely on the subtle ordering of input events, to provoke unintended behaviour or crash the code. This is generally exhibited in systems with complex threading models, or which comprise of many collaborating processes.

A threaded program might share its memory pool between two worker threads. Without adequate guarding, one thread might read information in the buffer that the writer thread did not intend to release yet.

This problem isn't restricted to threaded applications, though. Consider the following fragment of Unix C code. It intends to dump some output to a file, and then change file permissions on it.

```
fd = open("filename");
/* point A */
write(fd, some_data, some_data_size);
close(fd);
chmod("filename", 0777);
```

There is a race here that at attacker can exploit. By removing the file at point A and replacing it with a link to their own file the attacker gains a specially privileged file. This can be used to further exploit the system.

Integer Overflow

Careless use of mathematical constructs can cause a program to cede control in unusual ways. Integer overflow will occur when a variable type is too small to represent the result of an arithmetic operation. An unsigned 8 bit data type renders this C calculation erroneous:

```
uint8_t a = 254 + 2;
```

The contents of `a` will be 0, not the 256 you'd expect; 8 bits can only count up to 255. An attacker can supply very large numeric input values to provoke overflow and generate unintended program results. It's not hard to see this causing significant problems; the following C code contains a heap overrun waiting to happen thanks to integer overflow:

```
void parse_user_input(const char* input) {
  uint8_t length = strlen(input) + 11;
  char *copy = malloc(length);
  if(copy) {
    sprintf(copy, "Input is: %s", input);
    ... do something with copy ...
  }
}
```

It's true that `uint8_t` is an unlikely candidate for the string length variable, but the exact same problem manifests itself with larger data types.

This kind of problem is just as likely with subtraction operations (where it's called integer *underflow*). It's not only generated by such simple operations, and can stem from mixed signed/unsigned assignments, bad type casting, and multiplication or division.

Protection Racket

So what techniques will protect us from this mayhem? We'll start to answer this with a simple analogy from the Real World. If you were to secure a building there's a number of things you'd do:

- Close all the unnecessary entrances, brick up the back door, and board over the windows.
- Obscure the remaining windows so people can't easily see what's inside.
- Secure the entry points. Lock all doors, hide the keys, and make sure you use very good locks.
- Employ a guard to patrol inside and out.
- Add security mechanisms, like a burglar alarm, electronic pass cards, identity badges, etc. There's no point in installing these if they're not used properly, though. A door can be left ajar regardless of any fancy lock devices. A burglar alarm can be left unset.
- Put all your valuables in a safe.

In summary, you would cut down on the possible attack points and employ technology that deters, blocks, identifies, and repels attackers. These have many software-writing analogues which we'll investigate below. They can each be applied at a number of different development levels, including:

- On a particular system *installation*. The exact OS configuration, network infrastructure, and the version number of all running applications each have radical security implications.
- The software system *design*. We need to address design issues like: can the user remain 'logged in' for indefinite periods, how does each subsystem communicate, and what protocols are used?
- The actual program *implementation*; it must be flaw-free. Buggy code leads to security vulnerabilities.
- The system's usage *procedure*. If it's routinely used incorrectly, any software system can be compromised. We should design to prevent this as much as possible, but users must be taught not to cause problems. How many people write down their username/password on paper beside their terminals?

Creating a secure system is never easy. It will always require a security/functionality compromise. The more secure a system is, the less useful it becomes. The safest system has no inputs and no outputs; there's nowhere for anyone to attack. It won't do much, though. The easiest system has no authentication, and allows everyone full access to everything; it's just terribly insecure. We need to pick a balance. This depends on the nature of the application, its sensitivity, and the perceived threat of attack. To write appropriately secure code we must be very clear about such *security requirements*.

Just as you would take steps to secure a building, the following techniques will protect your software from malicious attackers.

System Installation Techniques

First we'll look at practices that will protect your software once it's been installed. Perhaps this is backwards, but it will highlight what holes remain to be plugged at a lower level. No matter how good your application, if the target system is insecure then your program is unprotected.

- Don't run any untrusted, potentially insecure software on your computer system.

  This raises the question: what makes you trust any piece of software? You can audit open source software to prove that it's correct (if you have the inclination). You can opt for the same software that everyone else uses, thinking that there's safety in numbers. However, if a vulnerability is found in that software you, and many other people, must all update. Or you can pick a supplier based on their reputation, hoping that it's a worthwhile indicator.
- Employ security technologies, like firewalls and spam/virus filters. Don't let crackers in through a back door.
- Prepare for malicious authorised users by logging every operation, recording who did what and when. Backup all data stores periodically so that bogus modifications don't lose all of your good work.
- Minimise the access routes into the system, give each user a minimal set of permissions, and reduce the pool of users if you can.
- Set up the system correctly. Certain OSes default to very lax security, just inviting a cracker to walk straight in. If you're setting up such a system then it's vital to learn how to protect it fully.
- Install a *honeypot*: a decoy machine that attackers will find more easily than your real systems. If it looks plausible enough then they'll waste their energy breaking into it, whilst your critical machines continue unaffected. Hopefully you'll notice a compromise of the honeypot and repel the attacker long before they get near your valuable data.

Software Design Techniques

As programmers this is the essential place to get our security story straight. You can try to shoehorn it into code at the end of a development cycle, and you'll fail. Security must be a fundamental part of your system's architecture and design.

So what design techniques will improve our software security? The simplest software design is the easiest to secure. So don't run any software at all. Failing that, run your program in a sealed box in an underground bunker in an undisclosed location in the middle of a desert. That way, crackers can't get anywhere near it. Otherwise you'll have to think about how your software will be used, and how to actively prevent anyone from abusing it. Here are the winning strategies:

- Limit access to the system as much as possible. The hardest kind of access to guard against is physical access to the computer itself; how can you stop an attacker switching it off, or installing their own evil software? Physical access notwithstanding, design your software to block as many entry points as possible.
- Limit inputs in your design so that all communication goes through only one portion of system. This way an attacker can't get all over your code. Their influence is limited to a secluded corner, and you can focus your security efforts there[1].
- Run every program at the most restrictive privilege level possible. Don't run a program as the system superuser unless it's absolutely necessary, and then take *even more* care than usual. This is especially important for Unix programs that run `setuid` – these can be run by any user, but are given special system privileges when they start.
- Avoid any features that you don't really need. Not only will it save you development time, it will reduce the chance of bugs getting into the program – there's less software for them to inhabit. In general, the less complicated your code, the less likely it is to be insecure.
- Don't rely on insecure libraries. An insecure library is anything you don't *know* to be secure. For example, most GUI libraries aren't designed forsecurity, so don't use them in a program run as the superuser.
- Avoid storing sensitive data. If you must, obscure or encrypt it. When you handle secrets be very wary where you put them; lock memory pages containing sensitive information so that your OS's virtual memory manager can't 'swap' it onto the hard disk, leaving it available for an attacker to read.
- Obtain secrets from the user carefully. Don't display passwords.
- Specify good locks. That is, use tightly controlled password access and employ strong encryption to store data.

The least impressive security strategy is known as *security through obscurity*, yet this is really the most prevalent. It merely hides all software design and implementation behind a wall, so that no one can see how the code works and figure out how to abuse it. 'Obscurity' means that you don't advertise your critical computer systems in the hope that no attacker will find them.

[concluded at foot of next page]

1 Of course, it's never quite *that* simple. A buffer overrun could occur anywhere in your code, and you must be constantly vigilant. However, most security vulnerabilities exist at, or near, the sites of program input.

Reviews

Bookcase

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

A Note from Francis

While we are very happy to have reviews submitted for books that you have bought please make sure that you include all the relevant information with your review (i.e. all the information provided with our reviews below apart from the £/$ ratio.)

In addition if you want to ask a publisher for a review copy on behalf of ACCU you must go through the correct process, that includes asking the book review editor. Publishers get unhappy when they are asked for a review copy when their records show they already provided one. They often recognise the names of our more prolific reviewers and assume that if one of those asks for a review copy they have been authorised to do so. I do not want our excellent relationship with book publishers damaged by thoughtlessness. They accept our, sometimes caustic, reviews and in return we should stick to the process. Just drop me an email and usually the answer will be to go ahead.

Prices

While I was in Redmond I stocked up with a couple of dozen Science Fiction books, not just because some of them were not yet available in the UK but because I was paying the same number of dollars that I would expect to pay in pounds this side of the Atlantic. That is just to put a little perspective on relative book prices.

However I think US readers might have a good reason to grumble about a book price that converted to more than two dollars to the pound (Wiley accomplished that this time with a conversion of 2.13) while Addison-Wesley managed the worst rate the other way (with a conversion of 1.03). That these are two of the best technical publishers around is no excuse for such terrible price comparisons across the Atlantic.

Prize Draw

(extended time – now closes midnight December 31st/January 1st)

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 December 31st/January 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.

It's a flawed plan. Your system *will* one day be found, and *will* one day be attacked.

