Contents

Reports & Opinions

Dialogue

Features

Reviews

Copy Dates

C Vu 17.2: March 7th 2005

C Vu 17.3: May 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

Well, the bunting is down, the turkey has well gone (and if it hasn't by the time this issue hits your door mats, then I seriously suggest throwing it out before it walks out!) and thoughts are turning to how to remove the couple of inches gained through the annual festival of excess.

While I can't help very much with the physical, in this edition I can certainly help with the mental! In the last issue, we had the first part of Derek Jones' statistical results from the last conference. This will conclude this edition, but it has sparked something in my inbox. It seems quite a few of you have been very interested in the results obtained. I've asked Derek if he'd like to submit some more along the same lines, so you never know...

In the Caverns of Your Mind...

I have many many many books in my office. Loads of them. I'd hate to imagine how much they'd cost to replace as there are some classics in my collection which are long out of date and even longer out of print.

Just before the holidays events really took hold as one of the shelves holding the books up (together with a good chunk of the wall) finally succumbed to the forces of gravity, and as usual, it was at around 4am. I don't think I've seen the dogs jump so high – even the deaf one!

While I cleaned up the mess, I came across three books which I thought had vanished when I moved to Haydock over 10 years ago. While they really wouldn't be worth a huge amount today, they are important to me; they were the first two programming books I ever bought and the first programming book I ever bought from eBay.

The books in question here are "Computer Spacegames" and "Computer Battlegames" for the ZX81, ZX Spectrum, BBC, TRS-80, Apple, Vic 20 & Pet. That should give you an idea of the age of them (both published in 1982 by Usbourne). I'd not really read these books in a very long time (well, in 1982 I was 11, so playing football was more important than my ZX81) and decided to look through them – mainly out of interest – and something struck me. While machine power has increased vastly, things have actually regressed in terms of computing.

You What?

Okay, I'll explain what I mean there. Take the following piece of code (as listed in the Computer Spacegames book)

```
INVERSE c
LET y=h*1.3+10
PLOT 200,y: DRAW 34,0
DRAW -4,20: DRAW -13,10
DRAW -13,-10: DRAW -4,20
RETURN
LET y = 172-a*32
INK c
PLOT 0,y
DRAW b,0
DRAW INVERSE 1,100-b,0
RETURN
LET i$=INKEY$
IF i$="a" THEN LET t=t+4
         : IF t>100 THEN LET t=100
IF i$="d" THEN LET t=t-4
         : IF t<0 THEN LET t=f
IF t>f THEN LET t=f
RETURN
```

This is the ZX Spectrum version of a game called "Touchdown". It's not an amazing game, it is one of those land the craft on the platform games. All the above code does is a bit of drawing on the screen and interacts with the player for which key they press to move the spaceship (a user defined 8x8 graphic). It's in BASIC (albeit Sinclair BASIC) and what you see is what you get.

Now, consider what you would have to do to get that to run on a modern Mac or PC (not bothered as to which OS the PC is running – it applies equally to Win32 and Linux/Unix/BSD variants). No cheating here (read no emulators!).

First you need a third party library for the graphics handling (say SDL – I'm keeping this cross platform as I know next to nothing about DirectX). Okay, there is a version for platform X and I can install it. Right. Good.

Next the code has to be converted. That means that some parts will be easier than others and some will need mapping over to SDL. Problem. I need to consult the documents for SDL to see what is the closest to the original. For the parts which are simple enough (simple logic), conversion to C is simple. Moving it to C++, C# or Java may be a bit of a pain, but can be done easily enough.

So we now have the basics. We can't do it natively (that is with *only* the operating system), but with dynamic linking, the final product can still be used by many people – as long as they too have the library I've used. Nevertheless, it is possible.

The code though will have grown quite a lot (in all probability) and to a beginner (which is the target audience for these books), unless it is well documented, it isn't going to be easy to use.

Then comes the make file. Shudder time!

So from something you sit down at, switch on, 2 seconds later have a command prompt and can start working on, you now have boot times of up to a minute, then load either a text editor or development environment, load the source and start to work. 2 seconds can become up to 10 minutes.

Of course, it would be plain daft of me to say development and software quality hasn't improved, but it would also be wrong for me to omit saying that the technology has developed as well.

We no longer use 8 bit machines with a maximum of 40K available memory, 8 colours and a piezo beep for a sound system, and software has come on in leaps and bounds, but is it still as accessible for the newcomer as things were in 1981 or has computing changed to a "them and us" whereby instead of the computer being a portal to the imagination, it is a tool for writing editorials, doing your home accounts or sending emails?

Does your average kid get the same kick out of writing some code as kids between 1981 and 1985 get when they used their BBC Bs, Orics, Spectrums, Dragons et al or has the fun been sucked out of it when you write something like:

```
void moveLeft(
       Position *currentPosition,
       Ship &Tardis) {
  Position newPosition;
  newPosition = checkBounds(
      currentPosition
            - sizeof(Tardis));
  if(!newPosition)
    moveLeft(currentPosition
            - sizeof(Tardis));
}
```

Yes it's logical, but is it really the same in terms of bright eyed fun?

And the Third Book?

Another classic from when I was a teenager: "Creating Adventure Games on Your BBC Micro" by Ian Watt (there were other versions for machines of that era, including the Amstrad 464, Spectrum and Dragon 32).

Now this really was a good book as it took the reader through just about every aspect (at that time) of writing an adventure – from the requirement to get everything down on paper first (the map planning being one of the most important aspects, followed by the puzzles) – to using non-player characters (NPCs). It was easy to understand and even by todays standard, is a great book to have.

Why Did I Bring These Up?

Am I drunk? Am I just in one of those moods to look back with rose-tinted glasses? Am I just one of those cranks who wishes that the BBC B was still the best thing since sliced bread?

The answer is no to all three.

I brought it up for one reason. In comparison to what we have today, things have undeniably moved forward, but at the same time, we've regressed. Books are the same. Sure, things are more complex, but the fun seems to have gone as, more importantly, has the attention to detail. We are getting more and more substandard books being published which really aren't helping.

I recently reviewed a book called "Linux Game Programming" which was truly awful. It was a book which (I've since learned) was written by a committee after the lead author found a new job. The attention to detail was lax to say the least with someone at the publisher adding in notes which instead of helping gave some very poor information, some of which had nothing to do with the material presented.

I doubt that in the 1980s this book would have made it. All right, I'm not that blinkered to say there were not some real turkeys out there (I remember one Oric-1 book which did everything with direct pokes to the screen etc instead of using the built in command – that was a horrid book!), but they were fewer and further between. Was it that back then books were not just off a conveyor belt (yes, Granada Publishing was the exception to that rule!) but written by authors not just interested in getting another DirectX, SDL, or OpenGL book out?

It would be refreshing, to say the least, if an update to the adventure game book was made available for users of C, C++, C# or Java.

Rant over. On with the show!

Paul F Johnson

View from the Chair

Ewan Milne <chair@accu.org>

Preparations for the conference are in full swing, and with a line-up featuring Stroustrup, Coplien, Sutter, Buschmann and many others, I hope you are as excited as I am about it. There is one event at the conference that understandably will be causing less excitement, but which I would like to draw your attention to nonetheless. On the next page you will see the announcement of the AGM. It is easy to forget about this with everything going on around it; but those with long memories will remember the conference's roots as a few presentations held alongside the AGM for added interest.

So, I admit it: Alan and I working through the agenda, the other officers presenting reports – it's not the most gripping hour of the event. But it is an essential task in the running of the association, for the whole membership as much as the committee, so your participation is strongly encouraged. This year we have planned to streamline the meeting as much as possible, minimizing the administrative details in order to use the time to better focus on any real issues that are raised.

One of the new ideas is to make the officers' reports available to the membership in advance of the meeting. The aim of this is to avoid much of the meeting itself being taken up by their presentation. Some, like the Treasurer's report, don't really suit a live reading. At the meeting we plan to move directly to taking questions on the reports. Of course, the key to cusses here is for the reports to have actually been read in advance, so please, when you receive your AGM pack, do please take the time to read and consider it. With your help we can have a more efficient, more productive AGM. I look forward to seeing you there.

Secretary's Report

Alan Bellingham <secretary@accu.org>

Members don't often know what takes place during committee meetings (although they're entitled to attend if they so wish, or receive the minutes on request). So this time, I'll give a quick report on what happened during our last meeting.

Meetings take place four times a year, usually at roughly 3-month intervals (but the summer vacation season tends to mean that a longer gap occurs then). Also, to spread the travelling around, the meetings won't always occur at the same place.

The most recent meeting took place at Jez Higgins' house in Moseley, Birmingham, in the English Midlands. It began at 13:30, and there were eight of us present, with another three experimenting with Skype in an attempt to attend virtually. In the event, domestic broadband Internet turned out not to be quite up to it.

The first activity, after recording who is present and who has sent their apologies, and after approving the minutes of the previous meeting, is to note those actions in the previous minutes that are complete, and those that aren't. This is where the committee keep track of ongoing activity – some things do take years to complete.

In this case, we noted that the 2003 writers' competition had finally been judged (deplorably late – we will do better next time), and that the long-drawn-out process of transferring the post of Treasurer to Stewart Brodie was finally complete. (The new banking rules make changing signatories on bank accounts a very tedious process.)

The next step is officers' reports. Not all officers make a report every time: for instance, since my main activity is organising agendas, writing minutes and the like, there's rarely anything new for me to report. Usually, these reports should be broadly similar to what appears in this journal, with the exception that with four meetings a year, and six journals, matters do get a little out of phase.

Unusually, we co-opted a member, Thaddeus Frogley, to take charge of advertising sales. There was also the announcement that Pippa Hennessy is stepping down as our production editor. (The production editor is the person responsible for the actual layout and so on of the journals, rather than for their content.) This is an important post (though not actually a committee role), and we hope that John Merrells (the Publications Officer) will find a replacement up to Pippa's standard.

The conference is always on our minds, but it's run by a subcommittee rather than by the main committee. We were informed that everything is on schedule, and that a programme has mostly been finalised.

The final item on the agenda (placed last because we knew it would need the most discussion), was the subject of the `accu.org` website. We're very aware that the current website is a bit of a monster – it contains thousands of pages, it looks old-fashioned, and it creaks a bit. Because of this, we have been trying to work out what to do with it. What has been happening recently is that Allan Kelly has taken charge of this, to produce a full plan of what we need, and to effectively outsource the rebuilding of it. This has led to the production of a requirements document (it's difficult to produce the right answer if you don't ask the right question), and this has now been approved. The next stage should be that we get a list of tenderers together, and by the next meeting, we should be going ahead.

Finally, the next meeting is scheduled for 19th February. However, that's a tentative date, and I know that I won't be able to make it, since I won't have returned from the Basler Fasnacht (`www.fasnacht.ch` if you're interested). Hence, it's likely to be rescheduled.

Membership Report

David Hodge <membership@accu.org>

At the end of 2004 the membership stood at 934 with the majority of new members and renewals using the website.

If you have a UK bank account and would like to save £5 on your next year's subscription by paying by standing order, ask for details by email.

Note that due to the lag in getting the banks to process standing order information, you need to initiate this process at least 8 weeks before your renewal date.

If you change your email or mail address it is important that you let me know so that I can update the database. There are occasions when journals can go astray, if you do not get your copies, please contact me.

Standards Report

Lois Goldthwaite <standards@accu.org>

The UK continues to punch above its weight in international standards development.

A UK delegation has attended every meeting of the international C standard committee (WG14) going back to the very first one in September 1986. I doubt if any country except the US could make a similar claim. The first document listed in the WG14 register is N001, 'Minutes of 10 Sep 86 Meeting', by Cornelia Boldyreff, who was the first convenor of the BSI C panel. N005, also by Boldyreff, is 'Comments from BSI' on issues discussed at the second meeting.

The first meeting of WG21, the C++ standard committee, was in June 1991. I haven't been able to determine yet if the UK was represented there, but we've sent delegates to all the ones since then. Derek Jones remembers attending the meeting of the C committee which decided a different group should develop the C++ standard, so he counts that as the zeroth meeting for C++. In 2004 members of the UK C++ panel authored or co-authored 24 discussion papers submitted to WG21.

Another area of UK standards activity is Posix. Posix is a standard for operating system interfaces – 'Unix' was originally a trademark of AT&T, who invented it, so the standard had to be called something else. In those days there were multiple standards for commercial Unix, as various groups of vendors formed themselves into consortia. Posix (Portable Operating System) began as an effort by IEEE and Unix users to standardise library functions so that applications could be ported to different platforms. The BSI Posix panel was officially established in September 1987 (an ad hoc one met a month earlier), also under the leadership of Cornelia Boldyreff. After that the UK played an active role in IEEE P1003 and WG15.

These days the Posix/Unix landscape has changed. There are now three bodies who issue standards on this subject, but there is only one standard. What is called the Austin Group consists of representatives from IEEE, the Open Group (successor to all the Unix vendor consortia), and ISO/IEC SC22 (the parent committee for WG15, which has been dissolved), plus a great many individuals representing their employers or themselves, all working together to produce a single unified standard document. Approximately 600 people take part in the Austin Group work. Their motto for the jointly-developed standard is 'Write once, adopt everywhere.'

An Austin Group plenary session took place in Reading in January, along with the initial meeting of the SC22 Posix Advisory Group which has succeeded WG15. Of the nine people who attended the PAG in person, four are members of the UK panel. Three of these people are regular attendees in the Austin Group's almost-every-week working teleconferences, wearing their other hats: Andrew Josey chairs the Austin Group and is Director of Certification for the Open Group, Nick Stoughton is the Organisational Representative from SC22 and liaison from the Free Standards Group, and Joanna Farley is an experienced expert from Sun.

Speaking of the Free Standards Group, their Linux Standard Base document has been submitted for adoption by ISO and IEC. Formulating the UK vote on whether or not to approve it is an important task facing the BSI Posix panel. If you have an interest in Linux and would like to participate in this work, please send an email to `standards@accu.org` for more information.

(To save myself the trouble of researching through all those meeting minutes, I consulted the memory of veteran UK delegates Francis Glassborow, Neil Martin, Derek Jones, and Cornelia Boldyreff, to whom all many thanks.)

Officer Without Portfolio

Allan Kelly <allan@allankelly.net>

When I agreed to join the ACCU committee last April I did so because I wanted to see the ACCU improve in some specific areas. Principally I had the website in mind. Well, you can't see any changes yet but things have been happening behind the scenes so I thought I'd fill you in on what's been happening and where we are at.

The new committee (including myself) met for the first time in May and decided to outsource the website. Steve Dicks (Electronic Communications Officer) and myself were tasked to develop a specification. Unfortunately this turned out to be more difficult than either of us expected.

In August I asked a friendly company how much they would charge to write such a specification for us. The committee decided this was too much and instead a "new web" sub-committee was formed. This produced a short specification that was put to the main committee in November. The main committee agreed the specification and gave us authority to seek tenders.

December was a busy month for everyone but those of us in the new-web group still found time to draw up a tender schedule, make final revisions to the documents and a couple of days after Christmas I mailed a request for tenders to `accu-contacts` and several firms who had already expressed interest in the work.

We have tried to keep the process as open as possible. We've had debates amongst ourselves about who should be allowed to bid and one person withdrew from the new-web group lest his position be questioned. Originally we intended to advertise the tender in the pages of C Vu itself, this hasn't happened for two reasons. First and foremost, given the schedule we are running to this would inject a two-month delay into the process. We didn't feel this was reasonable, not only would we lose momentum but we would lose any chance of having anything ready for the April conference. (It's worth noting that our schedule is dictated by the fact that we are all volunteers who fit this work around our real jobs. If we had had the time to do it quicker we would have done.)

Secondly, the new-web group (and myself specifically) feel that the companies we have asked to bid – and the mailing to `accu-contacts` – will give us plenty of competition in the bidding process. Every extra bid means work for the committee and potentially slows down the signing of the contract.

At the moment I'm hopeful that by the time you read this we will have a shortlist of bidders and be well on the way to signing a contract. I'd still like to think we'll have some new pages on show for the April conference but have to admit this is looking more unlikely.

One change we have asked bidders to look into is the possibility of raising revenue from the website. The obvious example is links from book reviews to booksellers' websites. While nobody on the committee wants to see the site overrun with advertising, we haven't forgotten the support given to the ACCU by the likes of Blackwells and PC-Bookshops. The objective is to lessen the costs to the ACCU and therefore your membership fees.

So there you have it. I know 8 months may seem a long time to do this sort of work – it does to me! – but it is surprising just how quickly it goes.

NOTIFICATION OF AGM

Alan Bellingham <secretary@accu.org>

Notice is hereby given that the 17th Annual General Meeting of The C Users' Group (UK) publicly known as ACCU will be held at 12:00 pm on Saturday 23rd April 2005 at the The Randolph Hotel, Beaumont Street, Oxford, OX1 2LN, United Kingdom.

Agenda

1 Apologies for absence
2 Minutes of the 16th Annual General Meeting
3 Annual reports of the officers
4 Accounts for the year ending 31st December 2004
5 Appointment of Auditor
6 Election of Officers and Committee
7 Other motions for which notice has been given.
8 Any other Annual General Meeting Business (To be notified to the Chair prior to the commencement of the Meeting).

The attention of attendees under a Corporate Membership is drawn to Rule 7.8 of the Constitution:

> '... Voting by Corporate bodies is limited to a maximum of four individuals from that body. The identities of Corporate voting and non-voting individuals must be made known to the Chair before commencing the business of the Meeting. All individuals present under a Corporate Membership have speaking rights.'

Also, all members should note rules 7.5:

> 'Notices of Motion, duly proposed and seconded, must be lodged with the Secretary at least 14 days prior to the General Meeting.'

7.6:

> 'Nominations for Officers and Committee members, duly proposed, seconded and accepted, shall be lodged with the Secretary at least 14 days prior to the General Meeting.'

and 7.7:

> 'In the absence of written nominations for a position, nominations may be taken from the floor at the General Meeting. In the event of there being more nominations than there are positions to fill, candidates shall be elected by simple majority of those Members present and voting. The presiding Member shall have a casting vote.'

For historical and logistical reasons, the date and venue is that of the last day of the ACCU Spring Conference. Please note that you do not need to be attending the conference to attend the AGM.

At the last two AGMs, there has been some protracted discussion that has led to a time overrun. For this reason, rather than be scheduled for the second hour of the lunch break, the meeting is scheduled for the first hour. On the other hand, there should be less that needs discussion, so the extra time should not be needed.

For more information about the conference, please see the web page at `http://accu.org/conference`

As far as business is concerned, we don't yet have any extraordinary items, and at the time of writing, all current officers and other committee members are willing to stand again. There is somewhat of a tradition that nominations are made from the floor during the AGM as per rule 7.7 rather than as per rule 7.6. However, it is still within the powers of the membership to select an entirely new committee should it so desire.

One innovation we intend this year is to prepare the reports, together with the minutes of last year's meeting, and distribute them by email to all members a fortnight before the meeting. This should allow attenders to study them and prepare questions in advance, rather than having to do so on the spot. Although there may be some reports that do not arrive before the day, this should help streamline the business and let members concentrate on what matters.

We will also be placing these documents on the website, which should from now on contain all AGM documents for this and subsequent years.

Dialogue

Student Code Critique Competition 32

Set and collated by David A. Caabeiro
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

Besides wishing you all (unpunctually) a prosperous 2005, a special thanks to those who participate in this competition for their support.

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

Editor's Note

Due to the large number of entrants this time, I have found this a particularly difficult SCC to judge; the entries are that good (as you'll see!). As we're at the start of a new year, **two** prizes will be awarded for this SCC.

Thank you to everyone who entered for making this the most difficult SCC to judge, and oddly enough, the most enjoyable – keep sending those entries in folks!

Student Code Critique 31 Entries

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.

From Tim Penhey <Tim.PENHEY@rbos.com>

I do have to admit that on first scan of the code, I didn't notice the error. It was only when typing the code in that I noticed it.

One thing that I often do when working with numbers is to actually transpose the numbers into the code and look for errors. It is very easy to get caught by "off by one" errors, however this is not one of those times. Firstly let's look at the first sequence of numbers:

```
80 84 60 100 90
```

Now put these into the code replacing the test variables (let's replace the comments too):

```
int lowest = 80;
// 80 is the lowest number unless
if (84 < 80) lowest = 84;
// 84 is smaller than 80 unless
if (60 < 84) lowest = 60;
// 60 is smaller than 84 unless
if (100 < 60) lowest = 100;
// 100 is smaller than 60 unless
if (90 < 100) lowest = 90;
// 90 is smaller than 100.
```

Now we can easily see that the logic is flawed. Checking the adjacent value will not choose the smallest. The simplest change that will get the code to work is the check against the current lowest value instead

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

Now to comment on style...

- maybe it is just me, but I prefer to have the comment above the code that it is referring to, not below. Perhaps it is just that I like to know the intent before I see the code. *[Production Editor – that was my fault, comments were at the end of the lines, and as there wasn't room within the standard layout I inserted line breaks with indentation, which is standard procedure for the ACCU journals. This is the only layout change I ever make to the code critique problem]*
- `<iomanip>` is not needed as the only manipulator being used is `endl`, and that is defined in `<ostream>` (which is included through `<iostream>`.
- use appropriate variable names. `average` in the example is not the average but the sum, and `averagescore` is the sum less the lowest.

What is going to happen if we now need to test six values, or ten, or even a class of 30? The algorithm being used is not particularly extensible.

One solution is to calculate the sum and the lowest while entering value. However when doing this we now have to handle the boundary cases where the user may enter any number of values. No values obviously has average of zero, while one value is by definition also the lowest, and the average of the rest (no values) is zero, so the average calculation is only valid where the number of entered values is greater than one. Here is an example that accumulates on the fly:

```
#include <iostream>
#include <limits>
using namespace std;
int main() {
  cout << "This program will gather "
       << "test scores and drop the\n"
       << "lowest score, giving you the "
       << "average of the remaining.\n\n"
       << "Enter test scores.  Terminate "
       << "the last score with a period.\n";
```

```
  int sum = 0;
  int count = 0;
  int value;
  int lowest = numeric_limits<int>::max();
  while(cin >> value) {
    if(value < lowest) lowest = value;
    sum += value;
    ++count;
  }
  int average = 0;
  if(count == 0)
    cout << "No entries entered\n";
  else if(count == 1)
    cout << "Only one value entered, "
         << "so it is the lowest value.";
  else
    average = (sum - lowest) / (count - 1);
  cout << "Average: " << average << endl;
  return 0;
}
```

`numeric_limits` is used to define the initial value of the `lowest` variable. Since any other integer value will be equal or less than this, then any value typed in as the first value will set the `lowest` to be that. Subsequent values are then checked against the current `lowest`.

The other "trick" in the code is using the `fail` flag on `cin` to terminate the entry loop. The `fail` flag happens when we ask to stream into an integer and the stream contains a non-whitespace non-integer value, hence the terminating the last score with a period.

Another solution is to use standard containers and use algorithms like `accumulate` to sum the values, but this I'll leave as an exercise for the reader (or other submitters).

From Thaddaeus Frogley <codemoney_uk@mac.com>

The code does not work because of a simple logical error, and not a language specific problem. Each `if` statement is evaluated "locally" and in effect ignores the preceding work done. Thus, as the student as observed, if `test5` contains a smaller value than `test4` then `lowest` is assigned `test5`, irrespective of the results of the preceding tests. The straightforward fix is to change the sequence of `if` statements to compare each time against the current `lowest` value, then I would expect the code to work.

This of course ignores issues of extensibility, design and style, but for a student of this level I would consider it more important to understand the logical flow required to solve the problem at this simple level. Ultimately knowledge of the standard library is second to a solid grasp of constructing a logical sequence of steps to solve a problem programatically. For future reading I would advise reading up on arrays and containers, and the `std::sort` algorithm. Constant use of `std::endl` vs \n would also be nice.

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

The first thing to do is to answer the student's question – they want to know what is wrong with the code. The answer is the sequence of comparisons of adjacent test values: the result of each stage (the new value of lowest) needs to be passed into the next comparison.