It's not always a conscious decision, and this technique works very conveniently when you forget to consider security in the system design at all. That is, it's convenient until someone does compromise your system. Then it's a different matter.

Code Implementation Techniques

With a bullet-proof system design your software is unbreakable, right? Sadly not. We've already seen how security exploits can capitalise on flaws in code to wreak their particular brand of chaos.

Our code is the front line, the most common route an attacker will try to enter through, and the place our battles are fought. Without a good system design even the best code is unprotectable, but under the shadow of a well thought out architecture we must build strong walls of defense with robust code. Correct code is not necessarily secure code.

- Defensive programming is the main technique to achieve sound code. Its central tenet – *assume nothing* – is exactly what secure programming is about. Paranoia is a virtue, and you can never assume that the user will employ your program as you expect or intend.
  Simple defensive rules like: 'check *every* input' (including user input, startup commands, and environment variables), and 'validate *every* calculation' will remove countless security vulnerabilities from your code.
- Perform *security audits*. These are careful reviews of the source code by security experts. Normal testing won't find many security flaws; they are generally caused by bizarre combinations of use that ordinary testers wouldn't think of, for example very long input sequences which provoke buffer overrun.
- Spawn child processes very carefully. If an attacker can redirect the sub-task then they can gain control of arbitrary facilities. Don't use C's `system` function unless there's no other solution.
- Test and debug mercilessly. Squash bugs as rigorously as you can. Don't write code that can crash; its use could bring down a running system instantly.
- Wrap all operations in atomic transactions so an attacker can't exploit race conditions to their advantage. You could fix the earlier `chmod` example by using `fchmod` on the open file handle, rather than `chmod`ing the file by name – it doesn't matter if the attacker replaces the file, you know exactly what file is being altered.

Procedural Techniques

This is largely a matter of training and education, although it helps to select users who aren't totally inept, if you have that luxury.

Users must be taught safe working practices: to not tell anyone their password, to not install random software on a critical PC, and to use their systems only as prescribed. However, even the most diligent people will make mistakes. We design to minimise the risk of these mistakes, and hope that the consequences aren't ever too severe.

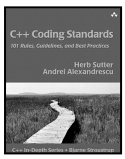
Conclusion

Programming is war.

Security is a real issue in modern software development; you can't stick your head in the sand and hide from it. Ostriches write poor code. We can prevent most security breaches by better design, better system architecture, and greater awareness of the problems. The benefits of a secure system are compelling, since the risks are so serious.

Pete Goodliffe

C & C++



C++ Coding Standards by Sutter & Alexandrescu (0-321-11358-6), Addison-Wesley, 220pp @ £26.99 (1.30)
reviewed by Francis Glassborow

Herb Sutter gave me a manuscript copy of this book when I was in Redmond recently. I thought readers might like the benefit of a fast review from me before someone else does a more in depth review for a later issue of C Vu. (Note that at the time of writing the book is only on distribution in North America but by the time you read this it should be available in Europe – check elsewhere in this issue of C Vu because I hope to arrange some form of special Christmas deal including this book with Blackwells.)

Before I go any further, I should make it clear that this is a book about best practices for programming in C++. I find that some people do not clearly distinguish between 'language standards' (nothing to do with this book) and 'programming guidelines' (essentially what this book presents the basis for).

The book starts with some general (largely non-code specific) guidelines. The first of these can be summarised as exhorting the reader to not waste time on such minor issues as code layout, just be consistent (with the style used by the file you are maintaining, the rest of your code or with the team you are a member of). I suspect that there are readers who will get very heated about example 4 in guideline 0. This concerns SESE (single entry, single exit). Now I happen to agree with the authors that requiring SESE in a coding standard is archaic and no longer appropriate, however there are well-respected members of the C++ community who would very strongly disagree. The solution is to accept the basic spirit of the guideline and not get over-heated if your instructor insists your code should be SESE.

The reason that I took time to call out the above example is that it is a particular example of a general objective of the authors; this is not a book telling the reader what they must do, it is a book setting out some general principles. If you understand the intent of the book as a whole and the individual guidelines in particular you will be able to adopt and adapt to your own needs and environment.

Despite there being two authors, both highly competent, who have done their research with care and attention to detail there is, inevitably, a degree of subjectivity in their 101 'rules'.

The individual guidelines vary from very simple, very specific ones such as 'Avoid magic numbers' (#17) to rather more general ones such as 'Design and write error-safe code' (#71).

I think that #17 does not go far enough in either explaining when a numerical constant is a magic number. There is a difference between an arbitrary limit (such as the maximum size for an array) and a mathematical constant such as π. Both deserve to be named but for rather different reasons. On the other hand there are simple integer values (such as the number of feet in a yard) that do not deserve to be named if you are only using them in a context where the meaning is clear (`length_in_feet = 3 * length_in_yards`).

I also believe that this guideline would have been more powerful had it been 'Avoid magic' and then gone on to explain that using named (inline) functions avoids overly complicated 'magic' expressions and that a `typedef` can giving more meaning to the type you are reusing.

Almost every one of the 101 items provides a basis for discussion. That is as it should be. I would offer a meta-guideline 'Do not slavishly follow guidelines'. I will leave it to another place and time to expand on that.

Many of the items in this book are already known to experienced programmers but they deserve their place because this book should be one that is read and digested by everyone from the aspiring novice to the long-term expert.

Now, I wonder if the authors have started on 'More C++ Coding Standards'. If they haven't, they should because there is still much more wisdom that deserves encapsulation.

An Introduction to GCC by Brian Gough (0-9541617-9-3), Network Theory Limited., 116pp @ £12-95 (1.54)
2nd review by Ian Bruntlett

This book is a good introduction to GNU C/C++ let down by three serious omissions: a) it does not show you how to use **gdb** to debug programs; b) it overlooks **make**, a critical tool for non-trivial applications; c) it omits to mention the **-Weffc++** compiler option that warns about violations of the style guidelines from Scott Meyer's Effective C++ book.

On the other hand, it does provide information about useful system utilities:

- `file` – list details about an executable;
- `nm` – list an executable/object file's symbol table / name table;
- `ldd` – list an executable/object file's dynamically linked libraries;
- `gcov` – GNU coverage testing tool; and
- `gprof` – GNU profiler

Verdict: Print a copy for yourself – I would not buy it until the next edition appears hopefully covering the omitted topics. (GNU Free documentation downloadable from `www.network-theory.co.uk/gcc/intro`.)

The Definitive Guide to GCC by Wall & Hagen (1-59059-109-7), Apress, 500pp @ £35-50 (1.41)
reviewed by Ian Bruntlett

The big question that I intend to address is: why buy this book instead of relying on the free book "An introduction to GCC for the GNU compilers gcc and g++?"

Well, this book does cover more ground than its rival – sadly, like its competitor it fails to document the GNU debugger (**gdb**) or the **-Weffc++** option or even **make** – it does cover automake which makes the make oversight surprising. However, in this book's favour it is unique in covering:

- How to build GCC from source.
- Using autocong and automake.
- Using libtool.
- Trouble shooting GCC (including build & installation problems).
- GCC online help (GNU info) - although it fails to mention that info is obsolete if you're using KDE - just open a Konqueror window and type in ##gcc.

Verdict: If you want a comprehensive reference, buy this book. Otherwise buy or print "An introduction to GCC". Recommended.

C# & Java

Learning Java 2ed by Patrick Niemeyer & Jonathan Knudsen (0 596 00285 8), O'Reilly, 807pp + CD @ £31-95 (1.41)
reviewed by Ivan Uemlianin

'Learning Java' is a well-written exploration of Java's features. However, its support for learning the language is poor. Ironically, given the book's history, 'Exploring Java' would have been a better title.

After two chapters introducing Java, chapters 3-8 cover the language basics, chapters 12-14 cover network and Internet programming, and chapters 15-20 cover the Swing GUI toolkit (the Swing chapters take up around a quarter of the book). Other chapters cover various topics like text, i/o, JavaBeans, Applets and XML.

The book is not organised into parts and, apart from the fact that simple topics are generally discussed before complex topics, the course of discussion can feel arbitrary. For example, the XML chapter is the last in the book, 14 chapters away from the text chapter, and 10 chapters away from web programming. This can make progress through the book feel a little haphazard.

It is clear that the reader is assumed to have a programming background, and the basics are covered very tersely (e.g., on p.90, do/while gets, "the do and while iterative statements have the familiar functionality").

There are no exercises. This means that the book cannot be used for self-study, and is of limited use for study with an instructor: instructors will have to use their own ingenuity or find another source for exercises. There are plenty of examples, however, almost all of which work as advertised.

The book is at its best when most discursive, (e.g., chapters 5-7 on Java's style of object-orientation). The authors explain their topics well when they give themselves the space: for example, five pages on internationalisation is not enough space, but I found the section on sockets (15 pages in a chapter on network programming) quite useful.