Simply change the sequence to:

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

This fixes the code – but there are several other things worth commenting on. Firstly, this sort of code cries out for a loop! In order to do this we want an array variable rather than 5 separate variables. C++ comes with a suitable collection object: the `vector`. So we can replace the list of variables `test1` to `test5` with:

```
std::vector<int> test(5);
```

then the input, the test and the addition can all be done by using loops – this is immediately generalisable to cases where you've got more (or less) than 5 numbers to process.

To make the code more robust, this should be the last time we use the hard-coded number '5' – the rest of the program can use the size of the `vector` to ensure it copes with changes to this number.

```
for(int i = 0; i != test.size(); ++i)
  cin >> test[i];
int lowest = test[0];
for(int i2 = 1; i2 != test.size(); ++i2)
  if(test[i2] < lowest) lowest = test[i2];
          // get the lowest number
average = 0;
for(int i3 = 0; i3 != test.size(); ++i3)
  average += test[i3];
```

I'd also like to change the names of the variables – the names don't match the contents. For example, `average` and `averagescore` both contain totals, not averages!

It is good to pick meaningful names for variables, but important to remember to keep the names of the variables consistent with their usage to avoid leading the reader of the code astray.

Lastly, we can get rid of some of the loops by using algorithms provided by the standard library. We can use `min_element` to find the lowest value and `accumulate` to perform the sum.

My final version of the program looks like this:

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;

int main() {
  cout << "This program will gather "
       << "five test scores and\n";
  cout << "drop the lowest score, "
       << "giving you the average\n" << "\n";
  cout << "Please enter Five Test scores\n";
  std::vector<int> test(5);
  for(int i = 0; i != test.size(); ++i)
    cin >> test[i];
  int lowest = *min_element(test.begin(),
                            test.end());
  int totalscore = std::accumulate(test.begin(),
                                   test.end(), -lowest);
          // total score (minus the lowest grade)
  int divide = totalscore / ( test.size() - 1);
     // average grade is then total divided by (n-1)
  cout << divide << endl;
          // final grade after division
  return 0;
}
```

My hope is that the resultant code is easy to understand and does the task well enough. This code may be slower to write than the original code the first time, but with practice it will become second nature. The time spent learning to do this is also repaid in the reduction in bugs.

From Richard Wheeler <acht85@ukgateway.net>

OK, so I am not a programmer but every now and then I get to look at code, especially when I want to know what actually happens in a program. (Especially when I do not believe the documentation or there is no documentation). Thank heavens this code compiles and runs – I don't think I could handle obscure syntax problems. Anyway, full marks to the student for carrying out sufficient testing to identify that there is a problem. Now, looking at the student's code a number of points to comment on jump out at me. I will treat them in order so the trivial are mixed in with the more important – but then with just a short piece of code it is not clear how a trivial comment would scale into a programme with thousands of lines of code.

1. Input validation

There is no validation of the input. This is not the student's issue but it is worth a remark as I would expect significant further programming effort to ensure that the input is properly validated (sensible means of stopping the programme if run in error, the correct number of exam grades are entered, each is a number, each within sensible bounds, meaningful error messages to the user, etc).

2. Choice of variable names

I do not like variable names which look like reserved words – `average` and `divide` (and further down `lowest`) make me uneasy. These names look as though they are self-documenting but I would prefer something like `average_score` and `lowest_score` as better self-documenting names. [See also comment 7 below].

3. Consistent programming style

I look for a consistent programming style as this should speed up comprehension of code. Here we have a block where all the `int` variables are defined except for the single variable `lowest` which is defined later. I may be making a mountain out of a molehill in this case but when a programme extends to thousands of lines then consistency is important.

4. Program logic

This is the guts of the student's problem. The process should be to set up a placeholder for the lowest grade. This is given any grade as its initial value (and the first grade is as good as any). Then each grade is compared in turn to the placeholder and if the grade has a lower value the placeholder is reset to that grade. From this description it follows that the `if` statements should read

```
if(testn < lowest)lowest = testn;
```

for each value of `n` from 1 to 5.

5. Generalisation (1)

I am against unnecessary generalisation but in this case I think it clarifies the program logic. If we allow for a variable number of tests we can generalise the logic into a `for` loop using an array of test scores. Something on the following lines should do

```
lowest_score = test_score(1);
for(int i = 1, i <= number_of_tests, i++) {
  if(test_score(i) < lowest_score)
    lowest_score = test_score(i);
}
```

(The student might want to change the input process to start with obtaining the number of grades, so giving a value to the new variable `number_of_tests` – there are other approaches which could be used).

A bit of cleverness might be to avoid the first iteration in the loop as this is unnecessary. But that, in my opinion, tends to obscure the logic process.

6. Comments should be meaningful (1)

I found the comments against all the `if` statements were not very helpful. In fact one reason for generalising to the `for` loop above was to think how the `for` loop should be commented and compare this to the existing comments. In fact I would not bother to comment the `for` loop at all.

7. Variable names are misleading

The variable `average` is set to a total and is not an average at all. As variable names, `average` and `averagescore` give no clues as to the different (or same) data entities they refer to. `divide` gives no clues at all to what it means. I would suggest that the following are more meaningful values

`total_score` for `average`
`adjusted_total_score` for `averagescore`
`final_grade` for `divide`

8. Comments should be meaningful (2)

The comments in the section of code which calculates the final grade are wrong. (Note the distinction – the variable names are misleading, the comments are wrong). With good variable names (such as those suggested above) I think that comments are unnecessary.

9. Magic numbers

The evaluation of `divide` uses the "magic number" 4. This comment is scarcely worth bothering about in this specific example but is something to be aware of if the programme is generalised to handle any number of grade scores.

10. Generalisation (2)

Following on from the generalisation of the logic there needs to be corresponding generalisation of calculation of the `total_score`. I would use

```
total_score = 0;
for(int i = 1, i <= number_of_tests, i++) {
  total_score = total_score + test_score(i);
}
```

Now there are two `for` loops. Perhaps compiler optimisation can roll these up into one. I would do this explicitly and, at last, include a comment

```
// calculate lowest score and total of all scores
lowest_score = test_score(1);
total_score = 0;
for(int i = 1, i <= number_of_tests, i++) {
  if(test_score(i) < lowest_score)
    lowest_score = test_score(i);
  total_score = total_score + test_score(i);
}
```

and, just for comparison, we could save the first iteration of the loop with

```
// calculate lowest score and total of all scores
lowest_score = test_score(1);
total_score = test_score(1);
for(int i = 2, i <= number_of_tests, i++) {
  if(test_score(i) < lowest_score)
    lowest_score = test_score(i);
  total_score = total_score + test_score(i);
}
```

But, as I said before, I think this is unnecessary cleverness which obscures what is happening.

11. User interface niceties

The student has taken care on input to explain to whoever runs the program what the program does. But for the results, the adjusted grade value is all that appears. I think it would be an improvement to have a line which explains the results. Something like

```
cout << "The adjusted grade for your "
     << "five test scores is \n";
```

Finally there are a number of other points which make me feel uneasy. I would want to discuss these with the student's tutor / mentor / project leader as to whether the student needs additional help. These are:

- The student's initial description of the problem does not come over as fluent English. Is the student a native English speaker? (I have worked on multi-national projects with English as the project language. I found that fluent English speakers were still worried that as "non-native speakers" they could misunderstand the subtleties of requirements etc). *[I don't know where he is from. David]*
- I wonder whether the student's incorrect comments about calculating the average etc are a problem with the English language or a problem with understanding the code.
- Whilst I give the student full marks for identifying the problem through testing, it is not that difficult to step through this code line by line with the two sets of test values and work out what is going wrong.
- I could quite well be over-reacting. After all, we all have off-days. However, I think it would be worthwhile to find out a bit more about the student and not stop after completing a coding criticsm.

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

To start with the question as asked, the code does not always find the lowest value because the value is only compared with its neighbours, not with the lowest value found so far. Obviously this can result in the value selected not always being the lowest.

Once this is fixed, I would expect the the program to work as expected, provided the input is exactly as expected, but it may not handle any unexpected variations in input gracefully and any extensions, e.g. to handle a different number of inputs, will require changes to the code.

After fixing the bug, I would start further improvements to the code by providing a more flexible input mechanism that allows the variable numbers of inputs. I would also break the program down into functions that will handle the individual tasks, so I might as well make this change by way of a new function, which I shall call `input_data`. I have defined `input_data` as:

```
std::vector<int> input_data(istream &in) {
  std::vector<int> data;
  while(!in.eof() && in.good()) {
    int val;
    in >> val;
    if(in.good()) {
      data.push_back(val);
    }
  }
```

```
    if(!in.eof()) {
      // We should only get here if there
      // has been an error on the stream
      cerr << "Input error reading data"
           << endl;
      exit(1);
    }
    return data;
  }
```

[Watch out here. This function contains some pitfalls. For instance, what happens when EOF is right after the last number? Is it pushed into the vector? David]

The first thing you should notice is that this function returns a `vector` of `int`s. A `vector` is a structure provided the standard template library, which is capable of handling a variable number of elements. As `vector`s are defined by templates they may be used to contain any type you choose, in this case `int`s. Bear in mind that other list structures are available and a `vector` may not always be the best choice.

You may also notice that the input stream is left as a parameter, so that the function may read data from any input stream, e.g. a file, instead of being restricted to reading from `cin`.

Also needed is a way of indicating that the end of input has been reached. I have chosen to request an end of file character to indicate the end of the list. Again, this is not the only possible choice and a non technical audience may prefer something like entering the word "end", but this approach is simpler to code.

Another important point is that there was no error checking in your existing code. This function checks for errors and exits when there is an error on the stream. You should consider whether this code should continue when an error has occurred, in which case it will need some action to reset the stream to a good state before it will be able to read further.

I have updated the prompts to read

```
cout << "This program will gather test "
     << "scores and drop the lowest" << endl
     << "score, giving you the average of the "
     << "remaining scores" << endl << endl
     << "Please enter your test scores" << endl
     << "When all scores have been entered "
     << "please terminate the list" << endl
     << "with an end of file character "
     << "(^D in Unix, ^Z in Windows)" << endl;
```

It is important that when you modify a program, comments and text shown to the user are updated to match. If this is not done at the same time it will often be forgotten.

Many will argue that the use of `endl` for all line endings is inefficient. I prefer to always use `endl` for consistency, unless a program has a serious I/O performance problem.

The next task is to find the lowest value, which can be done by a simple function, but rather than writing our own, we can see that there is a suitable function already provided in the STL called `min_element`, which we can use:

```
std::vector<int>::iterator lowest
   = min_element(data.begin(), data.end());
```

Similarly we can use the STL function `accumulate` to produce an initial sum. To avoid confusion you really should not use the name "average" for the initial sum, that's somewhat confusing and your other names are similarly poorly chosen. I would suggest something like:

```
int sum = accumulate(data.begin(), data.end(), 0);
int adjusted_sum = sum - *lowest;
int result = adjusted_sum / (data.size()-1);
```

Other things I would change include changing `using namespace std` to using specific items from the `std` namespace and declaring variables where they are used instead of declaring them all at the start of the function. This leaves the final program looking like:

```
#include <vector>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <numeric>
using std::istream;
using std::vector;
using std::cin;
using std::cout;
using std::cerr;
using std::endl;

std::vector<int> input_data(istream &in) {
  std::vector<int> data;
  while(!in.eof()) {
    int val;
    in >> val;
    if(in.good()) {
      data.push_back(val);
    }
  }
  if(!in.eof()) {
    // We should only get here if there
    // has been an error on the stream
    cerr << "Input error reading data" << endl;
    exit(1);
  }
  return data;
}

int main() {
  cout << "This program will gather test "
       << "scores and drop the lowest" << endl
       << "score, giving you the average of the "
       << "remaining scores" << endl << endl
       << "Please enter your test scores" << endl
       << "When all scores have been entered "
       << "please terminate the list" << endl
       << "with an end of file character "
       << "(^D in Unix, ^Z in Windows)" << endl;
  std::vector<int> data = input_data(cin);
  std::vector<int>::iterator lowest
       = min_element(data.begin(), data.end());
  int sum = accumulate(data.begin(), data.end(), 0);
  int adjusted_sum = sum - *lowest;
  int result = adjusted_sum / (data.size()-1);
  cout << result << endl;
  return 0;
}
```

From Margaret Wood <margaretwood@pocketmail.com.au>

I'm sure lots of people can tell you why the output of your program depends on the order you enter the numbers, but I think it will be more useful to help you work it out for yourself. You can do this by looking at the values of the variables as you progress through the code. Here is a modified version of your code, I have added some more calls to `cout`, to show the value of `lowest` after each comparison.

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

```
    cout << "lowest is " << lowest << endl;
    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 << endl;
    cout << divide << endl;
          // final grade after division
    return 0;
  }
```

If you run this version, with the values `80,84,60,100,90` it will print out

```
lowest is 80
lowest is 80
lowest is 60
lowest is 60
lowest is 90
81
```

Why has `lowest` increased to `90`? What was the program doing just before the change? It was comparing `test4` and `test5`. Since `test5` is smaller than `test4` the value of `lowest` is reset to `90`. However you only want `lowest` to be reset if the new value (`test5`) is smaller than `lowest`. Here is a modified version of your code which should give the answer you want.

```
#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;
  cout << endl;
  int lowest = test1;
        // test 1 is the lowest number unless
  if (test2 < lowest) lowest = test2;
        // test 2 is smaller than test 1 unless
  if (test3 < lowest) lowest = test3;
        // test 3 is smaller than test 2 unless
  if (test4 < lowest) lowest = test4;
        // test 4 is smaller than test 3 unless
  if (test5 < lowest) 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;
}
```

Now I would like to mention a few other things I noticed while looking at your code.

Some of the variable names are misleading. The variable you call `average` is in fact the total, `averagescore` is a modified total – perhaps you could call it `modTotal` – and `divide` is the average.

I'm not sure why you have included `iomanip` – the code works without it. *[Maybe he thought (mistakenly) it would be needed by* `std::endl`*. David]*

If this was my code I would calculate the average as a float. For the 5 numbers you entered the average is in fact 88.5, so presenting 88 as your answer is fair enough, but if you had chosen say, `80`, `84`, `60`, `91`, `100`, the average would be 88.75 and in many circumstances it would be better to round the answer up to 89. However I don't know the precise details of the problem you were asked to solve, so let's leave it as it is for now.

Finally I'd like to look at some ways of making your code more versatile. At the moment it requires exactly 5 inputs, it is relatively simple to make it work with any number of scores greater than one.

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

int main() {
  int inValue, total, lowest, count, average;
  cout << "This program will gather two "
          "or more test scores and\n";
  cout << "drop the lowest score, giving "
          "you the average\n" << "\n";
  cout << "Please enter at least two test scores\n";
  cout << "End your input with a single full stop\n";
  cin >> inValue;
  lowest = inValue;
  total = inValue;
  count = 1;
  while(cin >> inValue) {
    ++count;
    total += inValue;
    if(inValue < lowest) lowest = inValue;
  }
  if (count < 2) {
    cout << "This program requires at least "
         << "two values" << endl;
  } else {
    average = (total - lowest)/(count-1);
    cout << average << endl;
  }
  return 0;
}
```

Just one more improvement to go! In real life you may not have a user typing data in at the prompt – it may have come from a database, spreadsheet or a special user interface. So let's make your program into a function that returns the answer to whatever called it. We will assume that the calling program has already put the values into a `vector`.

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

int myAverage(vector<int> values) {
  int total, lowest, average;
  total = 0;
  lowest = values[0];
  for(vector<int>::iterator it = values.begin();
      it != values.end(); ++it) {
    total += *it;
    if(*it < lowest) lowest = *it;
  }
  average = (total - lowest)/(values.size()-1);
  return average;
}
```

From Nevin Liber `<nevin@eviloverlord.com>`

The question as stated is slightly wrong. Here is the correction:

```
80 84 60 100 90 ==> 81 (incorrect!)
80 84 100 60 90 ==> 88 (correct)
```

[Good, you verified the student's statement, which was probably a typo. David]

Improvement #1: Fix the bug

The bug is in the `if` statements: instead of comparing adjacent test scores, each test score should be compared against `lowest`. The corrected code:

```
#include <iostream>
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" << "\n";
  cout << "Please enter Five Test scores\n";
  cin >> test1 >> test2 >> test3 >> test4 >> test5;
  int lowest = test1;
  if (test2 < lowest) lowest = test2;
```

```
    if (test3 < lowest) lowest = test3;
    if (test4 < lowest) lowest = test4;
    if (test5 < lowest) lowest = test5;
    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;
  }
```

Style: not bad, actually. Only a few minor nits.

1. Don't include `iomanip`, since nothing in it is being used.
2. Be more consistent with whitespace.
3. Each variable declaration should be on a separate line.
4. Each variable declaration should be as close to its use as possible.
5. Put curly braces around the statements inside `if`s.
6. Pick better variable names (eg: `average` should really be `sum` or `total`). Note: Since I am trying to build upon the student's solution, I will not be changing his variable names even when I know that they aren't quite accurate.

Design: once the bug is fixed, his code gets the job done, albeit in a brute force sort of way.

Extensibility: here is the real shortcoming of this code. The number of test scores is fixed at 5. The number of scores we drop is fixed at 1. We can, of course, do better.

Improvement #2: Variable number of test scores

When I first read this problem, it screamed out to me that we should be using algorithms over a collection of test scores. Without changing the structure of the original solution too much, I came up with:

```
  #include <algorithm> // for std::min_element
  #include <deque>
  #include <iostream>
  #include <iterator>  // for std::distance
  #include <numeric>   // for std::accumulate
  typedef std::deque<int> Scores;
  int AverageTestScore(Scores::iterator first,
                       Scores::iterator last) {
    int lowest = *std::min_element(first, last);
    int average = 0;
    average = std::accumulate(first, last, average);
    int averagescore = average - lowest;
    int divide = averagescore
                 / (std::distance(first, last) - 1);
    return divide;
  }
  int main() {
    std::cout << "This program will gather test scores "
         << "and\ndrop the lowest score, giving you "
         << "the average\n\nPlease enter Test scores, "
         << "followed by \"end\"" << std::endl;
    // store all the ints in cin into scores
    Scores scores;
    int test;
    while(std::cin >> test) {
      scores.push_back(test);
    }
    // Need at least two elements for this calculation
    if(1 < scores.size()) {
      int divide = AverageTestScore(scores.begin(),
                                    scores.end());
      std::cout << divide << std::endl;
    }
    else {
      std::cerr << "The number of scores needed is "
           << "at least 2; you only entered "
           << scores.size() << std::endl;
      return 1;
    }
    return 0;
  }
```

Highlights:

1. I use a `deque` to store the elements. I could have just as easily used a `vector` or even a `list`. It is hard to make the tradeoffs between them without running this on real data and profiling.
2. Unlike the original solution, there is now a potential error condition when too few scores are given to perform the calculation. I had to add code to handle this situation.
3. I use the `min_element` algorithm to get the lowest score. Since I know there are at least two elements in `scores`, I also know that I can legally dereference the iterator returned from `min_element`.
4. I use `accumulate` to calculate the `average`. A better variable name would have been `sum` or `total`, but I was trying to keep this as close to the original solution as possible.
5. Both `min_element` and `accumulate` do not modify the collection, and they are "linear" (O(N)) algorithms.
6. Since there are at least two elements in `scores`, the division performed in `divide` will never result in a divide by 0 error.

Improvement #3: Variable number of low scores dropped

In order to do this, we need to sort the scores. There are a variety of different ways to do this. We could store them in a `multiset`. We could sort the entire collection. But this is overkill (in the sense of greater than linear time algorithms, such as Nlog(N)), as we don't need to sort the entire collection; we just need to group the low scores away from the high scores. And there just happens to be an algorithm which does what we want: `nth_element(...)`. What it does is put the nth element in the correct position as if the whole thing were sorted, and all the elements before the nth position are <= the nth element, and all the elements after the nth position are >= the nth element. Plus, `nth_element(...)` runs in linear time on average. However, `nth_element(...)` requires random access iterators, thus limiting the collection types to `vector` or `deque`, but not `list`.

```
  #include <algorithm> // for std::nth_element
  #include <stdlib.h>  // for exit
  #include <deque>
  #include <iostream>
  #include <numeric>   // for std::accumulate
  typedef std::deque<int> Scores;

  int NonnegativeIntFromCin() {
    int value;
    if(!(std::cin >> value) || value < 0) {
      std::cerr << "Next time, please enter a "
           << "non-negative integer" << std::endl;
      exit(1);
    }
    return value;
  }
  int AverageTestScore(Scores::iterator first,
       Scores::iterator low, Scores::iterator last) {
    // Put the lowest test scores in [first, low)
    std::nth_element(first, low, last);
    // Sum all the high [low, last) test scores
    int averagescore = 0;
    averagescore = std::accumulate(low, last,
                                   averagescore);
    int divide = averagescore / (last - low);
    return divide;
  }
  int main() {
    // Enter the number of low test scores to drop
    std::cout << "This program will gather test scores "
         << "and\ndrop the lowest score, giving you "
         << "the average\n\nPlease enter the number of "
         << "low Test scores to drop" << std::endl;
    int lowdropped = NonnegativeIntFromCin();
    // enter the test scores
    std::cout << "Please enter Test scores, followed "
         << "by \"end\""<< std::endl;
    Scores scores;
    int test;
    while(std::cin >> test) {
      scores.push_back(test);
    }
```

```
  // need at least 1 more score than number dropped
  if(lowdropped < scores.size()) {
    int divide = AverageTestScore(scores.begin(),
         scores.begin() + lowdropped, scores.end());
    std::cout << divide << std::endl;
  }
  else {
    std::cerr << "The number of scores needed is "
         << "at least " << lowdropped + 1
         << "; you only entered " << scores.size()
         << std::endl;
    return 1;
  }
  return 0;
}
```

The original functionality can be gotten by calling:

```
AverageTestScore(scores.begin(),
                 scores.begin() + 1, scores.end());
```

Improvement #4: Variable number of high scores dropped

That is another predictable extension, and it isn't hard to add. Basically, do the `nth_element(...)` trick on the high side of the collection as well, taking care not to resort the lowest scores.