For its first two editions this book had the title 'Exploring Java'. 'Exploring Java' is still a better title and fits the book better than 'Learning Java'. As a discussion of Java's features, with the reader released from the pressure of having to 'learn', the book works very well. The pitch of the book makes sense: the reader knows a bit of Java and would like a more rounded appreciation. As a book for learners, it is flawed: there are no exercises; the structure is piecemeal rather than progressive; the pitch level is not consistent.

I could perhaps recommend this book for either (a) an experienced programmer who wants to become familiar with the details of Java without bothering too much about the basics (although in this case something more focussed on a problem domain might be preferable); or (b) a learner who has a basic course under their belt and wants to explore the language a bit more.

Technical Java by Grant Palmer (0 13 101815 9), Prentice Hall, 466pp @ £39.99 (1.25)
reviewed by Ivan Uemlianin

Technical Java has two target audiences: the newcomer to Java from a natural science background; the seasoned programmer who is sceptical about using Java in this domain. I recommend the book for either audience.

The book has four parts. The first set of chapters deals briefly with Java's differences with Fortran, C and C++. The second section (chapters 5-16) covers the Java language itself. Then comes the 'technical' section (chapters 17-24) on mathematical modelling. The examples covered in this section come primarily from fluid mechanics (modelling gas mixtures, air flow, etc.), but the techniques are general, and include the Fourier transform. Finally, come three chapters on building the i/o wrappers (simple i/o, GUI and web) that turn models into applications.

Each of the technical chapters has a good project feel. A problem is clearly stated, along with its mathematical representation. In discussing a Java solution, due consideration is given to analysis and design (e.g., class hierarchies, public methods). The code that implements the mathematics is a succinct and 'obvious' response to the problem as defined. These are good examples of how to approach scientific programming.

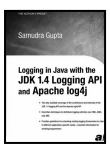
Those who know the language can safely skip the first 16 chapters, but for those new to Java, these chapters provide a workable introduction. In particular, the science and engineering flavour is there right from the beginning: instead of "Hello world!" we have "compute area of circle"; interfaces are illustrated by an "Electrostatic interface ... to be implemented by classes that represent charged bodies".

The language is terse almost to the point of being brusque ("If you want to create a GUI with Fortran you are out of luck"), but very clear. The author favours simple, direct sentences. This makes the text easy to follow. Similarly, it is easy to navigate, as the relevance of any particular section is apparent immediately.

The content seems clear and correct - apart from the peculiar assertion in the C and C++ chapters that Java has no void type (the most common line of code in the book is 'public static void main(String args[]) {'). I was surprised to find no discussion of floating point arithmetic, as this can be relied on to trip up the uninitiated.

This book will be useful to people evaluating languages for a scientific project. Although it cannot answer the question of whether Java is *the* language to use for scientific programming, it certainly makes a good case.

For the right person, this book could also be a good introduction to Java. I imagine someone from a natural science background would feel very at home with it, just as I did.



Logging in Java with the 3DK 1.4 Logging API and Apache by Samudra Gupta (1-59059-099-6), Apress, 324pp @ £32-00 (1.56)
reviewed by Silvia de Beer

This book describes two logging APIs. The first third of the book describes the simpler JDK 1.4 logging API, and the second part the more complex Apache log4j API, which also offers more features and flexibility. I think it is a good choice to cover both APIs in one book, because for small projects, the JDK Logging API might be sufficient. By reading this book, a developer can make a choice whether he needs the more complex Apache log4j, or whether the simple JDK logging API might do the job. The similarities and differences are well explained between the two APIs.

This book is not a reference with a complete interface of the two APIs, and it does not give a reference of the syntax of the configuration files, especially for log4j. However, I did not think that this is such a negative point of the book, because it would have added many pages that might not be very useful. If you want to find out about the details of the APIs, there is no better place than the JavaDoc and possible a few other documents online. The book is well written, and explains correctly the ideas behind the two logging APIs. A few UML class and interaction diagrams are given to explain the interaction between the various classes and interfaces that constitutes the two logging frameworks. The examples do support the text very well and I found this a very pleasant book to read. One thing is maybe not yet stressed enough in this book and that is the enormous value of good configurable logging statements in your application. It is very important that a developer learns to discern the points where a logging statement is required, to be able to correctly trace problems in the future.

C# in Easy Steps by Tim Anderson (1-84078-150-5), Computer Step*, 192pp @ £10-99 (no US price)
reviewed by Francis Glassborow

I guess for the raw amateur novice or for the professional who only wants a superficial quick once over this book might be of some use but for most its superficial approach makes it useless.

I have no doubt that it will get good reviews from first timers who will not realise that the ease with which they work through it has little to do with its use of English or its use of colour and interminable screenshots. The reason it will seem so much easier than many other books is that it is completely superficial and makes no immediate demands of the reader's intelligence.

It is only when the newcomer tries to actually do some programming for themselves (there are no exercises or other things that might disturb the reader's peace of mind by letting them discover how much more there is to both programming and C# than has been revealed in this book) will soon find that they are having problems with anything other than the most trivial of programs.

Yes, the book is cheap for a book on programming but even impoverished students understand the concept of value for money. As for the back-cover claim that it offers 'cost-effective training for your staff', well I think that if I had an employer who considered this book a substitute for proper training I would be looking for new employment.

The only person for whom this book has anything substantive to offer is one who is content with superficial knowledge. If you genuinely want to learn either C# or programming this is not the book for you.

Python & Other Languages

Perl for Oracle DBAs by Andy Duncan & Jared Still (0 596 00210 6), O'Reilly, 602pp @ £31-95 (1.41)
reviewed by Joe McCool

Perl, Oracle and Database Administration, along with the world wide web, go together like "horses and carriages". The ability of Perl to enable quick, knock-together, yet rugged scripts suits the tasks of the DBA admirably well.

My own exposure to Oracle has been limited. I am much more familiar with MySQL, but this does not at all limit my appreciation of the current efforts. Indeed the similarities are immediately obvious. A complete munging example is given where data is sucked from a MySQL database and poked into an equivalent Oracle one.

Perl's strength derives from the availability of Modules – ready built blocks of code that can be seamlessly incorporated into user programs. This is particularly true in the area of databases. The Perl DBI (database interface) is one of the most popular. (It is currently maintained by a friend of mine from Belfast). Modules can be pulled down from the web and installed effortlessly - there is even a module available to do this. All this is described fully.

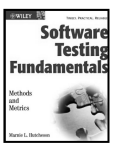
Part III describes a complete DBA Toolkit developed by the authors in Perl. It is available for both Win32 and Unix operating systems.

Another area of covered is the user interface, again implemented by modules. One being the Perl/Tk module, which enables the user to build professional Oracle DBA GUI's with little heartache. All explained well, especially GUI's for DB tuning.

Another thing that that I found useful about this book was an appendix on regular expressions - not just the technical aspects, but the historical development. I had no idea that Alan Turing played a part, nor that Godel was involved. It seems that the mathematics department at Princeton was central to the development of regexes. Godel, Turing, Stephen Kleene and Alonzo Church all lent a hand there at some stage.

Yes, I can thoroughly recommend this work. I got a lot out of it. The style is clear, jovial and a pleasure to read.

Methodologies & Testing



Software Testing Fundamentals by Marnie L. Hutcheson (D-471-43020-X), Wiley, 408pp @ £27-95 (1.43)
reviewed by Chris Hills

Another book on testing, this is good! It means that more people are taking testing seriously. It also means that testing is becoming a formal part of system design.

The book covers some familiar ground and gives the usual definitions but if you have not done much formal (I mean organised not Formal Methods) testing before these are needed. Actually they are needed anyway in a skill

where everyone seems to have their own definitions. The book is testing in general and will suite most commercial software: That is not the safety critical (usually embedded) systems.

The reason I suggest this is that the book has its feet firmly on the ground when it comes to how much work you can realistically do in general commercial development. One of the premises in the book is Most Important Tests method. What you need to test first and what can be left to later. This tends to suggest it is testing for desktop systems rather than embedded systems.

The other main theme is planning: Equipment, people, effort, cost and time. The book requires a Test Inventory and gives suggested templates for this method. Another method described is automating as much of the system infrastructure. That is the reporting and documentation not the actual testing. This uses MS Office, which most people have on their PC. The methods suggested here make this book worth the money on it's own.

There are some exercises and I became suspicious that this was going to be another "Course Book" but no, some of the answers are in the back of the book and the rest, along with other resources such as the templates used in the book are on `www.testersparadice.com`. The site is rather light but it is up to the readers to give some feedback and start something on the discussion forum. However I should not complain as I have several books that promised a web based support that never materialised.

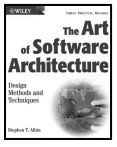
The exercises and suggestions at the end of some chapters are reasonably generic and they data analysis is reminiscent of Macabes Metrics. Though the only two coverage methods shown are statement and branch.

One fascinating area for me is the questionnaire at the back along with the results the author has at the time of going to press. These questionnaires the author has used on course on testing both sides of the Atlantic. The results are interesting, more so those that are split between the US and UK.