```
#include <algorithm> // for std::nth_element
#include <stdlib.h>  // for exit
#include <deque>
#include <iostream>
#include <numeric>   // for std::accumulate
typedef std::deque<int> Scores;

int NonnegativeIntFromCin() {
  int value;
  if(!(std::cin >> value) || value < 0) {
    std::cerr << "Next time, please enter a "
         << "non-negative integer" << std::endl;
    exit(1);
  }
  return value;
}
int AverageTestScore(Scores::iterator first,
       Scores::iterator low, Scores::iterator high,
       Scores::iterator last) {
  // Put the lowest test scores in [first, low)
  std::nth_element(first, low, last);
  // Put the middle test scores in [low, high)
  std::nth_element(low, high, last);
  // Sum all the middle [low, high) test scores
  int averagescore = 0;
  averagescore
       = std::accumulate(low, high, averagescore);
  int divide = averagescore / (high - low);
  return divide;
}

int main() {
  // Enter the number of low test scores to drop
  std::cout << "This program will gather test "
       << "scores and\ndrop the lowest score, "
       << "giving you the average\n\nPlease enter "
       << "the number of low Test scores to drop"
       << std::endl;
  int lowdropped = NonnegativeIntFromCin();
  // Enter the number of high test scores to drop
  std::cout << "Please enter the number of high "
       << "Test scores to drop" << std::endl;
  int highdropped = NonnegativeIntFromCin();
  // enter the test scores
  std::cout << "Please enter Test scores, "
       << "followed by \"end\"" << std::endl;
  Scores scores;
  int test;
  while(std::cin >> test) {
    scores.push_back(test);
  }
  // need at least 1 more score than number dropped
  if(lowdropped + highdropped < scores.size()) {
    int divide = AverageTestScore(scores.begin(),
         scores.begin() + lowdropped,
         scores.end() - highdropped, scores.end());
    std::cout << divide << std::endl;
  }
  else {
    std::cerr << "The number of scores needed is "
        << "at least " << lowdropped + highdropped + 1
        << "; you only entered " << scores.size()
        << std::endl;
    return 1;
  }
  return 0;
}
```

The original functionality can be achieved by calling

```
AverageTestScore(scores.begin(), scores.begin() + 1,
                 scores.end(), scores.end());
```

As you can see, this isn't much different than my solution for improvement #3. Since it didn't involve much extra engineering or testing, I felt it was worth adding this functionality. Your mileage may very.

At this point I am done. There are other ways to extend this code (for instance, making `AverageTestScore` a templated function instead of hard coding its parameters); however, they tend to get in the way of readability and understandability for a student first getting started with the language (my target audience), and I'll leave those as an exercise for the reader.

From Chris Main <chris@chrismain.uklinux.net>

"It's not fair!"

Inspector Slack was dozing peacefully in his favourite armchair after his Christmas dinner when he was interrupted by the familiar and unmistakeable sound of his children bickering.

"It's that computer game Sergeant Lake gave us for Christmas. Joy scored 80, 84, 60, 100 and 90 and got a grade of 88. I scored 80, 84, 100, 60 and 90 and only got 81", complained Matthew.

"Why is that unfair?"

"Because I got exactly the same scores, just in a different order".

"Just like those gloves I knitted for little Tommy Smith".

Slack ignored this remark from his house guest, a little old lady knitting quietly in the corner, and proceeded to vent his fury on his sergeant.

"Lake! I told him that Open Source Software would be no good. Bungling amateurs!".

"Did you say Open Source, Inspector?" inquired Miss Marple. "Doesn't that mean *anyone* can read the program? I should be most interested to see it, though I don't suppose I shall understand it."

Before Slack had time even to think "interfering old woman", Matthew had downloaded the source code from the internet and built it.

"See. If I enter my scores it gives me 81, but if I enter Joy's she gets 88."

"Oh dear!" exclaimed Miss Marple. "Do we have to type in the numbers every time we want to try it out?"

"I know," said Joy, "let's turn it into a function that can use any input stream. Then we can feed it test data in string streams and the real thing from standard input".

The children typed away busily, setting up a test function that used an `assert` to check the result of calculating a grade. With this rearrangement they could easily add other test cases too:

```
namespace {
  int CalculateGrade(istream &stream) { ... }
  struct TestCase {
    const char *scores;
    int grade;
  };
  void CheckCalculateGrade(const TestCase &testCase) {
    istringstream stream(testCase.scores);
    assert(CalculateGrade(stream) == testCase.grade);
  }
  void TestCalculateGrade() {
    const TestCase testCases[]
      = { { "80 84 60 100 90", 88 },
          { "80 84 100 60 90", 88 } };
    const unsigned count
      = sizeof(testCases)/sizeof(testCases[0]);
```

```
    for_each(testCases, testCases+count,
             CheckCalculateGrade);
  }
}
int main() {
  TestCalculateGrade();
  cout << "This program will gather five test "
       << "scores and\n";
  cout << "drop the lowest score, giving you the "
       << "average\n" << "\n";
  cout << "Please enter Five Test scores\n";
  cout << CalculateGrade(cin) << endl;
  return 0;
}
```

When they tried it out, it duly reported an assert failure.

"How thoughtful," said Miss Marple with approval, "the program prints out what it is supposed to do. Do all programs do that?"

"Sadly not," sighed Matthew.

"It should really only be output when a command line option such as `/?`, `-h` or `–help` is set," added Inspector Slack with a punctilious air of authority.

"Dear me, my eyesight is poor these days, I seem to be seeing double looking at this program," fussed the old lady as she adjusted her spectacles.

"It's not your eyes," replied Joy, "it's just that every line has a comment repeating what the previous line does."

"Well, my dears, let's get rid of all that. There's absolutely no point in stating the obvious."

Slack bristled as he felt sure that Miss Marple had glanced knowingly at him when making this last point, but now she was again scrutinizing the code with an expression of sweetness and innocence on her face.

"Ah, that's much clearer. Now, surely what is named an `average` is really a sum, and what is called `divide` is actually the average."

Matthew reworked the code. "You always manage to work out which people aren't who they say they are. I bet that fixes it." He ran the program, but it still failed. Slack allowed himself a smile of satisfaction; this problem demanded professional detection skills.

"I thought these computers were supposed to make tasks easier, but I notice you still have to add up the test scores in one big sum," observed Miss Marple.

"We could use `std::accumulate` instead," answered Joy, "but we have to put the scores in a container first, like a `vector`." Miss Marple wasn't quite sure what a `vector` was. Her nephew Raymond West had once taken her for a very fast drive in his sports car which she was sure was called a Vector. With this fond memory she encouraged Joy to make the change. From this it became apparent that the number 5 would make a useful constant for the input loop, and could be used in the average calculation.

"Such a pity," muttered Miss Marple as she considered the simplicity of the `accumulate` statement.

"What's a pity?" asked Matthew.

"I was thinking, if only there were a nice function already available for finding the lowest value, similar to `accumulate` for finding the sum".

"But there is, it's called `std::min_element`." Matthew replaced all the `if` statements with `min_element`. The first attempt failed to compile, then he remembered it returned an iterator rather than a value. After de-referencing it the code built. Even better, the tests ran successfully too.

"I've got it!" cried Inspector Slack, who had been working feverishly with pencil and paper.

"It's okay, Dad, Miss Marple's already fixed it," Joy informed him.

Crestfallen, Slack looked at their code:

```
namespace {
  const int scoreCount = 5;
  int CalculateGrade(std::istream &stream) {
    vector<int> scores;
    for(unsigned n = 0U; n != scoreCount; ++n) {
      int score;
      stream >> score;
      scores.push_back(score);
    }
    const int sum = accumulate(scores.begin(),
                               scores.end(), 0);
    const int lowest
       = *min_element(scores.begin(), scores.end());
    const int average = (sum-lowest) / (scoreCount-1);
    return average;
  }
}
```

"Yes, but that doesn't explain why the original code didn't work. You see, the `if` statements compare each value to the previous value, when they should compare each value to the current lowest value."

"How clever of you, Inspector," said Miss Marple. Slack beamed with pride.

"However," she went on, "it seems to me that the really interesting question is *why* the mistake occurred. The programmer says in, now what did Joy call them? oh, yes, in the comments that he is using 'the equation' to drop the lowest number. He must have either been given the wrong equation or, more likely, noted it down incorrectly. I remembered I once made a mistake copying a knitting pattern from Mrs McGillicuddy, and made a pair of gloves for little Tommy Smith where the fingers came out in the wrong order."

Seeing the look of disappointment on the Inspector's face, and feeling guilty for outwitting him whilst enjoying his hospitality, she made a proposal. "I should very much like to see one of your magic tricks, Inspector, I do so enjoy them."

"I've been working on sawing the lady in half. Perhaps you'd like to lie down in that box over there while I fetch my saw," suggested Slack, with just the slightest hint of menace.

From Ian Glover <ian@manicai.net>

First the bug, the code above only works if the numbers after the lowest value are in increasing order, so for instance 80, 84, 100, 60, 90 works because the sequence 60, 90 is increasing, but 80, 84, 60, 100, 90 does not as 60, 100, 90 is not an increasing sequence (the problem description is the wrong way round it terms of the output of these sequences). The simplest correction would be not to compare each value in the series to the previous but to compare it to the lowest found so far.

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

While this fixes the bug it does leave some aspects still wanting in the program.

To deal with some of the stylistic points first. The early declarations of `average`, `averagescore` and `divide` are unnecessary and should be shifted to where those variables are first defined. It would also be worth changing the name of `average` and `averagescore`, because the names do not match the meanings, perhaps `total` and `amendedtotal` respectively; `divide` could then be renamed `averagescore` which gives a better sense of its purpose. Another minor point is that the inclusion of `iomanip` is superfluous as nothing from this header is used. Personally I would also prefer to use a single `std::cout` reference for the printed block at the top, since this makes for fewer changes should we wish to send the output to a different stream in future (though this is a more marginal decision than the others).

A rather more major issue than those is the design of the program in that it does not easily allow extension. As more tests are added we would have to remember to update the code in seven places (the declaration of the test variables, the two references to five tests in the printed text, the `cin` statement, the comparison tests, the summation to produce the total score and the division to produce the average). The solution to this is to use one of the STL sequences to hold the scores.

While the initial thought might be to use `std::vector` or `std::list`, the arithmetic operations that we do on the sequence suggest a better choice in the form of `std::valarray`. This has convenient methods allowing us to find the minimum held value, the sum of the values and the length which are the three pieces of information we use. Implementing this change alters most of the program to produce something like:

```
#include <iostream>
#include <valarray>

int main() {
  const size_t number_of_scores = 5;
  std::valarray<int> scores(number_of_scores);
  std::cout << "This program will gather "
       << scores.size() << " test scores and\n"
       << "drop the lowest score giving you the "
       << "average\n\n"
```

```
            << "Please enter " << scores.size()
            << " test scores\n";
    for(size_t i = 0; i < scores.size(); ++i) {
      std::cin >> scores[i];
    }
    int averagescore = (scores.sum() - scores.min())
                          / (scores.size() - 1);
    std::cout << averagescore << std::endl;
    return 0;
  }
```

A couple of notes. As you can see I've changed things round so that the number of scores is only set in one place and everything after that checks the the `valarray` size. I've also got rid of the intermediate variables for calculating the average as the method names on `valarray` express what the intent of the formula accurately.

From Andrew Marlow <andrew@marlowa.plus.com>

The code is very close to working. There are two bugs: the first bug is in the code to calculate `lowest`. The code needs to compare successive grade results with the current value of `lowest`, like this:

```
int lowest = test1;
if (test2 < lowest) lowest = test2;
if (test3 < lowest) lowest = test3;
if (test4 < lowest) lowest = test4;
if (test5 < lowest) lowest = test5;
```

The code was comparing adjacent grades, which is why the bug was dependent on the order of the grades.

The other bug was in the calculation of the integer average. The average of 80, 84, 90 and 100 is 88.5, which is 89 to the nearest integer. The average should be calculated as a floating point number and then rounded to the nearest integer using the `ceil` function from `cmath`, thus:

```
divide = int(ceil(averagescore / 4.0));
```

The code could be left with these corrections and it would work but the code is not extensible; it only works for five grades. With the current approach, extending the program for a larger number of grades, say N, would result in code additions proportional to N.

There are a few minor points on style, such as inaccurate comments and misleading variable names, which can be taken care of whilst refactoring to make the code more extensible should a larger number of grades be required. The line:

```
const int scores_count = 5;
```

can be used to set the number of grades the program is to cope with. The `cout` statements at the start can use `scores_count` to say how many grades the program is for.

The grades can be held in an array, and functions `sum` and `lowest` used to return grade sum and lowest grade respectively. These functions have no need to belong to a class, so they can reside in the unnamed namespace.

A C-style array could be used to hold the grades, dimensioned to `scores_count` elements, but using a `vector` is better than using a C-style array for several reasons:

- once the `vector` is populated it knows its size so the size does not need to be given to the `sum` and `lowest` functions.
- the future addition of more complex functionality is easier with `vector` due to its rich interface.
- The use of fixed size arrays creates opportunities for memory problems such as accidentally subscripting out of range. `vector`s manage their memory automatically and provide natural ways to iterate over all the items contained.

The student's code had very misleading variable names used in the calculation. With the functions mentioned, the calculation can be done as follows:

```
int total = sum(test_scores);
int low = lowest(test_scores);
int average = int(ceil((total - low)
                       / double(scores_count-1)));
```

With clearer names the code becomes self-documenting and comments become unnecessary.

The program should annotate its output so that its meaning stands on its own. This is shown in the complete code for the amended program below:

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>

using namespace std;

namespace {
  int sum(const vector<int>& v) {
    int total = 0;
    for(vector<int>::const_iterator it = v.begin();
        it != v.end(); ++it)
      total += *it;
    return total;
  }
  int lowest(const vector<int>& v) {
    int result = v[0];
    for(vector<int>::const_iterator it = v.begin();
        it != v.end(); ++it)
      if(*it < result) result = *it;
    return result;
  }
}
int main() {
  const int scores_count = 5;
  cout << "This program will gather " << scores_count
       << " test scores and\n";
  cout << "drop the lowest score, giving you the "
       << "average\n" << "\n";
  cout << "Please enter " << scores_count
       << " test scores\n";
  vector<int> test_scores;
  for(int i = 0; i < scores_count; ++i) {
    int one_result;
    cin >> one_result;
    test_scores.push_back(one_result);
  }
  int total = sum(test_scores);
  int low = lowest(test_scores);
  int average = int(ceil((total - low)
                    / double(scores_count-1)));
  cout << "average = " << average << endl;
  return 0;
}
```

From Andrew Bache <andy@bache.eclipse.co.uk>

The problem. The program does not function as intended because the `if` tests are wrong. Each successive number tested should be compared against the current lowest number not the following number entered. The logic error is easily fixed by amending the `if` tests as follows:

```
int lowest = test1;
if (test2 < lowest) lowest = test2;
if (test3 < lowest) lowest = test3;
if (test4 < lowest) lowest = test4;
if (test5 < lowest) lowest = test5;
```

Having made this change the program will function as intended. Issues of extensibility, design and style were hinted at. I had better address them.

Style

Personally I thought the code as presented was well laid out and easy to read. It was clear to see what the author's intention had been, and therefore where he had gone wrong. Making code easy for others to read is the most important aspect of 'style' and on this count the code, as written, is not bad. Ideally we write code without bugs, but accepting that we all make mistakes, if code is written in a manner that clearly communicates *in English* what the intent is, then it is easier for both ourselves and others to spot the mistakes. The style could perhaps be improved in the following ways:

Variable names. The names `average`, `averagescore` and `divide` are wrong. Is is an arguable point whether we need these variables but if we are going to use them names should reflect meaning. The names `total_score`, `adjusted_total` and `adjusted_average` would be better. This follows from the point made about communicating our intent in English.

Declaration close to use: it is widely accepted that variables should be declared as close as possible to where they are used. The author does well in this respect with the declaration of `lowest`, which is initialised on declaration and then immediately used in the `if` tests. However the rest of the locally declared variables are all together at the start of `main` and in the case of `divide`, it is not used until almost the end of the function. The reason for this is again to make the code easier to read. By declaring variables as close as possible to where they are used we do not waste time scrolling and scanning to absorb the meaning and intent of the code.

Design and Extensibility

These should be considered in the light of current and possible future use.

Current use: One can imagine a teacher with a sheet of scores for each student in a class, typing them in one at a time and then scribbling down each adjusted average result at the end of an existing list of scores. As the program stands the teacher is going to need to retype the program name for every student or at least use the command history. Putting the core of the code into a loop with a user generated exit condition seems like an obvious improvement. Alternatively if the student names and scores are already available in a suitable electronic form, processing an existing file and generating an output file with the adjusted average appended to the end of each line would be an obvious enhancement.

Future use: Next time round there may be seven assignments to mark or perhaps just three. So variability of the number of test scores per student and corresponding low scores to drop are obvious enhancements. Indeed one may wish to change the scoring policy altogether. There may not be an immediate requirement for these features but adopting a good design early on will allow incremental improvements and new features to be easily added.

Having thought through some of the possible use cases we are in a position to focus back on specific design issues to see how a more extensible foundation might be laid.

`main()` is too busy. It prompts the user, collects input, processes it and displays the result. These tasks should be factored into their own functions. The next issue is the use of magic numbers. Although the hard coded values of 5 and 1 for the the total number of scores and the scores that should be dropped are adequate for the current requirement, lifting them out of the code and into constants of suitable scope will make it easier to parameterise these in the future as well as clarifying the current code by providing variable names that are meaningful in English.

The amended code, which functions in the manner intended by the original code, is shown below. The only functional difference is the use of the character `5` in our prompt rather than the word `five`.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
#include <cassert>
using namespace std;
typedef vector<int> score_collection;
const int scores_to_include = 5;
const int scores_to_drop = 1;
void display_prompt() {
  cout << "Enter your " << scores_to_include
       << " test scores now.\n";
}
void get_scores(istream& in,
                score_collection& scores) {
  assert(scores.empty());
  for(int i = 0; i != scores_to_include; ++i) {
    int score = 0;
    in >> score;
    scores.push_back(score);
  }
}
int apply_marking_policy(score_collection& scores) {
  assert(scores.size() > scores_to_drop);
  sort(scores.begin(), scores.end(), less<int>());
  return accumulate(scores.begin() + scores_to_drop,
                    scores.end(), 0)
           / (scores.size() - scores_to_drop);
}
void put_result(ostream& out, const int result) {
  out << result;
}

int main() {
  display_prompt();
  score_collection scores;
  get_scores(cin, scores);
  put_result(cout, apply_marking_policy(scores));
  return 0;
}
```

The only interesting thing about `display_prompt()` is that we now use the integer constant `scores_to_include` to construct the prompt.

`get_scores()` now receives an `istream` reference as its first parameter. This is the stream we read the scores from. The second parameter is a collection that we use to store the scores, also passed by reference. We could have returned the scores by value but that would involve an unnecessary copy. In order to keep things simple and similar to the original code we don't validate each entered score. In practice this method should check that each score entered is within in valid range e.g. >= 0 and <= the maximum score for the specified test. These values could be defined as constants like `scores_to_include` and `scores_to_drop`. The most important thing about this method is that, like `display_prompt()` we use the integer constant `scores_to_include`, this time in the exit condition of the `for` loop. We also use a precondition to make clear our assumptions about the initial state of the container.

`apply_marking_policy()` does the work of sorting our scores, accumulating the ones that we are interested in and returning the adjusted average. It uses the stl algorithms `std::sort` (along with the function object (or functor) `std::less`) and `std::accumulate`. These work as their names suggest. For the details refer to Josuttis[1] and Meyer[2]. The important thing to note in this method is how the second of our constants, `scores_to_drop`, comes in to play. Our policy of dropping low scores before taking the average makes no sense if we attempt to drop all of the scores or more than we have collected so we make that pre-condition explicit in the code. Next we sort the scores, writing the result back to the original container and use the `scores_to_drop` constant to skip over the low scores at the beginning of the sorted collection when we accumulate. Using this approach places a requirement that our collection of scores must support random access iterators. (Should you wish to use a `std::list<>` collection you would need to adjust this method to use the list member function `sort` and then increment the `begin` iterator to the correct position in a loop using `scores_to_drop`.) We use `scores_to_drop` to adjust our divisor.

`put_result()` is uninteresting and merits no further comment.

Now that the code is better organised, implementing both our 'current use' requirements and our 'future use' requirements should be easier. With a bit more work it should be possible to support both an 'interactive' mode where the results are typed in or a batch mode where we pass our program a file to process. If we want to adjust either the number of scores or the scores to drop, the `const` qualification can be removed and they can be read from the command line. If we want to process a file, that can be passed as a command line parameter. If no filename is passed we can assume 'interactive' mode. When in interactive mode the user could type 'q' to quit at any point.

Introducing these features inevitably leads to more complexity, but if the original design is sound that complexity can be managed. If the original design is rigid and inflexible it simply is not possible to introduce new features. (Consider the case with the multiple `if` statements – how do we handle a variable number of scores?).

A version that includes the full feature list from above, with both interactive and batch mode was developed (Unfortunately there is not room to display it here but email me for a copy if you are interested). There is still room for improvement with this version. Validation is not included and the basic policy for calculating the adjusted average is fixed. Supporting different policies might be an interesting application in the use of policy template classes.

References:

[1] Nicolas Josuttis *The C++ Standard Library A Tutorial and Reference.*
[2] Scott Meyers *Effective STL.*

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

First of all, let me answer the main question. Why does the piece of code don't do what's desired? Simply, because it's logically wrong. Consider the way it tries to choose the `lowest` element:

```
int lowest = test1;
if(test2 < test1) lowest = test2; // So far, so good.
if(test3 < test2) lowest = test3; // Why?
```

The question is that given the second condition is true, how does it mean that the lowest element so far is `test3`? The following sequence is one example where this conclusion is wrong: "10, 21, and 18". As you can see, here, the first condition doesn't get desired, and the conclusion after the second condition comes wrong. The rest of the algorithm falls into the same trap.

Apart from that, and assuming that the algorithm is correct, I'd suggest the student to reconsider the code so that he/she can see whether he/she can use better constructs in his code. The code is in fact full of poor issues:

- That vague `#include <iomanip>` is, IMHO, a pre-standard beast lurking at the beginning of the code, according to a misconception that it is needed to enable us to use `std::endl`. As per the (98) Standard (§27.6.1.3/637, Phrase 22, which is an example), to use `std::endl`, it is enough to `#include <iostream>`.
- `using namespace std;` is another root of evil as many of the authors have also mentioned that. It is best to get used to avoid that, even in these plain-Jane examples.
- This code works for just when you want to do desired job for a sequence of length *five*. If this is going to be case forever, then "5" turns out to become a magic number, and hence, should be stored in a constant. Otherwise, to take the code out of this silly hard-wired-ness, and to make it flexible enough to deal with sequences of any length, there are two major alternatives to store the input numbers in: dynamic arrays, or `std::vector<>`. I wouldn't recommend the former, and to know the reason, I would address the student to bewildering number of pages divined to this in the C++ literature.

This is how the first few lines of code will become after the above considerations

```
#include <iostream>
#include <vector> // Perhaps. See below.
int main () {
  using std::cout;
  using std::cin;
  using std::endl;
```

if five is going to live forever

```
  const size_t len(5);
  int test[len];
```

and if it may change

```
  using std::vector;
  vector<int> test;
```

Note that having chosen this, you need to add `#include <vector>` as well.

After a few `std::cout`s, the best way to read the input sequence is not to re-invent wheel. That is, to use `std::copy()`:

```
  using std::copy;
  using std::istream_iterator;
```

Which require `#include <algorithm>` and `#include <iterator>`.

Either

```
  copy(istream_iterator<int>(cin),
       istream_iterator<int>(), test);
```

Or

```
  using std::back_inserter;
  copy(istream_iterator<int>(cin),
       istream_iterator<int>(), back_inserter(test));
```

AFAI've understood, we're about to choose the minimum. That's an easy piece of job:

```
  using std::min_element;
```

Either

```
  const int lowest = min_element(test, test + len);
```

Or

```
  const int lower = min_element(test.begin(),
  test.end());
```

Another useful consequence of this method is the removal of those excessive comments. (See C++ Gotchas, Item # 1)

Next poor issue which catches my attention is the poor style of naming: The variable named `average` is, in fact, the summation of the elements, and `divide` is the average!

Neglecting the poor style of naming, I draw your attention to type of `divide`. I do doubt if this is what it is assumed to be.

To achieve the desired average, furthermore, I'd do one of the following:

```
  using std::accumulate;
```

Either

```
  const double average = (accumulate(test, test + len,
  0.0) – lowest) / (len – 1);
```

Or

```
  const double average = (accumulate(test.begin(),
       test.end(), 0.0) - lowest) / (test.size() – 1);
```

Note that to use `std::accumulate()`, we need to `#include <numeric>`.

There are two remaining points worse mentioning: First, the initial value sent to it should be 0.0 rather than 0. (See Effective STL, Item # 37.) Second, the `const`s are intentionally put there. They are not modified in the next steps. Thus, for the sake of `const`-correctness, they should be declared so.

Again, many thanks to the Standard as the remained excessive comments will be vanished. We're better, however, to add a comment to the last line.

A little remained point is that if this functionality is likely to be used a lot by the student, why not modularise it? It turns out that the following seems to be a best solution then

```
template<typename ReturnType, typename InputIterator>
ReturnType avgButMin(InputIterator first,
                     InputIterator last) {
  typedef typename iterator_traits<InputIterator>::
                               value_type value_type;
  const value_type m(*min_element(first, last));
  const ReturnType avg
      = (accumulate(first, last, ReturnType(0)) - m)
              / (distance(first, last) - 1);
  return avg;
}
```

And then use it like this

```
int a[] = {80, 84, 100, 90, 60};
const size_t n(sizeof(a)/sizeof(*a));
double avg = avgButMin<double>(a, a + n);
```

I won't offer any further explanations as how to construct that template function is another full issue for itself.

From Robert Lytton <robert.lytton@metagence.com>

Student: The problem is, if I change up number 3 and 4 I get a different answer.
Critic: Explain to me what your code is doing.
Student: Well, `main` is ...<snip>... and if `test5` is less than `test4` – Ooops I see my problem. I am not interested if `test5` is less than `test4`, I am only interested if `test5` is less than the lowest.
Critic: How are you going to fix it?
Student: I need to compare each test against the lowest found so far.
Critic: This sounds like a loop...