The book is written from many years experience and is honest. The author explains that at one time they were on "the other side of the fence" for some arguments and what changed her mind. She also understands why people have other views. This is refreshing in a book where an author usually evangelises their method against all others.

Personally I think this book is worth the money for all team and project leaders just for page 28… come to that just for the last paragraph on that page. Those 9 lines will be one of the most persuasive arguments for getting "creative", "free thinking, "programming is an art" programmers in line with procedures, standards and testing.

Overall I like this book. More for the web, database and desktop programmers than the embedded and high integrity areas but on balance a good book. Recommended.



The Art of Software Architecture by Stephen T. Albin (0-471-22886-9), Wiley, 312pp @ £31-50 (1.43) reviewed by Mark Easterbrook

I approached this book with some trepidation, partly because both "Art" and "Architecture" when applied to software can be controversial subjects, and partly because the title and cover hint at a difficult subject covered comprehensively and I have been disappointed in the past when books fail to deliver their promises. However, once I opened the book I found it lived up to and exceeded my expectations.

If "Art" is the "application of imaginative skill" (Collins Concise) then it does not preclude practising it using procedures and process as well as accepting an element of creativity. Architecture invites a comparison to civil architecture, but the author addresses the metaphor early on, comparing software systems with urban developments rather than individual buildings, and warns against the fallacy of this metaphor by analogy.

After introducing the subject in chapter one, the next few chapters delve deeper into the subject: Chapter two, with the slightly misleading title "Software Product Lifecycle", covers the development or project lifecycles from different viewpoints based on the Rational Unified Process (RUP). Chapter three "Architecture Design Process" looks at architecture design compared with engineering design looks at the interdependencies of design elements. Chapter 4 "Introduction to Software Design" discusses Function (what it does), Form (What it looks like) and Fabrication (how it is built) and reflects that software design is more a creative or artistic process rather than an engineering discipline. Chapter 5 covers complexity, modularity, coupling, cohesion and interdependences illustrated with examples of design structure matrix diagrams.

At this point, almost half way through the book, the narrative takes a step back from the detail of design and implementation and chapter 6 looks at models and knowledge representation: UI models, behavioural and functional models, and models of form. Chapter 7 follows on with architectural representation such as data view and process view. Chapter 8 covers the conflicting and differing views of quality including the use of metrics to measure quality.

The remaining chapters delve more deeply into software architecture building on the material introduced earlier in the book, and the book ends with a chapter on software architecture quality.

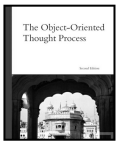
If I can find fault at all, it is in that, by necessity, it is a difficult book to read as it covers a complex subject area in depth. Thus, it is not a text to be consumed quickly, but a valuable resource to be sampled in manageable chunks over months and years in-between applying the knowledge and principals gained. Recommended.

Convergent Architecture by Richard Hubert (0 471 10560 0), Wiley, 276pp @ £29-95 (1.34) reviewed by Silvia de Beer

Convergent Architecture is a high ceremony methodology for developing software. The term Convergent Architecture indicates the whole process of development, from business modelling through to code generation, including testing and deployment. The author advocates one stream of development, using (where possible) one tool to refine the models of previous steps. By doing this, no information is lost between the different phases of the development.

The book is well written, but uses too many abbreviations to my taste. Halfway through the book I lost interest a bit, because Convergent Architecture was advocated as the new solution to all the software development problems. The descriptions given were very theoretical and repetitive and, above all, aimed at very large companies. The IT-organisational model discusses all the roles in the organisation, which you will never be able to fill in a small company, but only in a company of over 100 people if you would count the number of roles. Of course, a person can fulfil more than one role, but still, in my opinion there are too many roles for a small company.

I think that it is a good thing that the author advocates that the whole process of development should become a more continuous flow. Supported with tools during the whole development process, and with as much as possible be automated, i.e. code generation and automatic model validation, to avoid as much as possible the tedious programming and debugging phase, which might generate implementations which do not match completely with the model. The author advocates that we should avoid changing the source code, but rather we should change the model, and regenerate the code. Software development should become more repetitive, to avoid reinventing the wheel for every new project.



The Object-Oriented Thought Process 2ed by Matt Weisfeld (0-672-32611-6), Developer's Library*, 270pp @ £21-99 (1.36) reviewed by James Roberts

When I first picked up this book, I was hoping to read insightful commentary on the process of developing object-oriented software. Instead, the book opens up with basic information about OO constructs (e.g. 'what exactly is a class?'), illustrated with Java code. I found this somewhat baffling. Surely, anyone that can understand Java understands what a class is?

Some of the chapters of the book were potentially useful, discussing (in a relatively basic way), for example, the advantages of clearly defined interfaces, and abstraction. These chapters might be useful reading to someone who had just got to grips with the syntax of an OO language, but was lacking guidance for system design. A detailed example showing how a blackjack game might be designed worked well, and was clearly written.

Towards the end, I felt that the author was trying to bulk out the book. A nod to patterns included an implementation of 'Singleton'. Mentioning patterns I thought appropriate, but the level of detail was almost certainly not. Perhaps a more general description of the wider use of rather more interesting patterns might have been more useful here.

Some sections left me slightly baffled. A comparison of XML and HTML did not seem appropriate for this title, and neither did a section on JavaScript. They seemed to be

somewhat off-topic, and although they might form an interesting discursion for some readers were probably a little too detailed for this purpose. Some readers might also find them somewhat patronising (not having studied object orientation does not imply that you know nothing about computers).

In short, about half of this book would be reasonably good introduction to OO programmers who have just started learning a programming language but needed help seeing the big picture. Much of the rest, in my opinion, fails due to a lack of focus on the target audience.

Critical Testing Processes by Rex Black (0-201-74868-1), Addison-Wesley, 566pp @ £37-99 (1.32)
reviewed by Mark Symonds

Do not buy this book if you want to improve your debugging skills. This book is about the test process, giving examples from a fictional case study of a new release of a word processor.

The case study is for a major project at a large company with separate departments dedicated to programming and testing. The impression given is that the author's consultancy specialises in testing for large companies and no guidance is given on how to scale down the processes.

The book has four sections: Plan, Prepare, Perform and Perfect.

Plan covers risk analysis, and work planning. Much emphasis is given on obtaining stakeholder involvement. This does seem overlong and could have been improved with some pruning of the text.

Prepare covers hiring and building test teams, implementing test systems and system coverage.

Perform is the testing phase and covers handling new test releases.

Perfect describes the bug reporting process and the emphasis here is prioritising bugs and making sure that they can be reproduced before reporting them.

Throughout the book, the recurring theme is of using test feedback to improve both the testing process and the software under test.

There are errors in the book such as the graphics and text on page 392 being out of sync which should have been found during proof reading.

Much additional useful material is also contained on the author's web site at `www.rexblackconsulting.com`.

Embedded and Real Time

Real Time Systems Design and Analysis by P Laplante (0-471-22855-9), Wiley, 504pp @ £52-95 (1.53)
reviewed by Chris Hills

"This book is an introduction to real time-systems. It is intended not as a cookbook, but rather as a stimulus for thinking about hardware and software in a different way. It is necessarily broader than deep. It is a survey book, designed to heighten the reader's awareness or real-time issues."

At least that is what the first paragraph of the book's preface says and so far, it seems an accurate description. It goes on to say it is broad rather than deep and that, as it is pragmatic, some of the author's views may be controversial. I would agree with that too! I would argue some of the points in the book but I think controversial is a bit strong. This book is in its third edition so it is clearly doing something right.

This book covers real-time systems in general. As such it is not going to cover many topics people will need. This is inevitable in a book of less than several thousand pages. It looks at POSIX (mainly used by Linux/Unix) and real time OS so this is a book aimed mainly at the 16/32bit and upward systems. It is not going to be a great deal of use for those working in the 8-bit field (which is the largest group). That said it is a good general book on real time systems. I would suggest that students should have this book. Whilst the author is a lecturer I do not think this is a course book as such but would make a good general background and reference.

One chapter I was very please to see is the one on requirements and documentation. This includes a section, which I have not seen outside some standards, on words and phrases to use and avoid… for example "adequate", "if practical". It is this that helps set the reader thinking and hopefully realising how important properly worded specifications and requirements are. This leads into system design. That is "system" not software because real time systems are 50% software, 50% hardware and 50% Systems. The example used is a four-way traffic system. Not exactly tight, milli-second timing but a useful safety critical example.

As with all things in this book formal methods are lightly covered and, I think, objectively covered. Other things covered are Petri-nets, UML and, with less emphasis, structured methods.

The hardware side of the book is light but I would think students would need a separate book on digital electronics and MCU. However, the book is OK for the target audience. At the level of systems discussed there would be separate hardware and software teams.

As the author said in the introduction, this is a pragmatic book and reads more like an engineering book than an academic tome. This explains the chapters on performance analysis, engineering considerations and metrics. These are written with a "real-world" feel to them. Every time I dip into the book, it seems larger than its 450 pages…

I think the author has written a very good book. His notes indicate that this third edition is a complete re-write with 50% new material. So I would think even owners of the second edition might want to look at this book.

I would recommend this book for all students, though it is expensive, and for all new engineers as a general reference book. You will need other books for in depth information but this will be a good starting point. Recommended.

Linux for Embedded and Real-Time Applications by Doug Abbot (0-7506-7546-2), Butterworth-Heinemann, 250pp @ £32-50 (1.54)
reviewed by Chris Hills

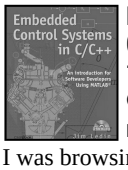
This is a short book… The page count is 250 but in fact, a lot of this is the RTAI and POSIX API information. The text itself is less than 195 pages of large type. Much of the information is very basic and in the Linux man pages or part of the installation guides.

The book starts with a simple description of the memory models for x86 (real and protected) before going into Linux at a fairly superficial level including the installation of Blue Cat Linux. (This is by a company originally called Lynx who also do a hard real-time POSIX RTOS. I did a couple of device driver courses at their offices in Sunnyvale one January).

However, the author seems to have not taken to Linux in that he relates everything, some times erroneously, to MS Windows and does not seem to know some of the reasons, history or background behind some things. For example, the author says that device independent IO is nothing new but Linux takes it further by treating every device as an entry in the file system. This is nothing new it is how UNIX has worked for a decade or so before DOS let alone Windows.

I found the book superficial and rather lightweight with a large type on small pages. I thought that I was being too harsh as I have a Unix background so I gave it to a colleague of mine who is just getting to grips with Linux and building some systems for embedded use. His comments were the same as mine. There is little that is not in the man pages or freely available in many on line documents. There is I am afraid no added value that would warrant buying the book. I cannot see why Newnes have done this book as they already have Lewin Edwards: "Embedded System Design on a Shoestring" which is also an embedded Linux book (targeting an ARM7 board).

Ironically, the CD with the book contains an electronic copy of the book. It is ironic because the last Appendix of the book is Richard Stallman's text "Why Software should not have owners" which, as far as I can see, would suggest that should be free to copy the electronic version of the book for free… Perhaps the author should make this version available on a web site and use the feedback, and more research, to create a better second edition. Not Recommended.



Embedded Control Systems in C/C++ by Jim Ledin (1-57820-127-6), CMP Books, 239pp + CD @ £38-00 (1.31)
reviewed by Francis Glassborow

I was browsing in Blackwells a few weeks ago when I noticed this book. What intrigued me was not the title but the sub-title, *An Introduction for Software Developers Using MATLAB*. My contact happily provided me with a review copy.

The interesting thing about that sub-title is that it couples software developers with MATLAB. For those of you that do not know, commercial licences for the latter start at around $1900 for an individual. That is an awful lot of money in the context of most development tools. It is not an unreasonable sum for someone working on high-integrity or safety-critical software. A considerable proportion of control system software would come under that heading. A company working

on products that would include one or more embedded control systems should be happy to pay out substantial blocks of money to ensure their employees have relevant tools.

I was not far into reading this book before I realised that there was a curious juxtaposition of a premise that the software developer reading the book would be unfamiliar with the mathematics of control systems and that the reader would be comfortable with mathematical formulae up to and including ones involving definite integrals.

It seems to me that the author expects the reader to use MATLAB as a simulation tool to test out his models before implementing them in either C or C++.

I think I have to stop this review here because without access to a copy of MATLAB it is effectively impossible to explore the potential experiences of someone in the target readership. The author's assertions that this book is for software developers without prior experience in control systems and without any mastery of advanced mathematics may be true. However I lack both the time and the tools (MATLAB) to validate the claim.

If there is a reader who meets the requirements (including that of lack of experience) and has access to MATLAB I would be happy to pass the review copy on to them for an in depth review.

Games Programming

MUD Game Programming by Ran Penton (1-59200-090-8), Premier Press*, 666pp + CD @ £32-00 (1.56)
reviewed by Paul F. Johnson

This book started well, albeit a tad slowly. Given the difficulty in teaching Multi User Dungeon (MUD) programming techniques, this is all well and good. I would have preferred if a properly platform independent stance had been taken, but that really is not that important given everything was explained clearly with time taken for those not using the Win32 platform.

The code examples and networking (an extremely important aspect in MUD games) are well thought out and well documented. The only downside is not really to do with the author, but the really awful way Premier Press lay out their books. They are not pleasant to read. The grey boxes (usually some form of note) and odd fonts are unfriendly – I have seen this in many Premier Press books. I find it really does get on my nerves as it detracts so much from the book's content.

MUD programming is not easy. Imagine a typical adventure game on your computer. You interact with the computer, it responds and that is how it plays. Cause and effect. Very simple; an idea which has been around since Crowther's "Adventure" game.

Now, imagine you have thousands of players, all at varied points in the game, all putting in some form of input, all of the input playing some role or other in the adventure arena and all the time, a central server having to control and play the game. Now you can see the difficulty in writing a Multi User Dungeon game. Through lots of explanation, the book demonstrates how best to do this task.

The CD, which comes with the book, is rather good with not only the code samples, but a better implementation, various libraries and even some bonus material.

The Web & Networking

Network Troubleshooting Tools by Joseph Sloan (0 596 00186 X), O'Reilly, 345pp @ £28-50 (1.40)
reviewed by Mark Easterbrook

This is an essential reference for anyone who has to diagnose and resolve problems with IP networks. Although the target audience is individuals new to network administration, it contains a wealth of information for anyone working with IP networks. The focus of the book is networks as seen by software at the hosts connected to those networks, and this it covers comprehensively, but it does not attempt to cover the core network infrastructure such as cabling and routers (nor the software running on routers) except for what can be seen and inferred at the edge of the network.

Each subject area is tackled by describing the technical detail including: the principles of good network design, examples of what can go wrong and how to look for and identify problems, and the tools needed to diagnose and fix them. Both *nix (Solaris, RedHat Linux and FreeBSD are explicitly covered) and Windows (95 to 2000, but not XP) based tools are covered in a pragmatic way, recognising that both have their place and should be in the toolbox of the network trouble-shooter, but the emphasis of the book is definitely Unix.

The subject areas covered include host and network addressing, network characteristics, low-level packet analysis, automatic and dynamic network configuration, device and performance monitoring, and application-level tools, as well as chapters focusing on network management and troubleshooting strategies. Finally, there are two appendices for software sources and resources and references.

In a fast changing industry, detailed technical information often becomes dated quickly, but apart from the appendix, this book is likely to remain relevant and topical until the widespread introduction of IPv6 networking.

2nd review by Alan Lenton

This is a useful book for those who are not full fledged system administration professionals, but who have to administer small networks as part of another job, such as programming.

One of the key problems when something goes wrong on a network is knowing where and how to start looking for the problem. This book is a good place to start. Apart from anything else, it tells you which tools are useful for dealing with which sort of problems. Always a big help when you are dealing with something that is not part of your primary work.

One of the things I did like about the book was the way it did not neglect the boring but important hardware level - including cabling problems, which in my experience are all too often overlooked. From there the book moves on through the different levels of the network including device driver problems, TCP and UP packets, software connectivity and application level programs.

The best way to read this book is to scan it through from cover to cover, so that you have an idea of where to look in it when something goes wrong. However, a good case could also be made for installing, and using, at least some of the measurement tools at an early stage. As the author correctly points out, unless you know how your network normally behaves, you are not like to spot trouble early enough to nip it in the bud.

The only caveat I have is to warn readers that none of the tools are dealt with in depth, because that is not the purpose of the book. However, the tools are covered in sufficient depth to get you up and running with each tool.

Definitely a useful book.

RADIUS by Jonathan Hassell (0-596-00322-6), O'Reilly, 190pp @ £24-95 (1.40)
reviewed by Mark Easterbrook

If you did not know that RADIUS is the "Remote Access Dial In User Service", a challenge and response authorisation access protocol, then you probably would not give this book (and this review) a second glance. The target audience is therefore those who already know basically what RADIUS is and what it can do for them, but need either a tutorial or a reference manual, possibly both.

The first four chapters take the reader from an introduction to AAA (Authentication, Authorisation, Accounting) through to detailed explanation of the base RADIUS message structure and use. There then follows two chapters describing how to configure and use freeRADIUS, an open source RADUIS server. The remainder of the book completes the study of RADIUS by examining other uses, security and new developments.

This book is a good introduction and tutorial and is worth reading before delving into the RADIUS RFCs. It is also a good reference with clear description of RADUIS attributes and a useful attribute reference appendix. However, the RADUIS standard is defined in RFC2058 and the book should be considered a complement to, and not a replacement for, the RFC document.