```
for(i = 0; i < num_tests; ++i) {
  if(test[i] < lowest) {
    lowest = test[i];
  }
}
```

Critic: By the way what was the value that should have been returned?
Student: <silence>
Critic: Before we worry about defining and initialising variables for the loop, let's take a big step back. Test Driven Development...

First a range of grades and the required results:

```
80,84,60,100,90  => 88.5
80,84,100,60,90  => 88.5
```

As the result expected has a floating point, we will use `float` to store the numbers. Next a 'test' harness:

As we are in C++ we will hold the test data in a `vector` instead of an array. We will parcel this up with the expected result in a structure and change `test` to `grade` to aid clarity. For simplicity we will be `using namespace std`.

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

struct test_grade {
  vector<float> grade;
  float expected_result;
};
void test_average_less_lowest() {
  // we can't pass an array range directly to the
  // vector initialisation...
  const float test_1[] = {80,84,60,100,90};
  const float result_1 = 88.5;
  const float test_2[] = {80,84,100,60,90};
  const float result_2 = 88.5;
  // but we can pass an array iterator range...
  const test_grade test_case[]
    = { {vector<float>(test_1, test_1 +
              sizeof(test_1)/sizeof(test_1[0]) ),
              result_1},
        {vector<float>(test_2, test_2 +
              sizeof(test_2)/sizeof(test_2[0]) ),
              result_2} };
```

```
  const int num_test_cases = sizeof(test_case)
              / sizeof(test_case[0]);
  cout << "TEST: average_less_lowest()" << endl;
  for(int i = 0; i < num_test_cases; ++i) {
    float result
        = average_less_lowest(test_case[i].grade );
    if(result != test_case[i].expected_result) {
      cout << "      Failed test " << i << endl;
    } else {
      cout << "      Passed test " << i << endl;
    }
  }
}
```

And finally we develop the code: We now have a specification for the interface – designed by the user not the implementer.

```
#include <algorithm>
float average_less_lowest(const vector<float>& grade){
  float total = 0;
  float lowest = grade[0]; // safe initial value
  for( int i = 0; i<grade.size(); ++i) {
    total += grade[i];
    lowest = min(lowest,grade[i]); // from <algorithm>
  }
  return (total - lowest) / (grade.size() - 1);
}
int main() {
  test_average_less_lowest();
  return 0;
}
```

Building and running all works. But what if we only pass in one grade or even none?

First we write the test cases:

```
void test_average_less_lowest() {
  const float test_1[] = {80,84,60,100,90};
  const float result_1 = 88.5;
  const float test_2[] = {80,84,100,60,90};
  const float result_2 = 88.5;
  const float test_3[] = {80};
  const float result_3 = 0;
  // const float test_4[] = {}; can't initialise
  // an empty array.
  const float result_4 = 0;
  const test_grade test_case[] = {
     {vector<float>(test_1, test_1 +
             sizeof(test_1)/sizeof(test_1[0]) ),
             result_1},
     {vector<float>(test_2, test_2 +
             sizeof(test_2)/sizeof(test_2[0]) ),
             result_2},
     {vector<float>(test_3, test_3 +
             sizeof(test_3)/sizeof(test_3[0]) ),
             result_3},
     {vector<float>(), result_4} };
  ...
```

And running the test we discover there is indeed a problem. The 3rd test case fails, the 4th causes an exception. It seems we need to make sure we don't divide by zero and also check for an empty container. In both of these situations, as specified by the test cases, we return zero. We can also benifit more from the standard library. We could use an iterator,

```
vector<float>::const_iterator i;
for(i = grade.begin(); i != grade.end(); ++i) {
  total += *i;
  lowest = min(lowest,*i);
}
```

but after checking Josuttis' *'Summary of STL Algorithms*" (try `google.com`), we can make our intentions clearer.

```
#include <numeric>
float average_less_lowest(const vector<float>& grade){
  float result(0);
  const int grades_to_count = grade.size() - 1;
  if(grades_to_count > 0) {
    const float total = accumulate(grade.begin(),
                                   grade.end(), float(0));
    // !grade.empty() so min_element() != grade.end()
    const float lowest = *min_element(
                         grade.begin(), grade.end() );
    result = (total - lowest) / grades_to_count;
  }
  return result;
}
```

Running the tests again, the code fixes the problem found in the new test cases and does not break the old test cases. How about some more tests? Considered them added and passing. (Any rounding error within the test case is left as an exercise for the reader.)

And now for the application:

We will need to use a `vector` container because `average_less_lowest()` expects one. Using a `vector` automatically gives us safety in accepting more or less than five grades. It also allows us to use stream iterators.

```
#include <iterator>
int main() {
  cout << "This program will gather grades and "
       << "drop the lowest, giving you the average\n"
       << "Please enter your grades (ending with "
       << "'=').\n";
  // create two iterators, and fill the vector using
  // them. Users separate grades with ANY white
  // space. Entry is terminated by non-white space,
  // non-float character.
  istream_iterator<float> intReader(cin);
  istream_iterator<float> intReaderEOF;
  vector<float> grade(intReader, intReaderEOF);
  cout << "Average grade (with lowest removed) for ";
  copy(grade.begin(), grade.end(),
       ostream_iterator<float>(cout, ", "));
  cout << " = " << average_less_lowest(grade) << endl;
  return 0;
}
```

Using Test Driven Development has naturally separated the algorithm from the application. This is a good thing. We could use a batch file to drive `average_less_lowest()` if we wished. Alternatively if using `float`s is not what we want, we can change the algorithm easily. Instead of having two versions we can make `average_less_lowest()` a template function. First our test harness: We will factor out the commonality of the testing loop and use templates.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <typename T>
struct test_grade {
  test_grade(const vector<T>& g, const T& r)
                  : grade(g), expected_result(r) {};
  vector<T> grade;
  T expected_result;
};
template <typename T>
void call_test(vector<test_grade<T> > test_case) {
  vector<test_grade<T> >::const_iterator i;
  int num = 0;
  for(i = test_case.begin();
      i != test_case.end(); ++i) {
    ++num; // start with test 1 instead of 0,
           // for non-programmers
    if(average_less_lowest(i->grade)
                        != i->expected_result) {
      cout << "      Failed test " << num << endl;
    } else {
      cout << "      Passed test " << num << endl;
    }
  }
};
void test_average_less_lowest() {
  // we will initialise the floats with ints and
  // ignore the warnings!
  const int test_1[] = {80,84,60,100,90};
  const int test_2[] = {80,84,100,60,90};
  const int test_3[] = {80};
```

```
    // const int test_4[] = {};
    vector<test_grade<float> > test_case_float;
    test_case_float.push_back(test_grade<float>(
          vector<float>(test_1, test_1 +
          sizeof(test_1) / sizeof(test_1[0])), 88.5));
    test_case_float.push_back(test_grade<float>(
          vector<float>(test_2, test_2 +
          sizeof(test_2) / sizeof(test_2[0])), 88.5));
    test_case_float.push_back(test_grade<float>(
          vector<float>(test_3, test_3 +
          sizeof(test_3)/sizeof(test_3[0])), 0));
    test_case_float.push_back(test_grade<float>(
          vector<float>(), 0));
    vector<test_grade<int> > test_case_int;
    test_case_int.push_back(test_grade<int>(
          vector<int>(test_1, test_1 +
          sizeof(test_1)/sizeof(test_1[0])), 88));
    test_case_int.push_back(test_grade<int>(
          vector<int>(test_2, test_2 +
          sizeof(test_2)/sizeof(test_2[0])), 88));
    test_case_int.push_back(test_grade<int>(
          vector<int>(test_3, test_3 +
          sizeof(test_3)/sizeof(test_3[0])), 0));
    test_case_int.push_back(test_grade<int>(
          vector<int>(), 0));
    cout << "TEST: average_less_lowest() with float"
         << endl;
    call_test(test_case_float);
    cout << "TEST: average_less_lowest() with int"
         << endl;
    call_test(test_case_int);
  }
  int main() {
    test_average_less_lowest();
    return 0;
  }
```

Compile and run – the test cases fail to compile as we have no `int average_less_lowest(vector<int>)` defined.

Next to fix our development code:

All we need to do is define `average_less_lowest()` as a `template<typename T>` function and replace occurrences of `float` with `T`.

```
  template<typename T>
  T average_less_lowest(const vector<T>& grade) {
    T result(0);
    const int grades_to_count = grade.size() - 1;
    if(grades_to_count > 0) {
      const T total = accumulate(grade.begin(),
                                 grade.end(), T(0));
      const T lowest = *min_element(grade.begin(),
                                    grade.end());
      result = (total - lowest) / grades_to_count;
    }
    return result;
  }
```

Compile and run and all test cases pass. We have confidence that our quick change to `average_less_lowest()` has not broken earlier code and it now full fills the new 'spec' documented in `test_average_less_lowest()`.

From Ken Munro <ken@kjmunro.co.uk>

I'm going to iterate through your code a few times, first, to fix the reported problem, and then until I'm happy with the code.

1) Fix error

Let's start with the boring bit and fix the error reported in your code: The overlong sequence of `if` statements is the location of the problem, your code only compares neighbouring scores, e.g.

```
  assume we have scores of: 80, 84, 60, 100, 90
                          = test1..test5
  test3 < test2 (60 < 84) is true thus
                          lowest = test3, ie, 60, but,
  test5 < test4 (90 < 100) is also true which resets
                          lowest = test5, ie, 90,
```

which is correct for the local statement but not for the full list of scores.

To fix the problem we need to replace the right-hand side of the `<` comparisons with `lowest` so that each score is compared against the current lowest score. Running through the above example again gives:

```
  test3 < lowest (60 < 80) is true thus
                           lowest = test3, ie, 60
  test5 < lowest (90 < 60) is now false
                           leaving us with lowest = 60
```

which is correct

The code now works 'correctly' with the suggested test data, although questions may be asked as to whether the test coverage is adequate.

2) Code improvements

Comments: A scan through the code reveals that your comments are not entirely helpful – they comment the obvious and are at too low a level to be useful. Comments, generally, should be about why and not how. A quick redraft would give just 3 comments:

```
  // get student's test scores
  cout << "This program will gather five test scores";
  ...
  // calculate adjusted average score
  int lowest = test1;
  ...
  // output average
  cout << divide << endl;
  ...
```

Unlike the existing comments these should (ideally) be written before your code is written – we'll return to the utility of these comments a little later.

Variable naming and usage: The names used within your code are poorly chosen – I would suggest that you spend more time thinking about the names used in your code – good code flows from well-chosen names.

To specifics: the variables `test1` through `test5` should ring alarm bells with most programmers. Clearly these variables are closely related and should be grouped together; we'll adopt the simplest approach of a fixed-sized array.

We'll also rename to `testScores` rather than using the anaemic 'tests.' While we're at it we'll define a constant (or three) as well, which should get rid of any unsightly magic numbers left in the code:

```
  const int MinScore = 0; // assume %age scores
  const int MaxScore = 100;
  const int NoTestScores = 5; // must be > 1
  typedef int TestScores[NoTestScores];
```

We'll leave the test score type as an `int`, although some might prefer to hide this decision with a `typedef`.

Misnamed variables:

`average`: is used to total the scores – we'll rename it `totalScores`.

`averagescore`: is used to store the total scores minus the lowest score – we'll rename it `adjustedTotal`.

`divide`: is used to hold the average score – we'll rename it `averageScore`.

Assumption: the requirements stipulate that the average is to be a rounded-down whole number.

Finally (more oft-repeated advice), if you're using C++ you should try to use the variables as close as possible to their declarations; this reduces the amount of mental effort (and stress!) required in looking through the code and significantly increases the chances of your variables being initialised correctly, or, indeed, at all. We can now replace the brittle `if` statements with a simple loop: see Listing 1.

You may like to note that we can change the number of test scores by simply changing the `NoTestScores` constant, cf, your original solution which would require an additional variable and `if` statement for each additional test score.

The above changes were run through the same tests and still appear to work.

3) More improvements

Whilst the code has improved, there are still grounds for concern:

1 Input of scores not restricted to valid values.
2 Lack of proper testing.
3 Mixing UI (user interface) with application code (separation of concerns).

I'm going to start by ignoring point 1 as it would expand this solution well past the point of boredom: if we were to do it full justice we'd have to look at filtering out non-numeric input (including control characters), preventing buffer-overruns, verifying the resulting numbers, procedural (C) vs OO (C++) approach, and, possibly even, accounting for platform differences. The remaining 2 points can be dealt with concurrently, more-or-less.

Unit testing

In any real program you'd be expected to demonstrate its veracity (I would hope) – with the current program you'd have to draw up a list of inputs, calculate their averages, and manually input each set and check the output. This could easily get quite tiresome as it has to be done *every* time you change your code, furthermore, you may well find that you need more than just a couple of data sets – more likely than not we'd have an un(der)tested program. What you really want is an automated test suite, which takes us into the realm of unit tests. The code, however, mixes the UI with the averaging algorithm, preventing us from testing the latter without the former. We therefore need to break up our monolithic code before we can configure some simple automated testing (I'm deliberately ignoring the unit test suites available for C++, such as CppUnit, partly as they're unlikely to be installed by default, and partly as I rarely use C/C++ in the real world).

You'll note that our revised comments each describe a separate part of the program – these can be conveniently turned into routines, after which, the comments can be deleted:

```
getTestScores(testScores);
averageScore = averageScoreFor(testScores);
outputAverageScore(averageScore);
```

Separating the code is mostly trivial: the only complication is that we'll have to put the `averageScoreFor` routine in its own module (Listings 2 & 3) so that we can use it separately.

A test program is now written, see Listing 4. We're using simple look-up tables for the test data but it could equally come from a file or be generated by an alternative implementation of the algorithm at run-time etc. The tests are run and, you'll be relieved to hear, all work correctly. You should get into the habit of writing unit tests as soon as possible; I can absolutely guarantee that you will write better code as a result.

Separation of concerns

We have through the above separated the UI and application code – they're now in separate routines and namespaces. In the current example, unit testing not withstanding, it is not a big deal but it is important that you are aware of the issue, otherwise, for example, if you move to GUI development, you'll find that many of the development environments make it all too easy to create your very own tar pit by seamlessly mixing UI and app code.

4) Future improvements

I'm now reasonably content with the code but I guess we should briefly consider the future well-being of our code.

Exception handling: I omitted the exception handling from the main functions in the interests of brevity/clarity, but in production code you would definitely need them, i.e. do as I say, not...

Expanding the number of test scores: As already noted this should be easy to do at compile-time. It is, however, a little more challenging at run-time: here, we'd need some sort of dynamic structure, such as a container or dynamically-allocated array, which would benefit from being encapsulated in a class – but that's another and longer story.

Using a different algorithm: Since the algorithm is now in its own separately-compiled module we can readily change it without impacting on the main program. Changing the algorithm at run-time could also be done but you'd need to set up a class hierarchy (Strategy pattern) or, if you're using pure C, an array of function pointers.

Validation of inputs: Discussed earlier (I hope).

Unit tests: If you do extend or modify the code remember to update the unit tests accordingly. A fairly useful approach to modifying (or creating) code with unit tests is to write your tests before writing new code – so-called test-driven development (TDD). Once convinced of the efficacy of unit testing (and you will be) you should start using a proper unit testing suite – most of them seem to be modelled on the original JUnit (for Java).

Listing 1: `main.cpp`

```
#include <iostream>
namespace {
  const int NoTestScores = 5;
  const int MinScore = 0;
  const int MaxScore = 100;
  typedef int TestScores[NoTestScores];
  void getTestScores(TestScores& testScores) {
    using namespace std;
    cout << "This program will gather "
         << NoTestScores << " test scores and\n";
    cout << "drop the lowest score, giving you the "
         << "average\n\n";
    cout << "Please enter " << NoTestScores
         << " test scores\n";
    for(int i = 0; i < NoTestScores; i++) {
      cin >> testScores[i];
    }
  }
  int averageScoreFor(const TestScores& testScores) {
    int lowestScore = MaxScore;
    int totalScores = 0;
    for(int i = 0; i < NoTestScores; i++) {
      if(testScores[i] < lowestScore) {
        lowestScore = testScores[i];
      }
      totalScores += testScores[i];
    }
    int adjustedScores = totalScores - lowestScore;
    return adjustedScores / (NoTestScores - 1);
  }
  void outputAverageScore(int averageScore) {
    std::cout << averageScore << std::endl;
  }
} // namespace
int main() {
  TestScores testScores;
  getTestScores(testScores);
  int averageScore = averageScoreFor(testScores);
  outputAverageScore(averageScore);
}
```

Listing 2: `averageScore.h`

```
#ifndef AVERAGE_SCORE_H
#define AVERAGE_SCORE_H
namespace Accu {
  const int MinScore = 0;
  const int MaxScore = 100;
  const int NoTestScores = 5;
  typedef int TestScores[NoTestScores];
  int averageScoreFor(const TestScores& testScores);
}
#endif
```

Listing 3: `averageScore.cpp`

```
#include "averageScore.h"
namespace Accu {
  int averageScoreFor(const TestScores& testScores)
     {...} // per listing 1
}
```

Listing 4: `testMain.cpp`

```
#include <iostream>
#include "averageScore.h"
namespace {
  using namespace Accu;
  const int NoUnitTests = 10;
  const TestScores UnitTestInputs[NoUnitTests] = {
    // test with lowest value in every position
    {60, 80, 84, 100, 90},
    {80, 60, 84, 100, 90},
    {80, 84, 60, 100, 90},
    {80, 84, 100, 60, 90},
    {80, 84, 100, 90, 60},
    // tests with same values
    {50, 50, 50, 50, 50},
    {MinScore,MinScore,MinScore,MinScore,MinScore},
    {MaxScore,MaxScore,MaxScore,MaxScore,MaxScore},
    // ascending and descending tests
    {10, 20, 30, 40, 50},
    {90, 80, 70, 60, 50}
  };
  const int UnitTestResults[NoUnitTests] = { 88, 88,
       88, 88, 88, 50, MinScore, MaxScore, 35, 75 };
```

```
  const std::string TestCorrect = ".";
  const std::string TestIncorrect = "X";
} // namespace
int main() {
  int result = EXIT_SUCCESS;
  for(int Test = 0; Test < NoUnitTests; Test++) {
    if(averageScoreFor(UnitTestInputs[Test])
                    == UnitTestResults[Test]) {
      std::cerr << TestCorrect;
    } else {
      std::cerr << TestIncorrect;
      result = EXIT_FAILURE;
    }
  }
  std::cerr << "\nUnit tests completed\n";
  return result;
}
```

The Winner of SCC 31

The editor's choices are:

Ken Munro and Richard Wheeler.

Special mention is to **Chris Main** for one of the most amusing answers I've ever seen here.

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

Francis' Commentary

There are several problems with this program. The first is that the problem has not been well specified. It seems by examining the code that the intention is to calculate the arithmetic mean of all but the smallest of five values.

Let us think how we would identify the smallest of a list of numbers. Assume that we have the list in a column. You would tentatively assume that the smallest number was the first one and then scan down the list checking each subsequent value to see if it was smaller than the current smallest. If it is we update the smallest value. When we reach the end of the list we know what the smallest value is.

Let me focus on this and assume that the student knows nothing about arrays or any other form of collection (yes, I know that is pretty silly, but sometimes instructors like to set these exercises as motivating examples for introducing collections.)

Assuming that we have initialised the five variables; `test1`, `test2`, `test3`, `test4` and `test5`; the following piece of code exactly duplicates the above description.

```
int lowest(test1); // tentatively assume test1 is
the smallest
if(test2 < lowest) lowest = test2;
if(test3 < lowest) lowest = test3;
if(test4 < lowest) lowest = test4;
if(test5 < lowest) lowest = test5;
```

Now we can see the student's critical error. There is no use comparing each value with the next one, we must compare each value with the lowest so far.

Next let me rewrite the final part of the student's code so that it is self documenting:

```
int const total(test1+test2+test3+test4+test5);
int const top4(total - lowest);
double const mean(top4/4.0);
cout << "The mean grade is " << mean << endl;
```

One of the important points here is that variables are only declared when we are ready to initialise them. In addition I advocate using `const` qualified variables extensively. The student was already part of the way there by using named variables to identify each step in the calculation, though the names he chose were not exactly the most appropriate ones.

I am not going to comment on the data capture part of the program because it is tedious and if the student was limited to placing everything in `main()` without using loops or containers there is not much option.

Once we allow the student to use a loop things become much neater because we read in the first value and use it to initialise both `total` and `lowest`. Next we loop four times reading in a new value, adding it into `total` and testing to see if we have to adjust the lowest. The code might look something like:

```
cout << "What is the first grade? ";
int total;
cin >> total;
int lowest(total);
for(int i(0); i != 4 ; ++i) {
  cout << "What is the next grade? ";
  int grade;
  cin >> grade;
  total += grade;
  if(grade < lowest) lowest = grade;
}
int const top4(total - lowest);
double const mean(top4/4.0);
cout << "The mean grade is " << mean << endl;
```

At some point the instructor will need to discuss problems arising from incorrect data being keyed in. Notice that we now have an extensible program which can deal with more than five grades by simply adjusting the number of iterations of the loop. That is a good point to talk about magic numbers.

Finally we have the issue of 'remembering' the individual grades rather than keeping a running total of them. At that point I would be talking to my students about using `std::vector`.

Now I have not seen the answers readers sent in, but I suspect most of them were very anxious to get into using an array or `vector`. For the problem as set (and even extended) containers are not that useful and there are a good number of other programming points that needed to be addressed first.

Student Code Critique 32

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

I still wonder about the lack of knowledge (or rather awareness) among beginners of the extensive functionality offered by the standard library. Let this be reflected in your answer to the student, with a corresponding solution.

This computes the product of two N by N matrices. It works fine in cygwin compiler, but it doesn't in VC++. The strange thing is when I have N = 2 no problem, but N = 3 makes problem. I am not sure I use 'new' operator correctly in the following program. Can someone help in finding the problem here ?