RADUIS is a base protocol containing many optional elements or context sensitive, it is also intended to be extended by use of the Vendor Specific Attribute. This means that for most uses of RADIUS a description of the base protocol is insufficient and needs to be supplemented with vendor or implementation specific documentation.

In the AAA domain, RADIUS is being superseded by Diameter (RFC3588) and so the RADUIS protocol, and thus this book, is only of use to those already committed to using it.

Fun Web Pages with JavaScript by J. Shelley (0-85934-520-3), Bambini Computer Books, 344pp @ £7-99 (1.83)
reviewed by Paul F. Johnson

JavaScript is one of those things you either like or dislike. In one respect, it brings interactivity to the Internet. On other hand, due to vendors implementing JavaScript differently it makes cross-browser compatibility troublesome. Then you have the additional problem of graceful

degradation for non-JS browsers. Such are the pleasures of Internet programming.

The Babani range is cheap, cheerful and surprisingly good. Fun Web Pages is no exception. It is assumed you know nothing at the start and takes you through the language. Even better the code has been tested on the main web-browsers and is new enough to be happy with the newest of Internet standards.

The book's strength is that everything is clearly explained and does not try to baffle the reader with the complexities. It even takes security and security issues seriously.

Where it falls down though is that interfacing to SQL is not covered neither are other newer technologies. Cookies, form validation and transparencies are in the book and well documented.

The questions are taxing enough to be enjoyable and make the reflective process of what you have learned.

General Programming

Object Thinking by David West (0-7356-1965-4), Microsoft Press*, 334pp @ £33-99 (1.47)
reviewed by Alan Lenton

This very useful book will provide much food for thought to those who think their programming is object oriented.

The aim of the author is to teach programmers to think in an object-oriented manner from the very start of a project, rather than confining it to a consideration of the solution domain. While David West favours agile programming methodologies, the book is far wider in scope, and contains lessons for all programmers.

The book starts by setting the concepts of object orientation and agile development firmly in their historical context. They are seen not merely as part of the debate in computer science between structural and object methods, but within the broader sweep of the debate about the role of formalism and hermeneutics in science. I confess that as a sociologist as well as a programmer, I loved this part of the book and was inspired to dig out my copy of Paul Feyerabend's 'Against Method' for a re-read.

The key idea of the book is that objects should be sought in the problem domain. West calls this 'domain anthropology'. Some of the analogies with regular anthropology are a little forced, but the fundamental idea is sound.

West is keen to change the culture of programming, which he identifies, correctly in my view, as crucial to improving the skills and abilities of programmers. The book is an excellent start, but there is a long way to go yet!

Recommended.

User Interface Design for Programmers by Joel Spolsky (1-893115-94-1), Apress, 159pp @ £21-50 (1.39)
reviewed by Mark Easterbrook

This book should be compulsory reading for anyone designing man-machine interfaces. This book could also be called "User Interface Annoyances" or "The Dummy's Guide to irritating your users". It will strike a chord with anyone who has struggled with a computer interface as the author barges his way through numerous examples of bad user interfaces, trashing the designers ruthlessly, and then shows how a little thought and better intentions, and to be honest, probably some 20-20 vision, could have produced something so much easier on the user.

Have you ever clicked the mouse but the pointer moved ever so slightly and it did the wrong thing? Or tried to get the cursor between two lower case l's in an edit box? Joel exposes the stupidity in chapter 10 "People Can't Control the Mouse", then shows how easy it is to change the design so accurate mouse control is no longer required.

"This will delete your file. Are you sure? Yes, No, Cancel". Are you confident you know which button to click? We all know nobody reads manuals. Joel shows they do not read the screen either. In most cases they just hit "Yes" to "Are you sure?" without thinking – the dialog box does not protect, it just annoys. What is wrong with providing an "undo" instead?

Why is it so difficult to take the Windows briefcase home with you? You do not have any problems with your real one! If you write on a document on your real desktop, it stays written on. So why does this not happen on your computer's desktop? Why do you have to save it? The "Broken Metaphors" chapter examines these issues and more.

The Microsoft Windows interface that is not the only target of Joel's tirade. The Macintosh interface also takes a bashing (Empty the trash can because it looks untidy – oops, you cannot undelete now). The [Li|U]nix command line interface is impossible without the manual or a guru, and we know "People don't read manuals".

The title of the final chapter sums up the book quite nicely: User Interface design is "Programming for Humans". Highly Recommended.

Interaction Design by Preece, Rogers & Sharp (0 471 49278 7), Wiley, 519pp @ £29-99 (2.13)
reviewed by Paul Usowicz

Owning a large number of books, I have various methods of organising them. One portion of one shelf is dedicated to the books that I consider special. These books are on various subjects but are all books that I will read repeatedly due to their excellent content, well written text and personal relevance (perhaps an ACCU article is brewing here!). Luckily there are only a few books in this section, which leaves enough room for this title to take its place along side my other 'classics'.

Interaction Design is quite simply a superb book. The authors know their subject and present it well. Although three separate people author the book, it was in no way disjointed and was a pleasure to read.

The book is about human-computer interfaces with a strong bias towards software. It would be wrong, however, to classify this book as 'software engineering'. It so much more than this and covers the whole aspects of human-computer interfaces including graphical user interfaces, the World Wide Web and wearable computers. Throughout the book are interviews with clever people, exercises to complete and tasks to carry out.

The book is well supported by its companion web site. It contains examples of tasks completed, links to sites of interest mentioned in the text and extras like power-point slides and case studies. This is the best HCI portal I have come across and have visited it regularly during the course of reading the book.

The preface lists various suggestions for usage of the book suggesting various relevant chapters to read dependant upon your particular needs. I would recommend that if you choose this route and read the relevant chapters suggested that you should then immediately read the rest of the book as I found no chapter lacking in useful information.

Everyone is recommended to read chapters 1 (What is interaction design?), 6 (The process of interaction design) and 10 (Introducing evaluation) with software developers also recommended chapters 7 (Identifying needs and establishing requirements) and 8 (Design, prototyping and construction). Chapters 7 and 8 were especially good with lots of common sense and good advice. As stated above, these five chapters will give you a good working knowledge but the rest of the book is worth reading as well. My software development methods are already changing for the better and I am trying to get my company to buy the book (I'm not letting mine go) so the rest of the department can read it. The book is also not limited in scope to just the software developers. Sales and marketing would learn a lot from this book and would end up requesting much more useable products.

The whole book is written with hardly any references to specific languages or operating systems making it a book that I will have around for some time and one that will not easily date. As a multi-platform developer (PC, PDA and Smartphone) I was glad to see some good advice on the need for differing interaction based upon the device being targeted. Too many people think porting to another device is simply a matter recompiling without realising the huge part that the user interaction plays. I think it will be some time before I find a better book than this on HCI (unless the authors are planning another one!). This book is definitely recommended for all software developers who target devices that require user-interaction.

Solutions to Parallel and Distributed Computing Problems by Albert Y Zomaya et al. (Editors) (0 471 35352 3), Wiley, 272pp @ £58-95 (1.70)
reviewed by Christoph Ludwig

This book is a collection of ten independent research papers by different authors that mostly report experimental results about heuristics for solving optimization problems. Though familiar with academic texts, I cannot access their academic contributions since they are not from my area of expertise.

The heuristics considered are inspired by ideas taken from nature: Genetic algorithms are

most common, but cellular automata, simulated annealing and neural networks are also considered. However, the discussions stay abstract; they ignore the details and complications one faces when implementing such algorithms on parallel or distributed hardware.

Most of the articles consider resource scheduling problems (e.g., Flow Shop Problems, Load Balancing Problems etc.) that are NP-hard in general. In practice, one relies on heuristics in order to find solutions that are "good enough". However, the fact that the presented heuristics are tailored to the specific problem they were developed for narrows the readership that can easily profit from these results. Only their general approach can be transferred to different problem domains.

Eight articles discuss aspects of genetic algorithms whence their general approaches are quite similar and there is some redundancy. The different priorities assigned by the respective papers are most likely significant only for readers interested in the specific problems discussed.

In the articles that study cellular automata and neural networks, readers without some background knowledge of the context where will not see how the results fit in. Thanks to their comprehensive bibliographies, they may serve as a starting point for further reading though.

Overall, I think this book fits well in an academic library where articles and textbooks for further reading are easily accessible. However, it is not suited for readers who want to learn how they can adapt the ideas of genetic algorithms etc. to their problem at hand.

The Shellcoder's Handbook by Jack Kozi et al. (0-7645-4468-3), John Wiley & Sons Ltd, 620pp @ £33-99 (1.47)
reviewed by Richard Putman

It is forgivable, looking at the main title, to think that this book is a reference for writing bash or korn shell scripts, but in fact 'shellcode' is the name given to the piece of code that is run after gaining control of a vulnerable program. Shellcode is so named because often the injected codes are instructions that will launch a root shell under unix.

If you have ever wondered about the story behind the security holes announced seemingly daily this book will show you why they occur, how the exploits work and the methods that led them to be discovered in the first place.