```
#include <iostream.h>
#include <process.h>
void main(void)
{
  int N, i, j, k;
  double **A, **B, **C;
  double sum = 0.0;

  cout << "Dimension of Matrix ?" << endl;
  cin >> N;

  A = new (double *);
  B = new (double *);
  C = new (double *);
  for(i=0; i<N; i++){
    A[i] = new double[N];
    B[i] = new double[N];
    C[i] = new double[N];
  }

  for(i=0; i<N; i++)
    for(j=0; j<N; j++){
      cout << "A[" << i << "][" << j << "] = ?" << endl;
      cin >> A[i][j];
    }

  for(i=0; i<N; i++)
    for(j=0; j<N; j++){
      cout << "B[" << i << "][" << j << "] = ?" << endl;
      cin >> B[i][j];
    }

  for(k=0; k<N; k++)
    for(i=0; i<N; i++){
      sum = 0.0;
      for(j=0; j<N; j++)
        sum += A[i][j]*B[j][k];
      C[i][k] = sum;
    }

  cout << endl << endl << endl;

  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
      cout << "C[" << i << "][" << j << "] = "
           << C[i][j] << endl;
}
```

Features

Patterns in C – Part 1

Adam Petersen <adampetersen75@yahoo.se>

Over the last ten years, the pattern format has gained a tremendous popularity as the format used for capturing experience. One of the reasons for this popularity is the unique success of the classic book *Design Patterns* by the Gang of Four [1]. The *Design Patterns* book definitively served the community by spreading the word about patterns.

Today, patterns in the software industry aren't limited to design; there exists a broad range of patterns, covering analysis patterns, patterns for organizations, patterns for testing, etc.

As most patterns are described in the context of an object oriented design, one is easily led to believe that patterns require a language with support for object orientation. By browsing a popular online bookstore, I noticed a lot of language specific pattern literature: design patterns in Java, C#, Smalltalk and other popular object oriented languages. But, where is the one targeting the unique implementation constraints and techniques for the C language? Isn't it possible to use patterns in the development of C programs or doesn't it add any benefits?

An important thing to realize about patterns is that they are neither a blueprint of a design, nor are they tied to any particular implementation. By those means, shouldn't it be possible to find mechanisms fitting the paradigm of C, letting C programmers benefit from the experience captured by patterns?

What You Will Experience in This Series ...

It is my belief that C programmers can benefit from the growing catalogue of patterns. This series will focus on the following areas:

- **Implementation techniques.** I will present a number of patterns and demonstrate techniques for implementing them in the context of the C language. In case I'm aware of common variations in the implementation, they will be discussed as well. The implementations included should however not by any means be considered as a final specification. Depending on the problem at hand, the implementation trade-offs for every pattern has to be considered.
- **Problem solved.** Patterns solve problems. Without any common problem, the "pattern" may simply not qualify as a pattern. Therefore I will present the main problem solved by introducing the pattern and provide examples of problem domains where the pattern can be used.
- **Consequences on the design.** Every solution implies a set of trade-offs. Therefore each article will include the consequences on the quality of the design by applying the pattern.

... And What You Won't

- **Object oriented feature emulation.** The pattern implementations will not be based on techniques for emulating object oriented features such as inheritance or C++ virtual functions. In my experience, these features are better left to a compiler; manually emulating such techniques are obfuscating at best and a source of hard to track down bugs at worst. Instead, it is my intent to present implementations that utilize the strengths of the abstraction mechanisms already included in the C language.
- **In depth discussion of patterns.** As the focus in these articles will be on the implementation issues in C, the articles should be seen as a complement to the pattern descriptions. By those means, this series will not include exhaustive, in depth treatment of the patterns. Instead I will provide a high-level description of the pattern and reference existing work, where a detailed examination of the pattern is found.

Pattern Categories

The patterns described in this series will span the following categories.

- **Architectural patterns.** Frank Buschmann defines such a pattern as *"a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them"* [2].
- **Design patterns.** These typically affect the subsystem or component level. Most patterns described in this series will be from this category, including patterns described in the classic *Design Patterns* [1] book.
- **Language level patterns.** This is the lowest level of the pattern-categories, also known as idioms. A language level pattern is, as its name suggests, mainly unique to one particular programming language. One simple, classic example is the `strcpy` version from Kernighan and Ritchie [3].

```
void strcpy(char *s, char *t) {
   while(*s++ = *t++);
}
```

The Foundation

Our journey through the patterns will start with a language level pattern that decouples interface from implementation, thus improving encapsulation and providing loose dependencies.

This pattern will lay the foundation for many of the subsequent parts of this series.

First-Class ADT Pattern

It's getting close to the project deadline as the project manager rushes into your office. "They found some problem with your code", he says with a stressed voice. "According to the test-team, you cannot register more than 42 orders for a given customer. Sounds strange, doesn't it?"

Darn. You knew it. Those hard coded limits. "Oh, I'll have a look at it", you reply softly. "Fine, I expect the problem to be solved tomorrow", the manager mumbles as he leaves your office.

"No problem", you reply, well confident that the design of the customer routines are highly modular and clearly implemented (after all, you've implemented it yourself).

You launch your favourite code-editor and open a file with the following content:

```
/* Customer.h */
< include guards and include files >
#define MAX_NO_OF_ORDERS 42
/* Internal representation of a customer. */
typedef struct {
  const char* name;
  Address address;
  size_t noOfOrders;
  Order orders[MAX_NO_OF_ORDERS];
} Customer;
void initCustomer(Customer* theCustomer,
          const char* name, const Address* address);
void placeOrder(Customer* customer,
                const Order* order);
/* and a lot of other related functions */
```

A quick glance reveals the problem. Simply increasing `MAX_NO_OF_ORDERS` would do, wouldn't it? But what's the correct value for it? Is it 64, 128, maybe even 2048 or some other magic number? Should customers with one, single order allocate space for, let's say, 2047 non-existing orders?

As you think of it, you realize that the current solution doesn't scale well enough. Clearly, you need another algorithm. You recall that a linked list exists in the company's code library. A linked list must do the trick. However, this means changing the internal structure of the `Customer`.

No problem, it looks like you thought of everything; the clients of the customer module simply use the provided functions for all access of the customer structure. Updating those functions should be enough, shouldn't it?

Information Hiding

Well, in an ideal world the change would be isolated to the one, single module. Given the interface above, clients depend upon the internal structure in at least one way.

At worst, the clients alter the internals of the data structure themselves leading to costly changes of all clients.

This can be prevented by frequent code-inspections and programmer discipline. In any case, we still have the compile-time dependencies and after changes, a re-compile of all clients is required and the compilation time may be significant in large systems.

The First-Class ADT pattern will eliminate both dependency problems. The pattern provides us with a method of separating interface from implementation, thus achieving true information hiding.

Definition of a First-Class ADT

ADT stands for Abstract Data Type and it is basically a set of values and operations on these values. The ADT is considered first-class if we can have many, unique instances of it.

Sounds close to the interface listed in the introductory example above, doesn't it? However, the data type in the example is not abstract as it fails to hide its implementation details. In order to make it truly abstract, we have to utilize a powerful feature of C – the ability to specify incomplete types.

Incomplete Types

The C standard [4] allows us to declare objects of incomplete types in a context where there sizes aren't needed. In our example implementation, we are interested in one property of incomplete types – the possibility to specify a pointer to an incomplete type (please note that the pointer itself is not of an incomplete type).

```
/* A pointer to an incomplete type (hides the
   implementation details). */
typedef struct Customer* CustomerPtr;
```

Instances of this pointer will serve as a handle for the clients of a First-Class ADT. This mechanism enforces the constraint on clients to use the provided interface functions because there is no way a client can access a field in the `Customer` structure (the C language does not allow an incomplete type to be dereferenced).

The type is considered complete as soon as the compiler detects a subsequent specifier, with the same tag, and a declaration list containing the members.

```
/* The struct Customer is an incomplete type. */
typedef struct Customer* CustomerPtr;
/* Internal representation of a customer. */
struct Customer {
  const char* name;
  Address address;
  size_t noOfOrders;
  Order orders[42];
};
/* At this point, struct Customer is considered
   complete. */
```

Object Lifetime

Before we dive into the implementation of an ADT, we need to consider object creation and destruction.

As clients only get a handle to the object, the responsibility for creating it rests upon the ADT. The straightforward approach is to write one function that encapsulates the allocation of an object and initializes it. In a similar way, we define a function for destructing the object.

```
/* Customer.h */
< includes and include guards as before >
/* A pointer to an incomplete type (hides the
   implementation details). */
typedef struct Customer* CustomerPtr;
/* Create a Customer and return a handle to it. */
CustomerPtr createCustomer(const char* name,
                           const Address* address);
/* Destroy the given Customer. All handles to it
   will be invalidated. */
void destroyCustomer(CustomerPtr customer);

/* Customer.c */
#include "Customer.h"
#include <stdlib.h>
struct Customer {
  const char* name;
  Address address;
  size_t noOfOrders;
  Order orders[42];
};
CustomerPtr createCustomer(const char* name,
                           const Address* address) {
  CustomerPtr customer = malloc(sizeof * customer);
  if(customer) {
    /* Initialize each field in the customer. */
  }
  return customer;
}
void destroyCustomer(CustomerPtr customer) {
  /* Perform clean-up of the customer internals,
     if necessary. */
  free(customer);
}
```

The example above uses `malloc` to obtain memory. In many embedded applications, this may not be an option. However, as we have encapsulated the memory allocation completely, we are free to choose another approach. In embedded programming, where the maximum number of needed resources is typically known, the simplest allocator then being an array.

```
/* Example of a static approach for memory
   allocation. */
#define MAX_NO_OF_CUSTOMERS 42
static struct Customer
objectPool[MAX_NO_OF_CUSTOMERS];
static size_t numberOfObjects = 0;
CustomerPtr createCustomer(const char* name,
                           const Address* address) {
  CustomerPtr customer = NULL;
  if(numberOfObjects < MAX_NO_OF_CUSTOMERS) {
    customer = &objectPool[numberOfObjects++];
    /* initialize the object */
  }
  return customer;
}
```

In case deallocation is needed, an array will still do, but a more sophisticated method for keeping track of "allocated" objects will be needed. However, such an algorithm is outside the scope of this article.

Copy Semantics

As clients only use a handle, which we have declared as a pointer, to the ADT, the issue of copy semantics boils down to pointer assignment. Whilst efficient, in terms of run-time performance, copies of a handle have to be managed properly; the handles are only valid as long as the real object exists.

In case we want to copy the real object, and thus create a new, unique instance of the ADT, we have to define an explicit copy operation.

Dependencies Managed

With the interface above, the C language guarantees us that the internals of the data structure are encapsulated in the implementation with no possibility for clients to access the internals of the data structure.

Using the First-Class ADT, the compile-time dependencies on internals are removed as well; all changes of the implementation are limited to, well, the implementation, just as it should be. As long as no functions are added or removed from the interface, the clients do not even have to be re-compiled.

Consequences

The main consequences of applying the First-Class ADT pattern are:

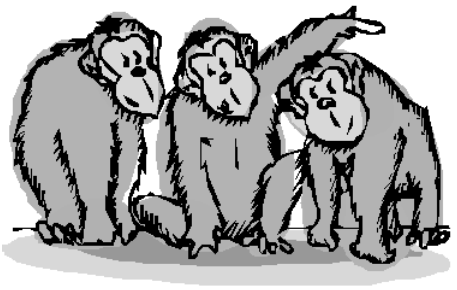
1. **Improved encapsulation.** With the First-Class ADT pattern we decouple interface and implementation, following the recommended principle of programming towards an interface, not an implementation.
2. **Loose coupling.** As illustrated above, all dependencies on the internals of the data structure are eliminated from client code.
3. **Controlled construction and destruction.** The First-Class ADT pattern gives us full control over the construction and destruction of objects, providing us with the possibility to ensure that all objects are created in a valid state. Similarly, we can ensure proper de-allocation of all elements of the object, provided that client code behaves correctly and calls the defined destroy-function.
4. **Extra level of indirection.** There is a slight performance cost involved. Using the First-Class ADT pattern implies one extra level of indirection on all operations on the data structure.
5. **Increased dynamic memory usage.** In problem domains where there may be potentially many instances of a quantity unknown at compile-time, a static allocation strategy cannot be used. As a consequence, the usage of dynamic memory tends to increase when applying the First-Class ADT pattern.

[concluded at foot of next page]

Professionalism in Programming #30

Code Monkeys (Part One)

Pete Goodliffe <pete@cthree.org>



> We are just an advanced breed of monkeys on a minor planet of a very average star. But we can understand the Universe. That makes us something very special.
>
> Stephen Hawking

As time marches relentlessly onwards we're drawing near to the 2005 ACCU conference (you have booked your place, haven't you?) I've been preparing this year's presentation, and so I thought that this would be a good opportunity to review what I presented last year.

In a previous article I asked the frivolous question: *how many programmers does it take to change a light bulb?* There could be any number of answers, but it really depends on who is doing the work. Different programmers work in different ways and will have their own individual approach to solve the same problem. There is always *more than one way to do it*[1], and different programmers' attitudes will lead them to make very different decisions.

In this series of articles we'll look at this; we'll investigate programmer attitudes, good and bad, and identify the key ones for successful programming. This includes how we approach the task of coding, and also how we relate to other programmers. We'll come to some surprising conclusions about what makes the best coders.

Monkey Business

The software factory is inhabited by a strange collection of freaks and social misfits. Any serious software system is built by a bunch of these people, with their different skill levels and attitudes, all working towards a common goal.

The way we work together and the kind of code we write will inevitably be shaped by our attitude to the work. If everyone was a diligent, hard working genius then our software would be a lot better; delivered on time, to budget, with no bugs. We're not perfect, and unfortunately it shows in the code we write.

To work out strategies to deal with this I'll lead us on a guided tour through a gallery of programmer stereotypes. We'll see the different types of code monkey. These are all directly based on the types of people I have met in the software factory. Of course it's a necessarily general list; you'll know programmers who fall into categories other than those listed here, or even fit several descriptions at once.

Even so, this shameless categorisation will highlight the important facts and show us how we can improve. We'll see:

- what makes different types of code monkey tick,
- how to work with each of them,
- how each code monkey can improve, and
- what we can learn from each of them.

As you read each code monkey description, ask yourself:

- Are *you* this type of programmer? How closely does the description match your programming style? What lessons can you learn to improve your approach to coding?
- How many people do you know like this? Are they close colleagues, and can you work with them better?

Without further ado, meet The Code Monkeys...

1. The Eager Coder

We'll start with this guy, because he[2] probably embodies the traits of most programmers. The Eager Coder is fast and fleeting; he thinks in code. An impulsive, natural born programmer, he tends to write code as soon as an idea forms in his head. He's not good at standing back and thinking first. So, although an Eager Coder does have very good technical skills, the code he writes never shows his true potential.

The Eager Coder often tries to use a new feature or idiom because it's fashionable, the best thing since the last big new idea. His desire to try out new tricks means that he applies technology even when it's not appropriate.

Strengths: Eager Coders are productive, in terms of code quantity. They write a *lot* of code. They love learning new stuff, and are really enthusiastic – even passionate – about programming. The Eager Coder loves his job, and genuinely wants to write good code.

Weaknesses: Due to his unfettered enthusiasm, the Eager Coder is hasty and doesn't think before rushing into the code editor. He does write a lot of code, but because he writes it so fast, it's flawed – the Eager Coder *spends ages debugging*. A little forethought would prevent many silly errors, and many hours ironing out careless faults.

Unfortunately the Eager Coder is a really bad debugger. In the same way he rushes into coding, he dives straight into debugging. He's not methodical, so he spends ages chasing faults down blind alleys.

He's a poor estimator of time. He'll make a reasonable estimate for the case when it all goes well, but it never *does* go according to plan; he always takes longer than expected.

What to do if you are one: Don't lose that enthusiasm – it's one of the best characteristics of a programmer. Because your joy lies in seeing programs work, to stand back and admire the beauty of code, work out practical ways to do this.

It mostly boils down to this simple piece of advice: *stop and think.* Don't be hasty. Work out personal disciplines that will help you, even something basic like writing *THINK* on a post-it-note stuck to your monitor!

How to work with them: When they work well, these are some of the best people to program alongside. The trick is to channel their energy into productive code rather than mindless flapping. They are great to get pair programming.

Ask an Eager Coder about what he's doing each day, and what his plans are. Show an interest in his design – it will encourage him to really think about it! If you rely on an Eager Coder's work, ask for early pre-releases, and ask to see their unit tests too.

1 The Perl programmers' mantra.

2 I'll describe all code monkeys as male, for no other reason than clarity of prose.

[continued from previous page]

Examples of use

The most prominent example comes from the C language itself or, to be more precise, from the C Standard Library – `FILE`. True, `FILE` isn't allowed by the standard to be an incomplete type and it may be possible to identify its structure, buried deep down in the standard library. However, the principle is the same since the internals of `FILE` are implementation specific and programs depending upon them are inherently non-portable.

Sedgewick[5] uses First-Class ADT to implement many fundamental data structures such as linked-lists and queues.

This pattern may prove useful for cross-platform development. For example, when developing applications for network communication, there are differences between Berkeley Sockets and the Winsock library. The First-Class ADT pattern provides the tool for abstracting away those differences for clients. The trick is to provide two different implementations of the ADT, both sharing the same interface (i.e. include file).

Next time

We will climb one step in the pattern categories and investigate a pattern from the Design Patterns [1] book. The next pattern may be useful for controlling the dynamic behavior of a program and in eliminating complex conditional logic.

Adam Petersen

References

[1] Gamma, Helm, Johnson and Vlissides, *Design Patterns*, Addison-Wesley

[2] Buschmann, Meunier, Rohnert, Sommerlad and Stal, *POSA, A System of Patterns, Volume 1*, Wiley

[3] Kernighan and Ritchie, *The C Programming Language*, Prentice Hall

[4] ISO/IEC 9899:1999, *The C Standard*

[5] Sedgewick, R., *Algorithms in C, Parts 1-4*, Addison-Wesley

Acknowledgements

Many thanks to Drago Krznaric and Andre Saitzkoff for their feedback.

An Eager Coder benefits from appropriate management, to help with his discipline. Make sure his time is carefully placed on a project plan (you don't have to plan his time yourself).

2. The Code Monkey

If you ever need an infinite number of monkeys, these guys would be your first choice. (I wouldn't advise it though, you'll be picking monkeys for a *loooong* time!)

The Code Monkey writes solid but uninspired code. Given an assignment, they'll faithfully plod through it, ready to be handed the next one. Perhaps it's a little unfair, but because of their menial work these guys are also known as grunt programmers.

Code Monkeys have quieter personalities. Afraid to push for good jobs, they tend to get sidelined on unglamorous projects. They carve out a niche as maintenance programmers, keeping the aged codebase going whilst the pioneers are off writing exciting replacements.

A junior Code Monkey will learn and progress given time and mentoring, but is given 'low risk' assignments for now. An older Code Monkey has probably stagnated, and will retire a Code Monkey. He'll be quite happy to do so.

Strengths: Give them a job and they'll do it, reasonably well, reasonably on time. A Code Monkey is reliable, and can usually be counted on to put in extra effort when it comes to crunch time.

Unlike an Eager Coder, a Code Monkey is a good estimator of time. They are methodical and thorough.

Weaknesses: Although a Code Monkey is careful and methodical, they don't *think outside of the box*. They lack design flair and intuition. A Code Monkey will follow the existing code design conventions unquestioningly, rather than address any potential problems. Since they are not accountable for the design, they don't accept responsibility for any problems that arise, and won't often take the initiative to investigate and fix them.

It's hard to teach a Code Monkey new stuff; they're just not interested.

What to do if you are one: Do you want to explore new areas and broaden your responsibility? If so, start to strengthen your skills by practicing on personal projects. Grab some books and study new techniques.

Push for more responsibility, and offer to join in the design work. Take the initiative in your current work – identify possible failure points early, and work out plans to avoid them.

How to work with them: Don't look down on a Code Monkey, even if you have stronger technical skills or greater responsibility. Encourage them – compliment their code and teach them techniques to improve their work.

Write your code thoughtfully to make the maintenance programmer's job as easy as possible.

3. The Guru

This is the fabled mystic genius, a program wizard. The Guru tends to be quiet and unassuming, perhaps even a little odd. He writes excellent code, but can't communicate well with mere mortals.

The Guru is left alone to work on the fundamental stuff: frameworks, architectures, kernels, and so on. He holds the deserved respect (and sometimes fear) of his colleagues.

Omniscient, the Guru knows all and sees all. He turns up sagely in any technical discussion to dispense his expert opinion.

Strengths: Gurus are the experienced magicians. They know all the new magic, and understand why the old tricks are better. These are the guys that created magic in the first place. They have a wealth of experience, and write mature maintainable code.

A good Guru is a wonderful mentor – there's so much to learn from him.

Weaknesses: Few Gurus can communicate well. They're not always tongue tied, but their ideas fly so fast and at a level beyond mere mortals', that it's hard to follow them. A conversation with a Guru either makes you feel stupid, confused, or both.

A bad Guru makes a fantastically bad mentor. They find it hard to understand why others don't know as much, or don't think as fast as them.

What to do if you are one: Try to step off your cloud and live in the Real World. Don't expect everyone to be as quick as you, or to think in the same way as you. It takes a lot of skill to explain something simply and clearly. Practice this.

How to work with them: If you cross paths with a Guru, learn from them. Absorb what you can – and not just technical stuff. To become established as a Guru takes a certain temperament and personality – knowledge but not arrogance. Observe this.

4. The Demiguru

The Demiguru *thinks* he's a genius. He isn't. He talks knowledgeably, but talks a load of rubbish.

This is probably the most dangerous type of code monkey; a Demiguru is hard to spot until the damage is done. Managers believe he's a genius, because he sounds so plausible and sure of himself.

A Demiguru is generally less quiet than a Guru. He's more boastful and full of himself.

Strengths: It's easy to assume that a Demiguru has *no* strengths, but his great asset is his belief in himself. It's important to trust your own abilities, and be secure that you write high quality code. However

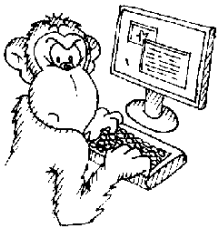
Weaknesses: The Demiguru's great weakness is his *belief in himself*. He overestimates his abilities, and when left to make important decisions will jeopardise your project's success. At worse, he's a serious liability. The Demiguru will haunt you, even after he's moved on to new pastures. You'll be left with the consequences of his bad decisions, and his overly clever code.

What to do if you are one: Right now, take an honest appraisal of your skills. Don't oversell yourself. Ambition is a good thing; pretending to be something you're not isn't.

You may not be doing this on purpose, so be objective about what you can and cannot do. Be more concerned about the quality of your software than how important or clever you look.

How to work with them: Be very, very careful.

5. The Arrogant Genius



This guy is a subtle, but significant, variation on the Guru species. He annoys the pants off of you – he's the killer programmer. Fast and efficient, he writes high quality code. Not quite a Guru, but he's hot.

But because he's all too aware of his own skills he's cocky, condescending and demeaning. The Genius is often terminally argumentative, because he's so used to being right and having to promote his correct view over other's wrong opinions. He's become used to it now.

The most annoying thing is that most of the time he is right, so you're bound to lose any argument with him. If you are correct, he'll keep talking until the argument moves on to something he is right about.

Strengths: The Genius has considerable technical skill. He can provide a strong technical lead, and will catalyse a team when everyone agrees with him.

Weaknesses: The Genius doesn't like to be proved wrong, and thinks that he must always be right. He feels compelled to act as an authority; the Genius 'knows' everything about everything. Even if he has no experience at all, he still tries to look knowledgeable. He can never say *I don't know*, suffering from a full humility bypass.

What to do if you are one: Not everyone achieves God-like status, but there are plenty of good programmers worthy of respect. Recognise this. Practice humility, and honour other people's opinions

Look for people who might have a more experienced viewpoint, and learn from them. Don't pretend – be honest about what you do and don't know.

How to work with them: *Do* show a Genius respect, and show respect *to other programmers* around him. Don't come up against him, and don't enter into unconstructive quarrels. But stand your ground – assert your reasonable opinions and views. Don't be daunted by him. Discussing technical issues with a Genius can make you a better programmer; just learn to detach your emotions first. If you're sure you're correct, gain allies to help fight the stance.

Take heed and avoid being cocky yourself.

Next Time

We'll look at some more programmer stereotypes, and work out what the 'ideal programmer' looks like. Stay tuned.

Pete Goodliffe

Wx – A Live Port – Part 3

Jonathan Selby <jon@xaxero.com>

In this, the final part of the series, Jon rounds off the port of an application from MFC to wxWidgets.

Internet Access

The `wxSocket` class provides a very simple interface over the Internet. You will need to have a phone connection or DSL/ISDN network connection established.

To get a web page is simplicity in itself.

The following is from the samples – Sockets – Client.

```
// define the URL
wxString urlname = "your.url.com"
wxURL url(urlname);
// Check to see url is valid
if(url.GetError() != wxURL_NOERR) {
  m_text->AppendText(("Error: couldn't parse URL\n"));
  m_text->AppendText(("=== URL test ends ===\n"));
  return;
}
// Get the data
wxInputStream *data = url.GetInputStream();
```

Read up in `wxStringBase` and `wxStringBuffer` for data manipulation.

This would take about a page in MFC.

For more sophisticated operations like perhaps talking to a mail server on port 25 you have to establish a socket connection directly using `wxSocketClient`. This is derived from `wxSocketBase` which handles both Server and Client connections.

Use `wxIPV4address` to store the address.

```
wxIPV4address sockAddr;
sockAddr.Hostname(pszHostAddress);
sockAddr.Service(nPort);
```

The host address can be a server name or a resolved IP Address.

Now we connect:

```
m_hSocket.Connect(lpSockAddr, TRUE);
```

Check we are OK

```
wxASSERT(m_hSocket.Error());
```

`TRUE` has the instruction wait for the connection to complete. The command returns true if a connection was made.

To avoid long timeouts you may want to set this flag to `FALSE` and use `WaitForConnect` after the connect. This will allow you to specify your own timeout.

Input and output is handled by `wxSocketBase`.

To read use the following

```
if(m_hSocket.WaitForRead(-1))
  m_hSocket.Read(pszRecvBuffer,256)
```

To write

```
wxString Buffer = "DATA";
  m_hSocket.Write(Buffer, Buffer.Length());
```

To close the connection:

```
if(!m_hSocket.Error()) {
  m_hSocket.Close();
}
```

For more advanced use of both client and server tools please take a look at the Sockets sample in the wxWidgets samples. The comments are very helpful.

Context Sensitive Help

wxWidgets supports winHelp, Microsoft HTMLHelp and wxHTMLHelp. The latter being a subset and useful for cross platform operation. There are also a lot of internal hooks to put in fast context help for dialogs.

Every wxWindow object can have some text associated with it. In practice this type of help seems to have been the most popular. Most users want a quick hint to get them on their way rather than a huge tome presented when they press the F1 button. The old Microsoft way was to pop up a document with the relevant passage. With a quick hint this would consume a huge amount of real estate on the screen and the many keystrokes required to get out of it detracted from the experience and generally soured the user on the F1 button.

So in this section we will design a simple window based help structure where every item has help of some kind and the F1 key can be used on the fly. Lastly we will create a hot link to the documentation file via Shift F1 or a menu item that will allow us to display the help.

If we have been using wxDesigner to its full extent the object functionality of the tool bar and menu bars will be already in place. You should see context sensitive help on the status bar as you pass the cursor over the object and on the toolbar, a tool tip should pop up. If this is not enough for our poor user, then they will have to start reading the documentation.

To provide a clearing house for a single point in the application to handle all help oriented commands (F1, context help), we use in our mainframe an implementation of `wxHelpProvider` that allows a great deal of flexibility here:

In the `App Class` header create:

```
wxHelpControllerHelpProvider* provider;
```

In the `mainframe` class create as private:

```
private:
  wxHelpController m_help;
```

and then create an inline function to return the controller

```
wxHelpController& GetHelpController() {return m_help;}
```

Now in the App inititialization before and after the `Mainframe` creation insert the help implementation:

```
>> provider = new wxHelpControllerHelpProvider;
>> wxHelpProvider::Set(provider);
```

Here we substantiate and set the controller class.

```
// create the main application window
m_mainFrame = new WXWindPlotFrame(m_docManager,
                    (wxFrame*) NULL, "WindPlot "+rs,
                    wxPoint(0, 0), wxSize(640, 480),
                    wxDEFAULT_FRAME_STYLE);
>> provider->SetHelpController(
                  & m_mainFrame->GetHelpController());
```

Here we link the controller with the main frame. We can now use the `SetHelpText` function of the `wxWindow` class for any object derived from `wxWindow` – views, dialogs etc. Just add the following text to the constructor for example

```
WXWindPlotFrame::WXWindPlotFrame(
          wxDocManager *manager, wxFrame *frame,
          const wxString& title, const wxPoint& pos,
          const wxSize& size, long type)
       : wxDocMDIParentFrame(manager, frame, -1,
                title, pos, size, type, "myFrame") {
  SetHelpText(("To review toolbar functions, rest
         mouse over the toolbar button and read the
         description on the bottom Status
         bar.\nSelect help menu Contents for
         detailed help"));
```

When you are in a window that has control and you press F1 you will see this text pop up in a neat compact frame window. It will disappear on a single mouse click.

To use the traditional Context Help cursor (the ? And pointer) we need to issue a context help command.

This is accomplished by associating a menu entry or toolbar button with `wxID_HELP_CONTEXT`.

When the context message is received do the following:

```
BEGIN_EVENT_TABLE(WXWindPlotFrame,
                  wxDocMDIParentFrame)
  EVT_MENU(wxID_HELP_CONTEXT,
           WXWindPlotFrame::OnCHelp)
END_EVENT_TABLE()

void WXWindPlotFrame::OnCHelp() {
  wxContextHelp chp(this, TRUE);
}
```

The help controller will handle the rest for you. The Cursor will change and you can take it where you will. Left Click to see the help text

The First Linux Compile

The main weapon of choice here is the preprocessor. While we are trying to make our code as portable as we can, some things will have to be done a little differently.

Using the preprocessor command we can eliminate Unix code from Windows compiles:

```
#ifndef WXMS_
—- Unix specific commands
#endif
```

Now comes the moment of truth. Prepare to be humbled and pay for all those sloppy little habits you picked up when using MSVC++.

We are moving over to Linux which though not entirely unfriendly is less forgiving in many respects. Firstly wxWidgets needs to be installed.

On RedHat 9 this went without a problem – I downloaded the GTK+ tarball and unzipped it into a working directory.

Following the instructions: `./config`, `make` and then a `make install` (the last one as root), we were up and running

wxWidgets sets up library and include paths. The command `ldconfig -v` sets up the linker paths and you should see `libwx_gtk` somewhere in there.

We need to be sensitive to shared libraries. I first tried to compile and link everything statically. With GTK this is not possible. Most of the high level widgets need to be loaded dynamically at run time. We can link in all the wx libraries though so when you configure use the command:

```
./configure –with-gtk –disable-shared
```

Now we need to get familiar with the development environment. I use kdevelop as this has a lot of similar functionality to MSVC. I am a GUI junkie and am more productive when I have a tool to relate messages to code.

A good move now is to try to run up one of the samples.

Follow These Steps

Navigate to the samples directory of the wx distribution and look for a project. DocViewMDI is a good one. Rename the file `makefile.unx` to `Makefile` and we have a make environment ready to go.

Now run up kdevelop and from the project menu generate a project file in same directory. You should now be able to press the cogwheel toolbar button to compile and run the program and verify that your wx build environment is ready to go.

Now on to the Port

MSVC is a lot better when it comes to managing projects and files and as a result my project was spread over several folders. I was re-using source code in several projects and had a common source folder. This is OK if you are on your own but in a commercial environment with multiple programmers it is intolerably sloppy. Common modules should be statically linked into a library and then included on the link path. Include files should be in a common include directory. Then all you need to do is put all your C++ code in a working directory and all the wdr generated includes in the `.wdr/` folder.

The construct of the makefile becomes very simple in this case and here it is:

```
# File: Generic Makefile for wxWidgets under GTK
CXX = $(shell wx-config –cxx)
PROGRAM = WxWindplot
#OBJECTS =  WXWindPlotApp.o *.o
OBJECTS =  $(patsubst %.cpp, %.o, $(SOURCES))
SOURCES = $(wildcard *.cpp)

# implementation
.SUFFIXES:  .o .cpp
.cpp.o :
  $(CXX) -c 'wx-config –cxxflags' -o $@ $<
### Uncomment next line if you need debugging
### information
###  $(CXX) -c 'wx-config –cxxflags' -g -o $@ $<

all:    $(PROGRAM)
$(PROGRAM): $(OBJECTS)
  $(CXX) -o $(PROGRAM) $(OBJECTS) 'wx-config –libs
                                          --static'
clean:
  rm -f *.o $(PROGRAM)
```

This makefile compiles all `.cpp` programs in the same directory and links them. If you uncomment the debugging line and comment out the line above you will get debugging information.

So Off We Go

Most of the errors you get will be due to include files not properly resolved. There will also be a few MS specific library calls and you will need to find ANSI equivalents or look in the wxWidgets documentation. Specific examples will be platform issues life file name resolution, variable persistence etc.

Remember Unix/Linux file names are case sensitive. Very soon all your source files will be lower case. Microsoft does not care and allows mixing of cases.

One thing to be aware of on the port. I have been quite sloppy about defining variables on the fly. MSVC has a slightly different visibility rule on dynamic variables. It is good practice to follow the old C rule of declaring local variables at the beginning of the subroutine to avoid porting problems. If you declare variable `i` within a `for` loop

```
for(int i=0 ; i<20 ; i++)
```

and you reference `i` again you will get an error.

```
int i;
for(i=0;i<20;i++)
```

The above is a better way of doing things.

Referencing the contents of `wxString` – always use `wxString::GetData()` and `wxString::GetChar(i)` as opposed to referencing the data directly.

Put on your flying goggles and go to work. For my 5,000 line windplot project over 10 cpp files we had clean compile in 4 hours. It would have been faster had I done this before. Running the program – Wow it ran first time. Not perfect but we were cooking. More important I was able to single step with kdevelop and get an idea where the problems lay.

Where classes have integral data types – `wxString` `wxLonLong` – you need to use the access functions rather than the classes themselves `wxLongLong` you need to pull the long value via `wxLongLong::ToLong`. Very important if you are using time variables where everything is `LongLong` (takes us over the next rollover event in 2034).

Regrettably a lot of the sexier features of MS Windows are not all implemented on GTK and especially in things like MDI windows there are some things lacking. The next step will be to look at these deficiencies and see how we can address them.

The Tuneup: Basic Debugging

When you get to work with Linux you experience a true multi threaded multitasking environment. Compared to Windows where your primary thread will hang on a dialog until you respond. Be prepared for some very fast lock ups if you put a modal dialog in a the `OnDraw` loop for instance.

I was having a dreadful time trying to sort out Windows Sockets and and my breakpoints seemed to be activated at random until I got the hang of what was going on. From then on it was plain sailing and kdevelop was a big help. Looking inside classes was not possible with the way I was using kdevelop and so I have to put in debugging statements to pull values out of `wxString`s to see them. For those who are used to an IDE, kdevelop worked very well. Above all the price is right.

The problems I encountered on my first run:

File visibility: This is very important and you need to agree on where your working files will be based.

Month 0 based: July was August until I put in a conditional compile statement to decrement the month.

Internet Hang-up: Serial Port Access.

Could not open a COM port (`/dev/cua0`). Using `setserial -G /dev/cua0` I verified the port was valid and then logged on as root and gave read/write access to all. After that everything worked. *[The name of the ports will vary – the serial ports on my linux box are all ttySx - Ed]*

Back to MSVC

The last job was to do a release build and put the windows version on the web. A nasty shock, every function crashed horribly. In debug it was perfect. What to do ?

Finally after looking through the support base, I recompiled wxWidgets specifically disabling optimizations and after the libraries were re-built the code worked perfectly. All that is needed is to go into the Project menu and in the settings go to the C++ tab and set optimizations to Disable (Debug). I had visions of a huge rewrite but fortunately a simple fix was all that was required.

The most important thing I found in all this was that the underlying thought process was similar so re-training in wxWidgets from MFC is a very easy and refreshing process. The Class Wizard and App Wizard that MFC prides itself on is in fact a great snare and a delusion. The work it saves you is quite trivial. I am convinced that this toolkit will be with us for a long time. As open source, its future is assured. The modified GPL license it is released under means that it will be attractive to commercial operations and it could be one of the levers that finally puts Linux in the forefront where it belongs.

Jonathan Selby

[resources section at foot of next page]

Elephant – A C++ Memory Observer

Paul Grenyer <paul@paulgrenyer.co.uk>

What is Elephant?

Elephant is a C++ memory observer. It keeps track of all calls to `new` and `delete` via custom implementations of `operator new` and `operator delete`. Observers can register to be notified of allocations and deletions and used to detect memory leaks, keep a track of maximum memory usage or for any other purpose, by implementing a simple interface.

A notification of an allocation consists of the address and size of the memory allocated. The line number, function name and file name in which the allocation takes place can be added by placing special macros in the client code. A notification of a deletion consists of the address of the memory being freed.

Elephant is not intended to ship in production code. It is intended as a debugging aid. Elephant's functionality can be removed simply by relinking without the Elephant static library. All other code can remain in place.

Elephant comes with a complete, Aeryn (`http://www.paulgrenyer.co.uk/aeryn`) based test suite to test that it behaves correctly on any given platform.

Where Can I Get Elephant?

Elephant is available for download from:
`http://www.paulgrenyer.dyndns.org/elephant/`

What Do I Need To Build Elephant?

Elephant uses up-to-date C++ techniques (including member function templates using the Aeryn unit tests), as well as some classes based on parts of Andrei Alexandrescu's Loki library (`http://sourceforge.net/projects/loki-lib/`) and therefore requires a modern compiler. It has been tested on, and provides make files or project files for the following compilers:

- Microsoft Visual C++ 7.1
- MinGW 3.2.3
- GNU G++ 3.2.3

It may be possible to get Elephant to compile on Microsoft Visual C++ 6.0.

How Do I Build Elephant?

Elephant consists of a group of headers and a static library. The full source is supplied with Elephant and the static library must be built. Building the elephant static library couldn't be easier:

Microsoft Visual C++ 7.1

To build the Elephant library, unit tests and the (test) supporting Aeryn library with Microsoft Visual C++ 7.1, simply open the Elephant solution located in the top level Elephant directory and select Build Solution from the Build menu.

To run the unit tests right click on the `TestClient` project in the Solution Explorer and select Set as StartUp Project, then select Start Without Debugging from the Debug menu. This should give you the following output:

```
Aeryn 0.4.0 beta (c) Paul Grenyer 2004
http://www.paulgrenyer.co.uk/aeryn
-----------------------------------
Ran 21 tests, 21 Passed, 0 Failed.
Press any key to continue
```

MinGW

To build the Elephant library, unit tests and the (test) supporting Aeryn library with MinGW open a command prompt and navigate to the top level Elephant directory. Making sure that the MinGW `bin` directory is in your path, type:

```
mingw32-make
```

To run the unit tests type the following:

```
bin\TestClient.exe
```

This should give you the following output:

```
Aeryn 0.4.0 beta (c) Paul Grenyer 2004
http://www.paulgrenyer.co.uk/aeryn
-----------------------------------
Ran 21 tests, 21 Passed, 0 Failed.
```

For `mingw32-make clean` to work correctly the `rm` tool from MSYS or cygwin must also be in your path.

g++

To build the Elephant library, unit tests and the (test) supporting Aeryn library with g++ open a command prompt and navigate to the top level Elephant directory. Checking that g++ and make are both installed correctly, type:

```
make
```

To run the unit tests type the following:

```
bin/TestClient.exe
```

This should give you the following output:

```
Aeryn 0.4.0 beta (c) Paul Grenyer 2004
http://www.paulgrenyer.co.uk/aeryn
-----------------------------------
Ran 21 tests, 21 Passed, 0 Failed.
```

The current version of Elephant was tested with g++ 3.2.3 on Red Hat Linux ES 3.0. If any of the tests fail on your platform Elephant may not work as expected. If you do have tests that fail, please send me the complete Aeryn output along with details of your g++ version and operating system.

How Do I Set Up My Environment To Use Elephant?

Before you can use Elephant, the Elephant static library must be built (see previous section):

Microsoft Visual C++ 7.1

`Elephant_debug.lib` (debug)
`Elephant.lib` (release)

MinGW

`libelephant.a`

g++

`libelephant.a`

Regardless of which compiler or platform is used the Elephant library is places in the `bin` directory which a subdirectory of the Elephant top level directory.

Your environment also needs to have access to the Elephant `include` directory which is a subdirectory of the top level Elephant directory. The actual Elephant include files are stored in further subdirectories called `elephant` and `tools` (`tools` is a subdirectory of `elephant`). This is so that Elephant include files can be identified from other include files which might share the same name. For example:

```
#include <elephant/newdelete.h>
```

Microsoft Visual C++ 7.1

Once you have created a solution containing the project which is going to use Elephant to monitor memory usage, you are ready to add Elephant to your environment.

There are at least two ways to add the Elephant static library to your solution:

Method 1: Add the ElephantLib project to the solution.

This method has the advantage that the `ElephantLib` project is included in a rebuild all.

1. Right click the solution name in Solution Explorer and select Add Existing Project from the Add menu item.
2. Navigate to the `ElephantLib` directory which is a subdirectory of the Elephant top level directory.
3. Select `ElephantLib.vcproj` and click open. (This will add the Elephant library project to your solution.)
4. Right click your project and select Project Dependencies from the menu. Then put a tick in the ElephantLib box and click Ok.

[continued from previous page]

Resources

wxWidgets : `http://www.wxwidgets.org`
wxDesigner : `http://www.roebling.de/`

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

Method 2: Add the Elephant static library directly to the project.

1 Right click your project and select properties.
2 Set the Configuration drop-down box to All Configurations.
3 Select the Linker folder and then the General item in the tree view.
4 Enter the path to the Elephant static libraries (`Elephant\bin`) into the Additional Library Directories box.
5 Set the Configuration drop-down box to debug.
6 Select the Input item from the Linker folder in the tree view.
7 Enter `Elephant_debug.lib` into the Additional Dependencies box.
8 Set the Configuration drop-down box to release.
9 Enter `Elephant.lib` into the Additional Dependencies box.
10 Click Ok

To make the Elephant headers available to your project in your solution follow these steps:

1 Right click your project and select properties.
2 Set the Configuration drop-down box to All Configurations.
3 Select the C/C++ folder and then the General item in the tree view.
4 Enter the path to the Elephant include files (`Elephant\include`) into the Additional Include Directories box.
5 Click Ok.

MinGW & g++

This description of configuring MinGW and g++ to link to Elephant assumes that you are using a make file to build your project. Of course this is not the only way.

To link Elephant to your executable (or shared library etc) two extra parameters need to be added to your link command: the path to the Elephant static library, preceded by `-L` and the name of library, preceded by `-l`. For example:

```
g++ myproj.o -LElephant/bin -lelephant myproj
```

The Elephant include files must be made available to every invocation of g++ that builds a source (cpp) file that includes, directly or indirectly, an Elephant include file. This is done by adding a single parameter, which consists of the path to the Elephant include directory preceded by `-I`. For example:

```
g++ -c -o myproj.o myproj.cpp -IElephant/include
```

How Do I Use Elephant In My Program?

Assuming that you have built the Elephant static library and integrated it into your environment (see previous two sections) you are now ready to use Elephant in your program.

`operator new` and `operator delete`

The custom implementations of `operator new` and `operator delete` are the key to Elephant's ability to monitor memory. There are overloads for the normal and array versions with corresponding no throw versions.

To use the Elephant's custom `new` and `delete` operators simply include the `newdelete.h` header in your program. For example:

```
#include <elephant/newdelete.h>
int main() {
  return 0;
}
```

It only needs to be included once, although multiple inclusions will not do any harm.

Every time a call is made to `new` or `delete` the Elephant operator overloads will register the call with the Elephant memory monitor. The Elephant memory monitor is observerable and you can register one of the provided observers or write your own to react to the allocations and de-allocations.

Example 1: Observing and Reporting a Memory Leak

Let's start of with a simple example of a memory leak:

```
#include <elephant/newdelete.h>
class SomethingToAllocate {};
int main() {
  SomethingToAllocate* p
                  = new SomethingToAllocate;
  return 0;
}
```

This program will compile and run and you will see absolutely no indication of the memory leak. In order to detect the memory leak you need the leak detector class, `LeakDetector`. The leak detector class is an observer of the memory monitor, so you need to register and unregister it as an observer:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
class SomethingToAllocate {};

int main() {
  using namespace elephant;
  LeakDetector leakDetector;
  // Register leak detector with memory monitor.
  MemoryMonitorHolder().Instance().AddObserver(
                                    &leakDetector);
  SomethingToAllocate* p = new SomethingToAllocate;
  // Unregister leak detector with memory monitor.
  MemoryMonitorHolder().Instance().RemoveObserver(
                                    &leakDetector);
  return 0;
}
```

To use the memory monitor and the leak detector you need to include the appropriate header files as shown. Running this program will still not indicate that there is a memory leak. To indicate the memory leak you need to interrogate the `LeakDetector` instance. For example:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
#include <cassert>
class SomethingToAllocate {};

int main() {
  using namespace elephant;
  LeakDetector leakDetector;
  // Register leak detector with memory monitor.
  MemoryMonitorHolder().Instance().AddObserver(
                                   &leakDetector);
  SomethingToAllocate* p = new SomethingToAllocate;
  // Unregister leak detector with memory monitor.
  MemoryMonitorHolder().Instance().RemoveObserver(
                                    &leakDetector);
  assert(!leakDetector.IsLeak());
  return 0;
}
```

The `assert` (which required the `cassert` header as shown) will indicate that a memory leak has occurred. This particular method of indicating a memory leak isn't particularly useful. The next step is to print the memory address and the size of the leak:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
#include <elephant/leakdisplayfunc.h>
#include <algorithm>
class SomethingToAllocate {};

int main() {
  using namespace elephant;
  LeakDetector leakDetector;
  // Register leak detector with memory monitor.
  MemoryMonitorHolder().Instance().AddObserver(
                                    &leakDetector);
  SomethingToAllocate* p = new SomethingToAllocate;
  // Unregister leak detector with memory monitor.
  MemoryMonitorHolder().Instance().RemoveObserver(
                                    &leakDetector);
  // Display the details of the leak.
  LeakDisplayFunc leakDisplay(std::cout);
  std::for_each(leakDetector.begin(),
                leakDetector.end(), leakDisplay);
  return 0;
}
```

The `LeakDisplayFunc` class constructor takes a reference to an output stream and has a function operator that can be used, as shown, to the write memory leak information to the stream. As

LeakDisplayFunc uses an output stream it is possible that memory will be allocated and not freed until the end of main. This is why the leak detector must be unregistered before the memory leak information is displayed. Otherwise the output stream allocation will appear as a further memory leak. One way to avoid having to unregister the LeakDetector is to write your own function object that displays the memory leak information without allocating memory using new. For example using printf.

The output from this program should be as follows, although the address will be a different value:

```
Address:        00320B70
Size:           1
```

Example 2: Recording Line and Filename of Allocation

In the previous example the memory leak was displayed as a memory address and a size. This can be useful in finding a memory leak, but not as usual as tracking the exact site of the allocation. Elephant can do this by introducing a special macro into every translation unit where this type of tracking is needed. The macro is called ELEPHANTNEW and can be included anywhere in the translation unit. The following code shows how the macro would be added to example 1:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
#include <elephant/leakdisplayfunc.h>
#include <algorithm>

#define new ELEPHANTNEW

class SomethingToAllocate {};
int main() {
  ...
}
```

The output should now look something like this:

```
Address:        00322878
Size:           1
Line:           22
Function:       main
File:           c:\...\example2\main.cpp
```

Some compilers, such as Microsoft Visual C++ 7.1 will show a fully qualified function name and a complete a full file path. Other compilers, such as g++ and MinGW will show only the local function name and file name without the full path. For example:

```
Address:        0x3d24f0
Size:           1
Line:           23
File:           main.cpp
```

Example 3: Using the Maximum Memory Observer

The other memory observer supplied with Elephant, MaxMemoryObserver, is for measuring the maximum amount of memory used at anyone time by an application. Its use is very similar to that of LeakDetector:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/maxmemoryobserver.h>
class SomethingToAllocate {};

int main() {
  using namespace elephant;
  MaxMemoryObserver maxMemory;
  // Register max memory observer with memory monitor.
  MemoryMonitorHolder().Instance().AddObserver(
                                        &maxMemory);
  SomethingToAllocate *p1
                    = new SomethingToAllocate[100];
  delete[] p1;
  SomethingToAllocate *p2
                    = new SomethingToAllocate[50];
  delete[] p2;
  // Unregister max memory observer.
  MemoryMonitorHolder().Instance().RemoveObserver(
                                        &maxMemory);
  // Display the max memory usage
  std::cout << "Max memory usage: "
            << static_cast<unsigned long>(
                                maxMemory.MaxMemory())
            << " bytes\n";
  return 0;
}
```

The output from this simple (not very exception safe) example is as follows:

```
Max memory usage: 100 bytes
```

The size of SomethingToAllocate is 1 byte. During the execution of the program a total of 150 SomethingToAllocate instances are created and destroyed. However, the program only has up to 100 instances allocated at any one time. Therefore the maximum amount of memory used by the program is 100 bytes.