The book has four parts: the first hundred pages covers an introduction to exploitation on Linux x86 systems, the second hundred looks at Windows and another hundred covering Solaris and HP Tru64 systems. The third part looks at how to discover vulnerabilities with some useful tools and a final more advanced section looks at alternative shellcodes, database and kernel hacking.

There are a number of typos in the text and no errata page has yet appeared on the publisher's website, indeed the links to resources mentioned throughout the book have yet to appear either, although the example code is there for download. The text is well written and structured with a conclusion at the end of each chapter.

Much of the book is assembler, often embedded in C code, or occasionally python scripts and although there is a brief review, you should already be comfortable reading assembler, or be prepared to learn quickly, to enjoy this book.

Many of the ideas are simple – overfilling buffers that are processing user input, but the low-level nature, restricted memory spaces and unknown elements, such as where the code will be executing in memory, often create layers of dependent problems magnifying the complexity. It can take considerable skill and ingenuity to turn a vulnerability into an exploit, not to mention a certain amount of luck, unsurprisingly it is often thought of as a black art.

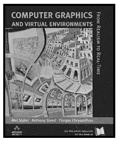
This book then is essentially a compendium of the techniques and resources used by several clearly experienced hackers; the aim being to teach a creative approach rather than list known exploits. What comes across in the tone of the book is the authors' desire for the reader to succeed and enjoy the challenge as much as they obviously do.

There is quite a bit of hand holding and encouragement early on to get past the point where most people give up but it is also a rich source of information with index and deserves the title 'handbook'.

For programmers who have no interest in creating their own exploits, is there anything in this book? Well yes, the section on vulnerability discovery contains interesting information about the authors' favourite tools; there is a chapter on fuzzing (generating automated test input to discover bugs in your program) and source code auditing showing many common faults in C code. However, the direction of the book is very clear – to subvert a target system.

Writing shellscripts is surprisingly good fun and the book will appeal to those who enjoy tricky programming puzzles and those who want an advanced but accessible low level security perspective on the programs they write and the operating systems they use. Highly recommended.

It seems to me you should read this book even if you never intend to crack anyone else's system. You need to understand what you must protect your programs from. Francis



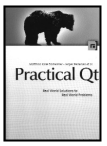
Computer Graphics and Virtual Environments by Mel Slater et al (0 201 62420 6), Addison-Wesley, 571pp @ £34-99 (1.86)
reviewed by Alan Lenton

This is a competent book covering similar territory to Foley and van Dam. As always with books on this subject a solid grasp of matrices, calculus and geometric algebra is needed although the authors provide a mini refresher course at the start. The book breaks with convention by starting from illumination rather than polygon drawing. I am not sure how much better this is as a teaching device, but it certainly does not detract from the book, which covers all the components of the standard graphics pipeline.

The last few chapters are, to my mind, a bit scrappy, being a whistle stop tour round the options available for a number of more advanced topics. I am not really sure that they add very much to the book, given their brevity.

One particularly useful aspect of the book is that the examples are in OpenGL and VRML97 as well as some 'C', which makes it relatively easy to 'borrow' examples for your own use. A lot of material is packed into the pages of this book and, in my opinion, it represents good value for money, although I suspect it would be more useful as a college textbook than as a reference for working programmers.

A useful book on a specialist subject.



Practical Qt by Dalheimer et al. (3-89864-280-1) 253pp. Available from amazon.de or from `http://www.kdab.net/practicalqt`. 36 Euros
reviewed by Paul F. Johnson

For those who have the O'Reilly book "Programming with Qt", you will already be familiar with Kalle's style. Clear, concise and easy on the eye.

This book is the perfect companion to it. The difference being though that this is an answers book. You need to be familiar with using moc to compile some of the code example.

The uberfurer of dire programming books (Schildt) has something on the front of his books which goes along the lines of if you want answers quickly, just ask the expert. Unlike Schildt though, this is one book I would dip into.

It quickly and clearly explains how to do certain things in Qt (such as circular widgets – it really is an answers book, but it still teaches the reader how it works and how to best approach a problem. Matthias (and his co-authors) really do know what they are on about.

There is only one thing wrong with the book: it is not big enough. I hope the authors bring out a volume 2 (and 3) and possibly even a Qt4 companion.

I have been using Qt for quite a few years now and the material in there has made me look afresh at some of the practices I had adopted – it is an eye-opener.

Highly recommended.

Non-Programming

High Tech Crimes Revealed by Steven Branigan (0 321 21873 6), Addison-Wesley*, 412pp @ £22-99 (1.30)
reviewed by Francis Glassborow

I am not going to do much more than draw your attention to this book because it is only indirectly of interest to readers of C Vu.

There is an element of the autobiographical in that the contents rely heavily on the author's direct experience. The author covers his (largely US based) experience with a range of IT based crime over the last decade. It makes disturbing reading in places because frequently detection depended on chance, or an exceptional level of curiosity from one of the participants.

The book starts with a chapter about an incursion on a telecom company's computers circa 1995 and then documents various other

computer based crimes before concluding with a number of chapters on what not to do and what the experience of the last ten years has taught the author (and hopefully, at least a few of the law-enforcement agencies).

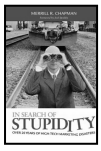
If you want a clearer understanding of what is behind some of the headline stories, or you are curious as to what happened after the story faded out from the public consciousness this book is well worth your attention.

Teach Yourself OpenOffice.org All in One by Greg Perry (0-672-32618-3), SAMS*, 515pp + CD @ £21-99 (1.36)
reviewed by Francis Glassborow

OpenOffice is a very useful free application bundle from the OpenOffice foundation (which is supported by Sun Microsystems). Among its advantages are that it handles most Microsoft Office files and it exists for Linux and Mac OS X machines as well as for Windows. I keep a copy on my laptop so that I can use material I have prepared with Office 2000 when away from home. (Note that if MS had more reasonable licensing for individuals, I would not need to do this).

The book is divided into five sections, one for each of Writer (word processor), Calc (spreadsheet), Impress (presentations) and Draw (simple graphics tool) and a final short section some other features of OpenOffice.

I think this book would make a good companion for anyone who has decided that they want to break their reliance on MS Office as well as those who want to use a office application suite that is largely independent of their choice of platform. It does not go into excessive detail and probably does not cover enough for those who are expert users of office applications (certainly I require more detail before I could use OpenOffice as more than a secondary suite).



In Search of Stupidity by Merrill Chapman (1-59059-104-6), Apress, 256pp @ $24.95 (no UK price)
reviewed by Chris Hills

This is a fascinating book. It is not technical and neither is it a business book, neither is it an autobiography but it is a sideways look at the computer (PC) software industry. Rick Chapman has spent his life in the US software industry. At various times as a programmer, an FAE, a salesman and in marketing with many of the Big Names. He has seen it all and in some cases was in the middle of some of the incidents in the book. This is a book written with hindsight and a lot of honesty. As the author says in a couple of places "I was completely right…. For all the wrong reasons!" and "I was wrong… for the right reasons!" There are lessons to be learned, if we can learn them. Though history does seem doomed to repeat itself.

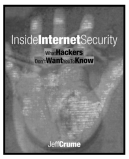
This book looks at why 9 out of the top 10 computer software companies of 1984 are not in the same list for 2001. In the intervening 17 years all the market leaders "committed suicide"... Yes, the only one in both lists is Microsoft. Not, according to the author, because it was clever or its software was the best but because it made fewer of the major "stupid mistakes" the author attributes to Aston-Tate, Novell, DR, Microfocus, Visicorp etc. He asks: "Given that Microsoft software is 'that bad' why are we all still using in it? What happened to Quattro-Pro, Word-Star, Lotus 1-2-3, D-Base. It is not just the software, what happened to the IBM-PC? (not the "PC-compatible" of DELL et al). Why did OS/2 not sweep the world?" The possible answers are in this book.

Just because something is better it does not mean that the world will use it. E.g. "Everyone" uses VHS video, except the professionals who use Betamax, which is technically superior, and no one uses, what was at the time, the technically even better Philips system. This book looks at the marketing equivalents of the Charge of the Light Brigade that caused the downfall of the, often technically better, market leaders.

The author worked for many of the companies concerned, or a closely related company, at the time of many of the stories told and has a unique, inside view. I did wonder if his middle name might be Jonah at one point! This is the sort of story that can only be told with hindsight and far enough away to avoid the law suites. Though that said the book was published in 2003 and covers up to mid 2002 so the later chapters are almost current.

For many this book will be a trip down memory lane for others a look into pre-history. It may only cover 25 years but for some in the industry they probably know more about the dinosaurs than some of the names from the 80's that are in this book. I am not sure if this is business, history, sociology or gossip.

The style is easy to read and humorous in a relaxed way. This is a fascinating read that will make a good book for the holiday or the long summer days. It should be required reading for all marketing departments, project managers, strategy groups and computer courses. This is a book for the summer holiday or to settle down with at Christmas. Recommended.



Inside Internet Security - What Hackers Don't Want You to Know by Jeff Crumme (0-201-67516-1), Addison-Wesley, 270pp @ £30-99 (1.03)
reviewed by Chris Hills

This was going to be one of four types of book: lots of technical detail and code fragments for programmers or sensational stories of the type found in the popular press. It could have dived off deep in to maths of algorithms and ciphers. Fortunately it is the fourth type – a sane sensible look at network security for managers.

I know it says Internet security but these days the Internet is just an extension of a normal office network. At one time viruses were spread on floppy disks, now the vast majority get on to the PC either directly from the Internet or across the office network.

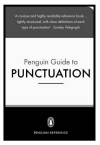
The book has no source code, no maths or protocol bits and bytes. What it does have is a non-sensational look at who hackers are, why they do it and what sort of holes there are. Most importantly, it tells you how to go about stopping them. Well actually it does point out you cannot stop hackers. So there are constant warnings that all you can do is minimise the risks and never get complacent. This is strategy and management rather than how to use specific software or systems.

Interestingly this book is going to make you see that antivirus software and firewalls are not infallible. You cannot just fit them and relax contented that you are safe. Then again it is not full of "scare stories" It is balanced, reasoned and at a level that most managers (technical or non-technical) are going to understand the problems and the solutions in general without getting demoralised or thinking it is easy.

Whilst the book has a slight US bias it is not a problem and everything should apply in most countries and hackers are of course international as on the 'net all geographical places are the same place.

There is the obligatory section on cryptography, public keys etc. and a very useful section on VPN, which is something, many companies now use and many do not for the exact same reasons!

This is by far the most dispassionate and well-balanced book I have come across in this subject. It handles a subject that is both precise yet very nebulous and riddled with myths in a way that lets you see clearly and assess the risks without panic. I recommend it for all non-technical managers... actually all managers, I bet half the technically astute managers do not know the realities of the myths etc.



Penguin Guide to Punctuation by R. L. Trask (0-140-51366-3), Penguin Reference, 162pp @ £6-99 (1.83)
reviewed by Christopher Hill

As collator of the ACCU book reviews, I come across many of styles of writing, spelling and punctuation. I know that there is not one correct form for these aspects of writing, but there ought to be some firmish guidelines, so that you know when you push against one of the mores of writing English.

I enjoyed reading Eats, Shoots and Leaves; Lynne Truss has a very comical turn of phrase and I did learn a little about punctuation, but the lack of an index made it very difficult to use as a reference book.

The Penguin Guide on the other hand has not left my desk. While it does not have the humorous input, Trask writes very clearly with an eye to the detail of the issues under consideration, while at the same time making it easy to remember the points. Did you know there are four separate uses for the comma: listing, joining, gapping and bracketing?

There are many examples of good and poor punctuation on almost every page, with the poor examples flagged with an asterisk to remove any possibility of confusion or doubt. Having read the book from cover to cover, which is very easy to do, the four pages of index make this a really useful reference.

If you are writing (specifications, requirements, books, emails, letters) or editing the same, then you cannot but benefit from reading this book, after which it will find a cosy place sat next to your mouse and keyboard. Highly Recommended.

Secrets & Lies (revised) by Bruce Schneier (0-471-25311-1), Wiley, 414pp @ £11-99 (1.50)
2nd review by Mark Easterbrook

We all live in an increasingly digital and networked world. We also live in a world that seems increasingly hostile, at both the personal level and the global level. Yet, so few of us really take security really seriously: maybe we all lock our doors and windows and install firewalls and virus scanners, but this is just basic stuff – when did you last perform a security audit on your house or your Internet connection?

This book examines the security of the digital networked world and the domains that interface and interact with it, including us, in a pragmatic, myth-busting, sometimes humorous, and often worry-inducing way. It is divided into three parts:

Part 1 – The Landscape – sets the scene, who are they, what do they want, why they want it, how might they get it, and why are they targeting you. If the answer is "I don't know", as it is often the case, you just have to guess and hope you are somewhere close.

Part 2 – Technologies – is the largest section and comprehensively covers the technology used in attack, defence, detection and alerting. The common theme here is that security is like a chain, and is only as strong as its weakest link.

Part 3 – Strategies – looks at the practical side of securing your part of the world. This takes a realistic look at threat and risk analysis and how sufficient defence strategy can be created. Not surprisingly, technology is only part of the problem, and only part of the answer – security is a human issue as much as it is a technical one.

When you have read this book, and I strongly urge that you do, there will be one of two outcomes: You will take security much more seriously, or you will sleep much less easily at night. Recommended.

The E-Myth Revisited by Michael Gerber (0-88730-728-0), Harper Business, 268pp @ £10-99 (1.46)
reviewed by Chris Hills

The E-Myth is the Entrepreneur Myth. It has nothing to do with the Internet, Email or Dot.Com. This book was recommend to me by a successful business owner. He told me that his only regret was not reading it 10 years earlier! I read it and I must say it fires you up with an [organised] enthusiasm. I showed it to a friend and he is now using this book as a start up guide for a business he is planning.

This is not a Get-Rich-Quick book, nor does it seek to sell you anything else. However, there is a web site with additional resources and curses the author runs but these are US based and not needed to use the book.

This will be a difficult review to do as explaining the content gives all the secrets away! The model used for this book is a pie shop run by one person (no, not the well known breakfast/fast food illustration) but it will work for any business. Indeed there are illustrations from other industries including the hotel business. Which is interesting as I recently read the Marriot (hotel) story…. It was in the draw next to the Gideon and Mormon Bibles in the Marriot in Heidelberg! I can see traces of the ideas in this book in the Marriot system.

The story in the E-Myth how the owner of a failing, one person, independent pie shop looks at their business, under the guidance of the author. They analyse the problems and look at the way to get it back on it's feet is in the style of the stories by Plato where the questions set up the next explanation. It is almost readable as a story. At each meeting the pie shop owner has with the narrator more things are explained and suggestions on what to do next is suggested. However this book will work for a business that you have not yet started as well as one that has been running a decade.

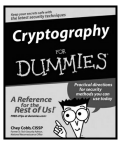
I found the book fascinating as it covered various things I have seen in small companies before. It explained the underlying reasons. It explains why most small companies run into the problems they do after the initial wave of enthusiasm. Why many small companies fall over or do not progress after an initial growth of personnel.

Recognising the symptoms is half the problem the other half is what to do about it. There are several novel solutions which, when you look at them are plain common sense. A couple of the ideas and illustrations come from the authors own company with mistakes they made. So the author has been there and done it himself. Interestingly there is no patented system to buy, no franchise method, just sensible ideas. Though in the US the author does do lectures and business consultancy on the subject. Much like the Team-Start /Mustard and Small business advisory groups in the UK

This is a business book, therefore it is orthogonal to all businesses from IBM to MacDonald's to small SW house to a one-man contractor be it in IT, carpentry or the fishing industry.

The important thing to note is this book is simply ideas and methods. It is not a silver bullet. You have to have a viable business; goods or services to sell that are in need. Be prepared to work hard, the money to start and the discipline to do the paperwork. What this book does is give you the map to the obstacles, and the navigation skills; it does not give you the transport to reach the goal. You have to provide that yourself.

As will be clear from my comments I recommend this book highly. I would use it myself were I to start a business. Further to that I recommend that any one with a small business, including "one-man" outfits to look at it even if you have been running a decade. It could do you a lot of good. I would say at the price (lass than 11GBP) you can not afford to miss this book. However non- business owners, be careful! It could enthuse you to start one yourself. Highly recommened.



Cryptography For Dummies by Chey Cobb (0 7645 4188 9), John Wiley & Sons Ltd, 304pp @ £16-99 (1.47)
reviewed by James Roberts

I was not really sure what to expect from this book. I was interested how cryptography could be described in such a way as it would be suitable for 'dummies', and wondered what the content could possibly be.

It turned out to be a bit of a mixed bag, some of which was potentially useful, while some seemed to be padding rather than sensible content.

On the plus side, the book gave a reasonable summary of the installation of PGP (most examples in the book revolved around how cryptography applied to PGP). In addition, the book gave some good advice on selection and remembering passphrases (although if you read the book from cover to cover you notice a certain amount of repetition in this area).

The book includes some basics – for example, a description of how binary numbers work. Perhaps this is sensible for a book aimed at complete beginners. However, this technical detail was not carried forward beyond a description of the XOR function. (The use of the XOR function in conjunction with a 1-time pad was not explicitly covered). This left me wondering what the intended audience of the book would have made of it.

Other annoyances included little clear overview of how the protocol of use of public keys is used. (A diagram of the interactions required to generate and use a PKI might have been useful.)

In short, this book might be suitable for someone needing basic information about cryptography and cryptographic products that did not want to understand the details. However, in this case I think that there might be several chapters which would not be particularly useful or relevant - the book would condense down to a 'how to use PGP, with some handy hints on remembering your password'.

Not recommended.