Example 4: Writing a Custom Memory Observer (Part 1)

Elephant can be used for more than just detecting memory leaks and the maximum memory used by a program. Elephant can be used to monitor any characteristic of new and delete based memory usage via custom memory observers. Custom memory observers are simple to create. All that is required is the implementation of the following interface:

```
namespace elephant {
  class IMemoryObserver {
  protected:
    IMemoryObserver();
  public:
    virtual ~IMemoryObserver() = 0;
    virtual void OnAllocate(void* p, size_t size,
                  size_t line, const char* file) = 0;
    virtual void OnFree(void* p) = 0;
  private:
    IMemoryObserver(const IMemoryObserver&);
    IMemoryObserver& operator=(
                           const IMemoryObserver&);
  };
}
```

All that needs to be done to implement the interface is to inherit from it and override the OnAllocate and OnFree pure virtual member functions. The OnAllocate function has the following arguments:

p – A pointer to the memory that has been allocated. This is useful for getting the address.

size – The size of the memory that has been allocated.

line – The line number on which the memory was allocated. This is 0 unless the ELEPHANTNEW macro has been used correctly.

char – The file in which the memory was allocated. This is an empty string unless the ELEPHANTNEW macro has been used correctly.

The OnFree function has the following argument:

p – A pointer to the memory that has been allocated. This is useful for getting the address.

The default constructor of the interface is protected to show that the class should be inherited from. The copy constructor and assignment operator are private to prevent attempts to copy the interface or its subclasses (unless the subclasses define their own copy constructor and assignment operator) and the destructor is virtual to ensure proper destruction should a dynamically allocated subclass by destroyed via a pointer to the interface.

The example below is of a simple custom observer which records the total memory allocated by a program during its lifetime:

```
class TotalMemoryobserver : public
elephant::IMemoryObserver {
private:
  size_t totalMemory_;
public:
  TotalMemoryobserver()
      : totalMemory_(0) {}
  virtual void OnAllocate(void* p, size_t size,
                  size_t line, const char* file) {
    totalMemory_ += size;
  }
  virtual void OnFree(void* p) {}
  size_t TotalMemory() const {
    return totalMemory_;
  }
}
```

The `OnAllocate` override is used to accumulate the size of every allocation. The other parameters are ignored as they are not needed. The `OnFree` function does nothing as we are not interested in de-allocations. In, for example, the leak detector, the value of `p` passed to `OnFree` is used to match against a previous value of `p` passed to `OnAllocate` to show that the memory has been deleted.

Replacing `MaxMemoryObserver`, from the previous example, with `TotalMemoryobserver` and making a couple of other minor changes:

```
int main() {
  using namespace elephant;
  TotalMemoryobserver totalMemory;
  // Register max memory observer with memory monitor.
  MemoryMonitorHolder().Instance().AddObserver(
                                    &totalMemory);
  SomethingToAllocate *p1
                  = new SomethingToAllocate[100];
  delete[] p1;
  SomethingToAllocate *p2
                  = new SomethingToAllocate[50];
  delete[] p2;
  // Unregister max memory observer.
  MemoryMonitorHolder().Instance().RemoveObserver(
                                    &totalMemory);
  // Display the max memory usage
  std::cout << "Total memory usage: "
            << static_cast<unsigned long>(
                        totalMemory.TotalMemory())
            << " bytes\n";
  return 0;
}
```

gives the following output, which correctly indicates the total memory used by the program:

```
Total memory usage: 150 bytes
```

Example 5: Writing a Custom Memory Observer (Part 2)

Sometimes you want to store information about allocations and de-allocations in a container within a custom memory observer. Containers do of course allocate memory in order to contain. This could lead to erroneous memory usage observations and, in a worst case scenario, infinite recursion.

The simple answer is to use a container that uses `malloc` and `free` instead of `new` and `delete`. Or, to be more precise, a container that uses an allocator that allocates with `malloc` and `free` instead of `new` and `delete`. Elephant comes with just such an allocator, called `malloc_allocator`, which can be used with any of the C++ standard library containers. It should be used as follows:

```
#include <elephant/tools/mallocallocator.h>
#include <vector>

...

std::vector<size_t,
      elephant::tools::malloc_allocator<size_t> >
      allocStore;
```

Naturally a `typedef` can make life a lot easier.

The following example shows a custom memory observer that uses a container with the `malloc_allocator` to store two lists of the addresses, allocations and de-allocations:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/imemoryobserver.h>
#include <elephant/tools/mallocallocator.h>
#include <vector>

class AllocationMemoryobserver : public
                        elephant::IMemoryObserver {
private:
  typedef std::vector<void*,
        elephant::tools::malloc_allocator<void*> >
        MAllocContainer;
  typedef MAllocContainer::const_iterator
        const_iterator;
  MAllocContainer allocations_;
  MAllocContainer deallocations_;
  void Print(const MAllocContainer& cont,
             std::ostream& out) {

    const_iterator current = cont.begin();
    const_iterator end   = cont.end();
    for(; current != end; ++current) {
      out << "\t" << (*current) << "\n";
    }
    out << "\n";
  }
public:
  AllocationMemoryobserver() : allocations_(),
                               deallocations_() {}
  virtual void OnAllocate(void* p, size_t size,
                  size_t line, const char* file) {
    allocations_.push_back(p);
  }
  virtual void OnFree(void* p) {
    deallocations_.push_back(p);
  }
  void PrintAllocations(std::ostream& out) {
    out << "Allocations:\n";
    Print(allocations_, out);
  }
  void PrintDeallocations(std::ostream& out) {
    out << "Deallocations:\n";
    Print(deallocations_, out);
  }
};

class SomethingToAllocate {};

int main() {
  using namespace elephant;
  AllocationMemoryobserver allocationObserver;
  // Register max memory observer with memory monitor.
  MemoryMonitorHolder().Instance().AddObserver(
                                &allocationObserver);
  SomethingToAllocate *p1
                  = new SomethingToAllocate[100];
  SomethingToAllocate *p2
                  = new SomethingToAllocate[50];
  delete[] p2;
  delete[] p1;
  // Unregister max memory observer.
  MemoryMonitorHolder().Instance().RemoveObserver(
                                &allocationObserver);
  allocationObserver.PrintAllocations(std::cout);
  allocationObserver.PrintDeallocations(std::cout);
  return 0;
}
```

The output from this example is as follows:

```
Allocations:
        00322850
        00322910
Deallocations:
        00322910
        00322850
```

If `malloc_allocator` is replaced by the default allocator, there is no output, not even an error message, with both Microsoft Visual C++ and MinGW.

Elephant and Threading

Elephant has not yet been tested in a multithreaded environment.

The use of the `Mutex` class and its various implementations are based on previously known working examples.

Offers to test Elephant in a multithreaded environment will be gratefully accepted.

By default, Elephant is not thread safe. The `mutex.h` header file is included in a number of places and the `Mutex` class, along with the `Guard` class (for exception safety) is used to protect those parts of the library that may cause problems if accessed by two threads at the same time.

[concluded at foot of next page]

An Introduction to Objective-C

Part 4 – Some Further Topics

D.A. Thomas

Type Introspection

Objective-C has a rich set of methods by which the contents and capabilities of an object can be queried. `NSObject` implements:

`(Class)class` returns the class object for the receiver's class.

`(Class)superclass` returns the class object for the class from which the receiver inherits.

`(BOOL)isMemberOfClass:(Class)class` returns YES if the argument to the method is an instance of the specified class.

`(BOOL)isKindOfClass:(Class)class` returns YES if the argument to the method is an instance of the specified class or of a class that inherits from it.

`(BOOL)respondsToSelector:(SEL)aSelector` returns YES if the receiving object is capable of handling a certain message.

There are also functions to query classes for their instance variables and class and instance methods, and methods can be queried for information about their arguments.

Extensions

NeXT and Apple have extended the language specified by Cox in "Object-Oriented Programming, an Evolutionary Approach" with categories, protocols and, most recently, Java-style exception-handling, thread synchronisation and support for invoking methods in remote processes. The last two are considered too specialised to be dealt with in this article.

Categories

Categories add new functionality to an already existing class. They are particularly useful where you are using a third-party class library and you are not free to amend that library's source code. One solution to this problem is to derive a new class from the one you need to extend, but this may require detailed knowledge of the superclass, and inheritance notoriously breaks encapsulation. To create a category, you declare interface and implementation sections as shown in the pseudocode below:

In the header file, `CategoryName.h`:

```
#import "ClassName.h"
@interface ClassName (CategoryName)
method declarations
@end
```

In `CategoryName.m`:

```
#import "CategoryName.h"
@implementation ClassName (CategoryName)
method definitions
@end
```

A class has a size that is fixed at compilation time, so it is not possible to add instance variables to an existing class in this manner; the only way to do this is to use inheritance.

The file `StringTokenizer.m` contains the following lines:

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

Since `-skipDelimiters` is not meant to be directly accessible to the users of a class, it would be inappropriate to declare it in `StringTokenizer.h`, and so I have created a category in the implementation file to contain declarations of private methods. This is not strictly necessary, as an Objective-C compiler emits a warning, not an error, when it is required to compile a message to an undeclared method, and since the programmer knows that the method has been defined, the program would work perfectly well without such a category declaration.

Categories can also be used to split up the implementation of a class into separate units, with perhaps each having its own implementation file; this would facilitate the development of classes to which more than one programmer contributes. They can also be used to declare informal protocols, of which more below.

Protocols

Protocols involve the declaration of a list of methods whose implementation is deferred to any class that chooses to implement them. If a class adopts an informal protocol, it can choose which methods to implement, whereas with a formal protocol, implementations of all the methods listed must reside either in the class itself or in its superclasses. This is a way of associating classes that share similar behaviour but are not closely related in the inheritance hierarchy.

Informal Protocols

There is little language support for informal protocols, but in the Foundation framework, informal protocols are often declared as a category of the root class, `NSObject`. Here is the list of methods in GNUStep's version of Foundation for the informal protocol `NSKeyValueCoding`, which defines a mechanism in which the properties of an object are accessed indirectly by name (or key), rather than directly through invocation of an accessor method or as instance variables:

[continued from previous page]

Elephant Mutexes

If you open the `mutex.h` header file, you will see it looks like this:

```
#ifndef ELEPHANT_TOOLS_MUTEX_H
#define ELEPHANT_TOOLS_MUTEX_H
#include <elephant/tools/nullmutex.h>
//#include <elephant/tools/boostmutex.h>
//#include <elephant/tools/win32mutex.h>
#endif // ELEPHANT_TOOLS_MUTEX_H
```

There are three types of mutex supplied with Elephant:

Null Mutex

An empty mutex class intended for use in single threaded programs so that no performance is lost creating, entering or leaving an unnecessary mutex.

Win32 Mutex

Implemented using the Win32 API for use with Windows compilers only.

Boost Mutex

A mutex implemented using `boost::mutex` (`http://boost.org/libs/thread/doc/mutex_concept.html`).

The Null Mutex is used by default. To use one of the other mutexes simply include its header file in `mutex.h` instead of `nullmutex.h` and rebuild (a rebuild all is recommended).

Custom Mutexes

A custom mutex can be written simply by implementing the following class in its own header file and including it in `mutex.h` instead of the other mutex header files:

```
namespace elephant {
  namespace tools {
    class Mutex {
    public:
      Mutex() {}
      ~Mutex() {}
      void Enter() const {}
      void Leave() const {}
    private:
      Mutex(const Mutex&);
      Mutex& operator=(const Mutex&);
    };
  }
}
```

Note: As the `Enter` and `Leave` member functions are `const`, you may need to make the object that holds the current state of the mutex `mutable`.

Where Next?

This is the very first beta release of Elephant. Therefore I expect I, and hopefully other people, will find plenty of bugs or new features that should be implemented, over the coming months.

So far, planned for future releases:

- Threading testing and unit tests.
- Black and white allocation lists
- Client memory tracking

Paul Grenyer

```
@interface NSObject (NSKeyValueCoding)
+ (BOOL) accessInstanceVariablesDirectly;
+ (BOOL) useStoredAccessor;
- (id) handleQueryWithUnboundKey: (NSString*)aKey;
- (void) handleTakeValue: (id)anObject
                  forUnboundKey: (NSString*)aKey;
- (id) storedValueForKey: (NSString*)aKey;
- (void) takeStoredValue: (id)anObject forKey:
                  (NSString*)aKey;
- (void) takeStoredValuesFromDictionary:
                  (NSDictionary*)aDictionary;
- (void) takeValue: (id)anObject forKey:
                  (NSString*)aKey;
- (void) takeValue: (id)anObject forKeyPath:
                  (NSString*)aKey;
- (void) takeValuesFromDictionary:
                  (NSDictionary*)aDictionary;
- (void) unableToSetNilForKey: (NSString*)aKey;
- (id) valueForKey: (NSString*)aKey;
- (id) valueForKeyPath: (NSString*)aKey;
- (NSDictionary*) valuesForKeys: (NSArray*)keys;
@end
```

Any object that derives from `NSObject` can select from this list which methods it needs to implement in order to acquire appropriate key-value coding functionality.

Formal Protocols

Formal protocols are enforced by the language. They are declared as in the following pseudocode:

```
@protocol ProtocolName
method declarations
@end
```

Here is a declaration for the `NSCoding` protocol for the serialisation ('flattening') and deserialisation (reconstruction) of objects associated with archiving from disk or some other form of distribution to another address space.

```
@protocol NSCoding
- (void) encodeWithCoder: (NSCoder*)aCoder;
- (id) initWithCoder: (NSCoder*)aDecoder;
@end
```

If the class `Person` needed to be stored on disk, it would adopt the `NSCoding` protocol:

```
#import <Foundation/Foundation.h>
@interface Person : NSObject <NSCoding>
{
  NSString *name;
  NSString *address;
}
// Accessor methods
- (NSString *)name;
- (NSString *)address;
- (void)setName:(NSString *)aName;
- (void)setAddress:(NSString *)anAdress;
// Other methods ...
@end

@implementation Person
// Accessor methods
- (NSString *)name {return name;}
- (NSString *)address {return address;}
- (void)setName:(NSString *)aName
{
  [aName retain];
  [name release];
  name = aName;
}
- (void)setAddress:(NSString *)anAdress
{
  [anAddress retain];
  [address release];
  address = anAdress;
}
// NSCoding methods
- (void) encodeWithCoder: (NSCoder*)aCoder
{
  [super encodeWithCoder:coder];
  [aCoder encodeObject:name];
  [aCoder encodeObject:address];
}
- (id) initWithCoder: (NSCoder*)aDecoder;
{
  self = [super initWithCoder:coder];
  name = [[coder decodeObject] retain];
  address = [[coder decodeObject] retain];
  return self;
}
// Called when the object is deallocated
- (void) dealloc {[name release]; [address release]}
@end
```

Formal protocols are equivalent to interfaces in Java; indeed, the designers of Java have copied this idea from Objective-C. Assuming that Foundation had been implemented in C++ you would write something like the following abstract class definition:

```
class NSObject;
class NSCoding {
  virtual void encodeWithCoder(NSCoder& aCoder) = 0;
  virtual NSObject* decodeWithCoder(
                          const NSCoder& aCoder) = 0;
};

class Person : public NSObject, public NSCoding {
  NSString *name_, *address_;
public:
  // Accessor functions
  NSString* name();
  void setName(const NSString* aName);
  NSString* address();
  void setAddress(const NSString* anAddress);
  // NSCoder virtual functions
  void encodeWithCoder(NSCoder& aCoder);
  NSObject* decodeWithCoder(const NSCoder& aCoder);
  // Other functions ...
  // Called when the object is deallocated
  virtual ~Person();
};
```

The implemention of these methods in C++ is left to the reader's imagination.

Unlike C++, neither Objective-C nor Java implements multiple inheritance, and so these languages need a separate mechanism for adopting protocols.

It cannot always be known at run-time whether a particular object implements a formal protocol; it can be tested in the following way:

```
if([anObject conformsTo:@protocol(NSCoding)])
   [anObject encodeWithCoder:myCoder];
```

Exceptions

Simple exception-handling code could be written as follows;

```
Cup *cup = [[Cup alloc] init];
@try {
  [cup fill];
}
@catch (NSException *exception) {
  NSLog(@"main: Caught %@: %@", [exception name],
[exception reason]);
}
@finally {
  [cup release];
}
```

Code that might throw an exception is enclosed within a `@try` block, and the exception should be caught in a `@catch` block. A `@finally` block contains code that must be executed whether an exception is thrown or not.

`Cup`'s `fill` method might throw an exception like this:

```
NSException *exception = [NSException
exceptionWithName:@"HotTeaException"
          reason:@"The tea is too hot" userInfo:nil];
@throw exception;
```

Any kind of Objective-C object can be thrown.

An exception can be re-thrown by means of `@throw` without an argument.

D. A. Thomas

Memory For a Short Sequence of Assignment Statements

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

This is the second of a two part article describing an experiment carried out during the 2004 ACCU conference. The previous part was published in the previous issue of C Vu. This second part discusses how the `if` statement part of the problem affected subject performance.

The `if` statement problem can be viewed as either a time filler for the assignment remember/recall problem, or as the main subject of the experiment (with the assignment problem acting as a smoke screen to make it more difficult for subjects to notice any patterns in the `if` problems). The reason for this second possibility is that studies have found patterns in the errors made by subjects when performing various kinds of deduction tasks.

Given that some kind of filler task had to be performed, your author decided to take opportunity to try and replicate some of the error patterns seen in some studies of deduction.

As Table 1 shows, relational operators commonly occur in `if` statements.

Operator	% Controlling Expression	% Occurrence of Operator
==	31.7	88.6
!=	14.1	79.7
<	6.9	45.6
<=	1.9	68.6
>	3.5	84.9
>=	3.5	76.8
no relational/equality	47.5	–
||	9.6	85.9
&&	14.5	82.3
no logical operators	84.2	–

Table 1 – Occurrence of equality, relational, and logical operators in the conditional expression of an `if` statement (as a percentage of all such controlling expressions and as a percentage of the respective operator). Based on the visible form of over 3 million lines of C source. The percentage of controlling expressions may sum to more than 100% because more than one of the operators occurs in the same expression.

Linear Syllogisms

The psychology of deduction uses the terms *linear syllogisms* or *linear reasoning* to describe deduction between statements involving relational operators. The term usually used to describe a (sub)expression containing a relational operator, in programming language specifications, is *relational expression*.

Linear syllogisms are part of mathematical logic and the skills associated with being able to make deductions based on relational information are usually assumed simply to be a component of the general reasoning ability that people have. However, studies have found that a number of animals have the ability to adapt their behaviour to given situations based on relational knowledge they have acquired. For instance, aggressive behaviour may occur between two animals to determine which is dominant, relative to the other. Such behaviour can lead to being injured in a fight and is best avoided if possible. The ability to make use of relative dominance information (e.g., obtained by a member of a social group watching the interaction between other members of the group) may remove the need for aggressive behaviour during an encounter between two members of the same group who have not met face to face before (i.e., the member most likely to lose immediately behaves in a subservient fashion).

One study [1] allowed a social dominance hierarchy to become established in several independent groups of birds (Pinyon jays). Two birds from different groups were then placed in a cage and given time to establish their relative social dominance (a process that involves staring, looking away, chin-up and beg, etc). The interaction of the two birds was witnessed by a bird belonging to one of the two groups from which the two birds came (this bird could not participate in any social interaction with the birds it witnessed). The witness bird had previously encountered one of the birds in the interaction it witnessed, but had never seen the other before. The witness bird was then allowed to interact with the bird from the other group. Analysis of the social interaction that occurred between the two birds on their first encounter showed that in those cases where the witness bird had sufficient information to reliably deduce its relative social status, it more often behaved in a way consistent with that social position, than an experimental control that had not witnessed any interaction.

The results from a related study using Western Scrub jays (a less social species, closely related to Pinyon jays) showed less evidence for the ability to make use of relational information. Those animals that live together in social groups are likely to have various kinds of relational information available to them. The benefits of being able to make use of this information appears to have resulted in at least some social species developing the cognitive abilities needed to process and make use of this information.

Relational Reasoning in Humans

If some animal brains (that don't have what are considered higher level cognitive reasoning abilities) have developed a mechanism to combine relational information to create *new* information, it is possible that humans also possess a similar mechanism (this is not to say that they don't have any other cognitive systems that are capable of performing the same task). A possible consequence of having such a special purpose reasoning mechanism is that it may not handle all relational expressions in the same way (i.e., it is likely to be optimised for handling those situations that commonly occur in its owner's everyday life). Some of the studies of human linear reasoning have found that subjects are slower and make more errors when the operands in a sequence of relational expressions occur in certain orders.

One study [2] used a task that was based on what is known as social reasoning (using the relations *better* and *worse*). Subjects were shown two premises, involving three people, and a possible conclusion (e.g., Is *Mantle worse than Moskowitz?*). They had 10 seconds to answer *yes*, *no*, or *don't know*. All four possible combinations of conclusions were used.

	Premises	% Correct Responses
1	A is better than B, B is better than C	60.5
2	B is better than C, A is better than B	52.8
3	B is worse than A, C is worse than B	50
4	C is worse than B, B is worse than A	42.5
5	A is better than B, C is worse than B	61.8
6	C is worse than B, A is better than B	57
7	B is worse than A, B is better than C	41.5
8	B is better than C, B is worse than A	38.3

Table 2 – Eight sets of premises describing the same relative ordering between A, B, and C (people's names were used in the study) in different ways, followed by the percentage of subjects giving the correct answer. Adapted from De Soto, London, and Handel [2].

Based on the results (see Table 2) the researchers made two observations (which they called *paralogical principles*; cases 5 and 6 possess both, while cases 7 and 8 possess neither):

1 **People learn orderings better in one direction than another.** In this study people gave more correct answers when the direction was *better*-to-*worse* (case 1), than mixed direction (case 2, 3), and were least correct in the direction *worse*-to-*better* (case 4). This suggests that use of the word *better* should be preferred over worse (the British National Corpus [3] lists *better* as appearing 143 times per million words, while *worse* appears under 10 times per million words and is not listed in the top 124,000 most used words).
2 **People *end-anchor* orderings.** That is, they focus on the two extremes of the ordering. In this study people gave more correct answers when the premises stated an end term (*better* or *worse*) followed by the middle term, than a middle term followed by an end term.

A related experiment in the same study used the relations *to-the-left* and *to-the-right*, and *above* and *below*. The *above*/*below* results were very similar to those for *better*/*worse*. The *left-right* results showed that subjects performed better with a *left*-to-*right* ordering than a *right*-to-*left* ordering.

Since this original study additional factors have been discovered and a number of models have been proposed to explain the strategies used by people in solving linear reasoning problems, including:

- **The spatial model** [2][4], in which people integrate information from each premise into a spatial array representing all known relationships.
- **The linguistic model** [5], in which people represent each premise using linguistic propositions (the individual premises are not integrated).
- **The algorithmic model** [6], in which people apply some algorithm to the structure of the linguistic representation of the premises. For instance, given "*Reg is taller than Jason; Keith is shorter than Jason*" and the question "*Who is the shortest?*", a so called *elimination strategy* was used by some subjects in the study. (The answer for the first premise is Jason, which eliminates Reg; the answer to the second premise is Keith which eliminates Jason, so Keith is the answer).
- **The mixed model** [7], in which the information in the premise is first decoded into a linguistic form and then encoded into a spatial form.

The strategy used to solve a given problem has been found to vary between people. A study by Sternberg and Weil [8] found a significant interaction between a subject's aptitude (as measured by verbal and spatial ability tests) and the strategy they used to solve linear reasoning problems. However, a person having high spatial ability, for instance, does not necessarily use a spatial strategy. A study by Roberts, Gilmore, and Wood [9] asked subjects to solve what appeared to be a spatial problem (requiring the use of a very inefficient spatial strategy to solve). Subjects with high spatial ability used non-spatial strategies, while those with low spatial ability used a spatial strategy. The conclusion made was that those with high spatial ability were able to see that the spatial strategy was inefficient to select as alternative strategy, while those with less spatial ability were unable to perform this evaluation.

If the evaluation of relational expressions in source code is performed using a cognitive mechanism that has been optimised for certain kinds of operations, then it is possible that developers' performance will be worse for some forms of expressions (e.g., the rate of making mistakes will be greater). The form of the `if` statements used in this study was designed to look for differences in subject performance that depended on the form of the relational expressions appearing in the control expressions.

Subject Motivation

When reading source code developers are aware that some of the information they see only needs to be remembered for a short period of time, while other information needs to be remembered over a longer period. For instance, when deducing the effect of calling a given function the names of identifiers declared locally within it only have significance within that function and there is unlikely to be any need to recall information about them in other contexts. Each of the problems seen by subjects in this study could be treated in the same way as an individual function definition (i.e., it is necessary to remember particular identifiers and the values they represent, once a problem has been answered there is no longer any need to remember this information).

Subjects can approach the demands of answering the problems this study presents them in a number of ways, including the following:

- seeing it as a challenge to accurately recall the assignment information (i.e., minimizing *would refer back* answers),
- recognizing that *would refer back* is always an option, but that it is more important to correctly answer the `if` statement question,
- making no conscious decision about how to approach the answering of problems.

Experience shows that many developers are competitive and that accurately recalling the assignment information, after solving the `if` statement problem, would be seen as the ideal performance to aim for. The experimental format did not allow for easy debriefing of subjects after they had answered the questions, and none was performed.

The only applicable instruction given to subjects was: "*Read the variables and the values assigned to them as you might when carefully reading lines of code in a function definition.*"

Results

The raw results for each subject are available on the study's web page [10].

`if` Statement/Assignment Recall Interaction

Answering the `if` statement portion of the problem requires time (information held in short term memory decays over time and unless it is regularly refreshed it will soon be lost) and use of short term memory resources. If subjects require more time or use more short term memory resources to answer some forms of relational expression problem, then performance in recalling assignment information is likely to be poorer after comprehending expressions having the *more complicated* form. The results (Figure 1) suggest that such a correlation may exist, at least for the first eight answers.

However, the difference in performance characteristics between the first eight answers and the ninth and subsequent problems may have been caused by subjects learning and making use of patterns in the assignment recall questions (which could reduce the need for short term memory resources). Alternatively some information occurred sufficiently often (e.g., the same identifier) that it was stored in a longer term memory subsystem, where it was not so susceptible to interference from the `if` statement problem.

`if` Statement Performance

This study differed from others on the topic of reasoning in a number of ways, including:

1 Researchers of human reasoning are usually attempting to understand the mechanisms underlying human cognition. For this reason they use subjects who have little or no experience in using formal mathematical logic. This study was interested in the performance of subjects in evaluating particular kinds of logical expressions and subjects were chosen because they had significant amounts of experience in evaluating the kinds of logical expressions that occur in source code.

2 The problems used in studies by researchers investigating the mechanisms of human cognition are usually expressed in forms that

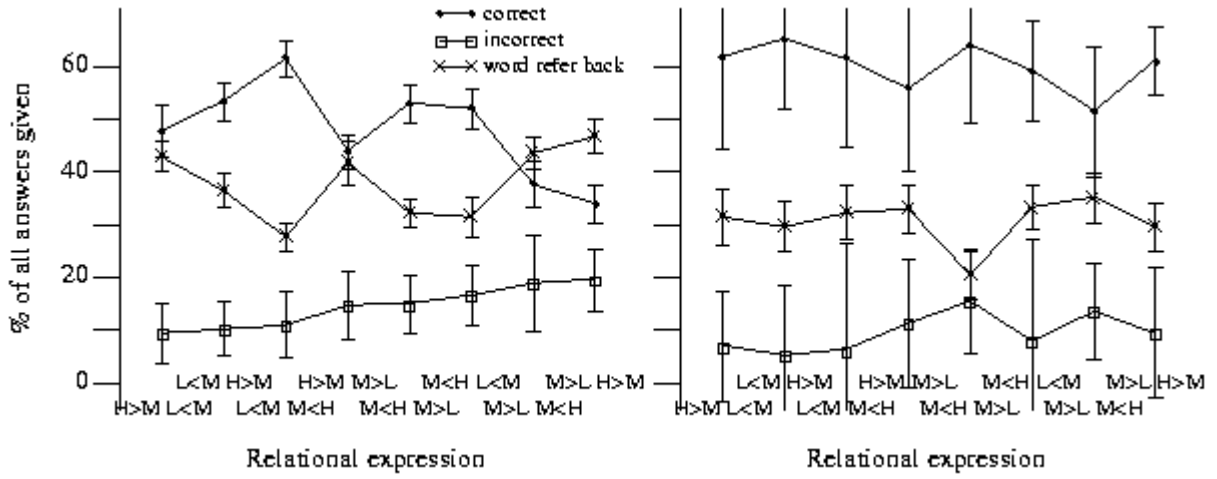


Figure 1 – The percentage of *would refer back*, correct and incorrect answers for each kind of relational expression. The left graph is based on answers to the first eight problems, while the right graph is based on the answers from the ninth and subsequent problem answers. Variation in subject performance is denoted by the error bars, which encompass one standard deviation. The ordering of relational expressions along the x-axis is sorted on the percentage of incorrect answers to the assignment problem, for the first eight `if` statement problems. H denotes high, M denotes middle, and L denotes low. So "H > M M > L" denotes "high greater than middle and middle greater than low".

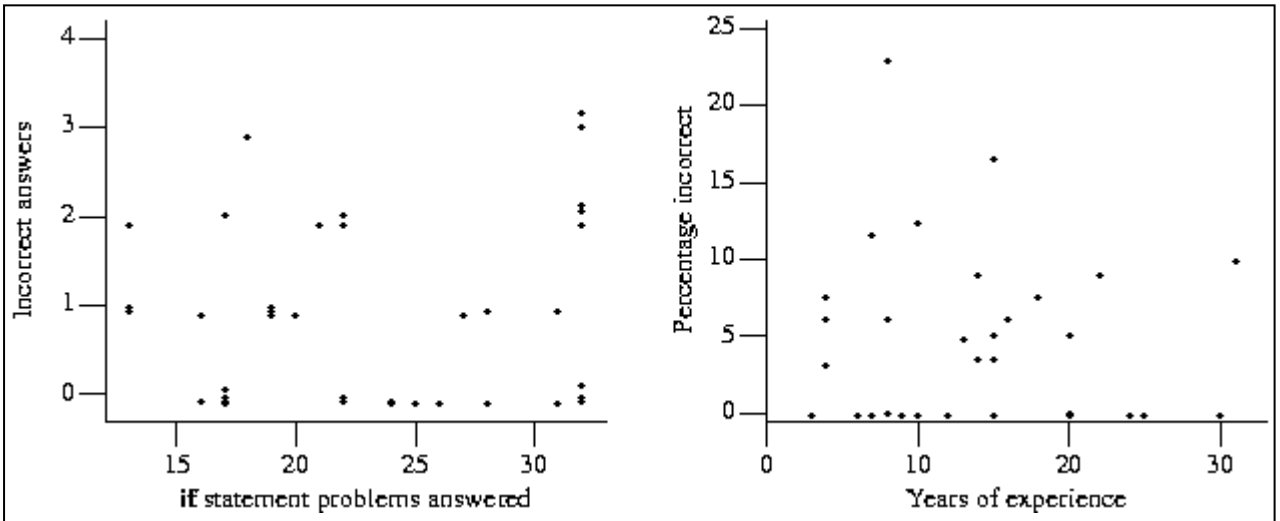


Figure 2 – The left graph plots the number of problems answered by each subject against the number of incorrect answers they gave. The bullets are offset from the y-axis to try to show those cases where more than one subject had the same problems answered/incorrect answers pair. The right graph plots the number of years of subject experience against the percentage of incorrect answers they gave.

occur in everyday life, i.e., they are natural language descriptions of everyday situations (e.g., "*If Jim deposits 50p, he gets a canned drink.*"). One of the complications caused by expressing problems in this form is that the words and phrases used are often open to multiple interpretations. It is also possible that subjects will base their answer on expectations they have about how the *real world* operates [11] .

3 In this study no limits were placed on subjects (De Soto et al. [2] required that an answer be given within 10 seconds), the mode of presentation mimicked that encountered in program comprehension (in the Huttenlocher [4] study subjects heard a tape recoding of the problem)

A total of 844 `if` statement problems were answered. There were 40 (4.7% of all answers) incorrect answers, an average number of incorrect answers per subject of 1. However, the incorrect answers were not evenly distributed across subjects. The number of incorrect answers did not appear to depend on the number of problems answered (Figure 2). While performance on reasoning tasks has been found to decrease with age [12] , years of experience (which is likely to be highly correlated with age) does not appear to have been a factor affecting the number of incorrect answers given to `if` statement problems.

Two of the reasons why subject performance could differ across different forms of relational expressions are:

1 Subjects may have a cognitive *relational deduction* mechanism (this may be actual hardware, i.e., a cluster of brain cells, or software, i.e., a neural network whose weights have been tuned through experience) that is optimised for handling problems (i.e., those that commonly occur in everyday life) that are expressed in a particular form.

2 The amount of cognitive resources required to solve a relational expression may depend on the form in which the expression is presented (this difference might simply be a consequence of how the human cognitive subsystem handles relational reasoning).

The paper and pencil format of the experiment meant that it was not feasible to obtain information on the amount of time taken to answer each problem.

Although subjects were told: "*Treat the paper as if it were a screen, i.e., it cannot be written on.*", there was nothing to prevent them using any paper that they happened to have on them as a temporary work area. Several subjects did write notes on the paper next to `if` statement problems (in one case for all the answered problems) and the answers to these problems were not counted. Except for the one case the number of such answers was very small (in the one case the subject was not included in the subject count).

The error rates reported by other studies (where subjects read a problem typed on a card) were: De Soto et al [2] 39.2 – 61.7%, Clark [5] 6%, Potts [13] 5%, Mayer [14] 4 – 36%, Quinton et al[6] not given, Sternberg et al[8] 1.7 – 3.5%. A study where subjects heard a tape recording of the problem[4] reported an error rate of 8 – 19%.

In order to look for patterns in the errors made by subjects it is necessary to have a statistically significant sample of the errors made by them. Unfortunately, there were not enough incorrect answers to the `if` statement problem (Table 3) to enable any statistically significant analysis to be performed.

Possible techniques for producing a greater number of incorrect answers include: running the experiment for a longer period of time (it seems reasonable to assume that the number of errors will increase as the number questions answered increases), or making the problem more difficult (e.g., using longer sounding identifiers).

General Conclusions

It was hoped that the results of this experiment would provide some insight into subjects' performance in handling short sequences of assignment and `if` statements. If the results of this experiment followed the pattern of behaviour seen in other (non-software related) experiments, it would be possible to claim that the models of human cognition created to explain that behaviour were also applicable here. The following summarises the conclusions:

Assignment information held in working memory. While there was some correlation between the duration of the spoken form of the identifiers appearing in assignment statements and subject performance, the content of long term memory also seems to play a significant role.

relational form	correct (first 8)	correct (9th and subsequent)	incorrect (first 8)	incorrect (9th and subsequent)	incorrect (total)
H > M M > L	34	80	0 (0.0%)	2 (2.5%)	2 (1.8%)
L < M H > M	38	66	0 (0.0%)	3 (4.5%)	3 (2.9%)
L < M M < H	37	64	1 (2.7%)	3 (4.7%)	4 (4.0%)
M < H M > L	40	69	3 (7.5%)	2 (2.9%)	5 (4.6%)
H > M L < M	40	64	4 (10.0%)	2 (3.1%)	6 (5.8%)
M > L M < H	28	73	4 (14.3%)	2 (2.7%)	6 (5.9%)
M < H L < M	41	71	2 (4.9%)	5 (7.0%	7 (6.2%)
M > L H > M	39	60	1 (2.6%)	6 (10.0%)	7 (7.1%)
Totals	297	547	15 (5.0%)	25 (4.6%)	40 (4.7%)

Table 3 – Errors. Number of correct and incorrect responses for the first eight and ninth and subsequent answers (parenthesized value is percentage of incorrect responses). H denotes high, M denotes middle, and L denotes low. So H > M M > L denotes "high greater than middle and middle greater than low".

Performance differences in evaluating conditional expressions. The form of relational expression had some impact on assignment recall performance (figure 1). However, the operand orderings giving the *best performance* (i.e., lowest number of errors made when recalling assignment information) were not the same as those for which subject *performed best* (i.e., lowest number of incorrect answers to logic problem) in other studies [2][4][5][6][7]. There was insufficient error data (Figure 2) for any reliable statistical analysis of subject `if` statement evaluation performance to be carried out.

Where Next?

While developers are often exhorted to think about the *meaningfulness* of identifiers, when creating new ones, the usability of identifiers within expressions and statements is rarely considered (apart, that is, from typing effort). More experiments need to be performed before it is

[concluded at foot of next page]

An Introduction to Programming with GTK+ and Glade in ISO C and ISO C++ – Part 4

Roger Leigh <rleigh@debian.org>

GTK+ and C++

In the previous article, it was shown that Glade and GObject could make programs much simpler, and hence increase their long-term maintainability. However, some problems remain:

- Much type checking is done at run-time. This might mean errors only show up when the code is in production use.
- Although object-oriented, using objects in C is a bit clunky. In addition, it is very difficult (although not impossible) to derive new widgets from existing ones using GObject, or override a class method or signal. Most programmers do not bother, or just use "compound widgets", which are just a container containing more widgets.
- Signal handlers are not type safe. This could result in undefined behaviour, or a crash, if a signal handler does not have a signature compatible with the signal it is connected to.
- Signal handlers are functions, and there is often a need to resort to using global variables and casting structures to type `gpointer` to pass complex information to a callback though its data argument. If Glade or GObject are used, this can be avoided, however.

gtkmm offers solutions to most of these problems. Firstly, all of the GTK+ objects are available as native C++ classes. The object accessor functions are now normal C++ *class methods*, which prevents some of the abuse of objects that could be accomplished in C. The advantage is less typing, and there is no need to manually cast between an object's types to use the methods for different classes in the inheritance hierarchy.

The gtkmm classes may be used just like any other C++ class, and this includes deriving new objects from them through inheritance. This also enables all the type checking to be performed by the compiler, which results in more robust code, since object type checking is not deferred until run-time.

Signal handling is also more reliable. gtkmm uses the `libsigc++` library, which provides a templated signal/slot mechanism for type-safe signal handling. The slot objects allow signal handlers with a different signature than the signal requires to be bound, which gives greater flexibility than the C signals allow. Perhaps the most notable feature is that signal handlers may be class methods, which are recommended over global functions. This results in further encapsulation of complexity, and allows the signal handlers to access the member data of their class. Unlike the Qt library, gtkmm does not require any source preprocessing, allowing plain ISO C++ to be used without extensions.

`libglademm` is a C++ wrapper around `libglade`, and may be used to dynamically load user interfaces as in the previous section. It provides similar functionality, the exception being that signals must be connected manually. This is because the `libsigc++` signals, connecting to the methods of individual objects, cannot be connected automatically.

`C++/glade/ogcalc`, shown in Figure 1, is identical to the previous examples, both in appearance and functionality. However, internally there are some major differences.

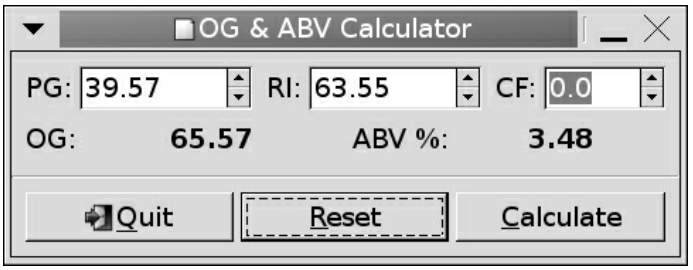


Figure 1: `C++/glade/ogcalc` in action.

[continued from previous page]

possible to reliably draw any firm conclusions about the consequences of using different kinds of identifier spellings in assignment statements and on developer performance during source code comprehension. Other experiments might use a greater number of different character sequences (e.g., abbreviations, or identifiers containing two known words), randomise the order in which identifiers appear in the table of assignment answers, or use more commonly occurring character sequences. Other experiments might also use different filler tasks.

Source code comprehension involves problem solving and developers are likely to use a variety of strategies to solve the problems that arise. The strategies used by developers can affect even such apparently simple tasks as remembering information about assignment statements. For instance, while some developers may choose to remember information about the identifiers appearing in an assignment using an encoding that involves their spoken form, other developers may use a different encoding (e.g., an abbreviated form of the identifier such as its first letter, or the encoding of the semantics that the identifier represents). Any study of developer cognitive performance needs to ensure that the subjects taking part in an experiment are only using their cognitive resources in a way has been anticipated by the experimenter (even simple tasks such as counting have been found to require cognitive resources [15]).

The problems used in this study could be answered by subjects having insignificant amounts of experience in software development (e.g., undergraduate computer science students). It would be interesting to compare the performance of inexperienced subjects against that of subjects having a significant amount of experience. However, care needs to be taken when using inexperienced subjects to take into account the possibility of performance improvement through learning of the underlying coding problem itself.

Derek M Jones

References

[1] Pazymino, Bond, Kamil & Balda, "Pinyon jays use transitive inference to predict social dominance", *Nature*, 430:778-781, Aug. 2004

[2] De Soto, London & Handel, "Social reasoning and spatial paralogic", *Journal of Personality and Social Psychology*, 2(4):513-521, 1965

[3] Leech, Rayson & Wilson, *Word Frequencies in Written and Spoken English*, Pearson Education, 2001

[4] Huttenlocher, "Constructing spatial images: A strategy in reasoning", *Psychological Review*, 75(6):550-560, 1968

[5] Clark, "Linguistic processes in deductive reasoning", *Psychological Review*, 76(4):387-404, 1969

[6] Quinton & Fellows, "Perceptual strategies in the solving of three-term series problems", *British Journal of Psychology*, 66:69-78, 1975

[7] Sternberg, "Representation and process in linear syllogistic reasoning", *Journal of Experimental Psychology: General*, 109(2):119-159, 1980

[8] Sternberg & Weil, "An aptitude x strategy interaction in linear syllogistic reasoning", *Journal of Educational Psychology*, 72(2):226-239, 1980

[9] Roberts, Gilmore, & Wood, "Individual differences and strategy selection in reasoning", *British Journal of Psychology*, 88:473-492, 1997

[10] Jones, *Experimental data and scripts for short sequence of assignment statements study*, `www.knosof.co.uk/cbook/accu04.html`, 2004

[11] Evans, Barston & Pollard, "On the conflict between logic and belief in syllogistic reasoning", *Memory & Cognition*, 11(3):295-306, 1983

[12] Gilinsky & Judd, "Working memory and bias in reasoning across the life span", *Psychology and Ageing*, 9(3):356-371, 1994

[13] Potts, "Storing and retrieving information about ordering relationships", *Journal of Experimental Psychology*, 103(3):431-439, 1974

[14] Mayer, "Qualitatively different encoding strategies for linear reasoning premises: Evidence for single association and distance theories", *Journal of Experimental Psychology: Human Learning and Memory*, 5(1):1-10, 1979

[15] Camos & Barrouillet, "Adult counting is resource demanding", *British Journal of Psychology*, 95:19-30, 2004

Further reading

For a readable introduction to human reasoning see "*Reasoning and thinking*" by Ken Manktelow. "*The Cognitive Animal*" edited by M. Bekoff, C. Allen, and G. M. Burghardt contains 57 short, wide ranging, essays (of varying quality) on animal cognition.

Acknowledgments

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.

Firstly, the `main()` function no longer knows anything about the user interface. It merely instantiates an instance of the `ogcalc` class, similar to `C/gobject/ogcalc`.

The `ogcalc` class is derived from the `Gtk::Window` class, and so contains all of the functionality of a `Gtk::Window`, plus its own additional functions and data. `ogcalc` contains methods called `on_button_clicked_calculate()` and `on_button_clicked_reset()` These are the equivalents of the functions `on_button_clicked_calculate()` and `on_button_clicked_reset()` used in the previous examples. Because these functions are class methods, they have access to the class member data, and as a result are somewhat simpler than previously.

Two versions are provided, one using the basic C++ classes and methods to construct the interface, the other using `libglademm` to load and construct the interface as for the previous examples using Glade. Only the latter is discussed here. There are a great many similarities between the C and C++ versions not using Glade, and the C Gobject version and the C++ Glade version. It is left as an exercise to the reader to compare and contrast them.

Code Listings

```
// C++/glade/ogcalc.h
#include <gtkmm.h>
#include <libglademm.h>
class ogcalc : public Gtk::Window {
public:
  ogcalc();
  virtual ~ogcalc();
protected:
  // Calculation signal handler.
  virtual void on_button_clicked_calculate();
  // Reset signal handler.
  virtual void on_button_clicked_reset();
  // The widgets that are manipulated.
  Gtk::SpinButton* pg_entry;
  Gtk::SpinButton* ri_entry;
  Gtk::SpinButton* cf_entry;
  Gtk::Label* og_result;
  Gtk::Label* abv_result;
  Gtk::Button* quit_button;
  Gtk::Button* reset_button;
  Gtk::Button* calculate_button;
  // Glade interface description.
  Glib::RefPtr<Gnome::Glade::Xml> xml_interface;
};

// C++/glade/ogcalc.cc
#include <iomanip>
#include <sstream>
#include <sigc++/retype_return.h>
#include "ogcalc.h"
ogcalc::ogcalc() {
  // Set the window title.
  set_title("OG & ABV Calculator");
  // Don't permit resizing.
  set_resizable(false);
  // Get the Glade UI and add it to this window.
  xml_interface = Gnome::Glade::Xml::create(
                "ogcalc.glade", "ogcalc_main_vbox");
  Gtk::VBox *main_vbox;
  xml_interface->get_widget("ogcalc_main_vbox",
                            main_vbox);
  add(*main_vbox);
  // Pull all of the widgets from the Glade interface
  xml_interface->get_widget("pg_entry", pg_entry);
  xml_interface->get_widget("ri_entry", ri_entry);
  xml_interface->get_widget("cf_entry", cf_entry);
  xml_interface->get_widget("og_result",
                            og_result);
  xml_interface->get_widget("abv_result",
                            abv_result);
  xml_interface->get_widget("quit_button",
                            quit_button);
  xml_interface->get_widget("reset_button",
                            reset_button);
  xml_interface->get_widget("calculate_button",
                            calculate_button);
  // Set up signal handers for buttons.
  quit_button->signal_clicked().connect(
        SigC::slot(*this, &ogcalc::hide));
  reset_button->signal_clicked().connect(
        SigC::slot(*this,
            &ogcalc::on_button_clicked_reset));
  reset_button->signal_clicked().connect(
        SigC::slot(*pg_entry,
            &Gtk::Widget::grab_focus));
  calculate_button->signal_clicked().connect(
        SigC::slot(*this,
            &ogcalc::on_button_clicked_calculate));
  calculate_button->signal_clicked().connect(
        SigC::slot(*reset_button,
            &Gtk::Widget::grab_focus));
  // Set up signal handlers for numeric entries.
  pg_entry->signal_activate().connect(
        SigC::slot(*ri_entry,
            &Gtk::Widget::grab_focus));
  ri_entry->signal_activate().connect(
        SigC::slot(*cf_entry,
            &Gtk::Widget::grab_focus));
  cf_entry->signal_activate().connect(
        SigC::hide_return(SigC::slot(*this,
            &Gtk::Window::activate_default)));
  // Ensure calculate is the default.  The Glade
  // default was lost since it was not packed in
  // a window when set.
  calculate_button->grab_default();
}

ogcalc::~ogcalc() {}

void ogcalc::on_button_clicked_calculate() {
  // PG, RI, and CF values.
  double pg = pg_entry->get_value();
  double ri = ri_entry->get_value();
  double cf = cf_entry->get_value();
  // Calculate OG.
  double og = (ri*2.597)  -(pg*1.644) - 34.4165 + cf;
  // Calculate ABV.
  double abv;
  if (og < 60)
    abv = (og - pg) * 0.130;
  else
    abv = (og - pg) * 0.134;
  std::ostringstream output;
  // Use the user's locale for this stream.
  output.imbue(std::locale(""));
  output << "<b>" << std::fixed
         << std::setprecision(2)
         << og << "</b>";
  og_result->set_markup(
              Glib::locale_to_utf8(output.str()));
  output.str("");
  output << "<b>" << std::fixed
         << std::setprecision(2)
         << abv << "</b>";
  abv_result->set_markup(
              Glib::locale_to_utf8(output.str()));
}

void ogcalc::on_button_clicked_reset() {
  pg_entry->set_value(0.0);
  ri_entry->set_value(0.0);
  cf_entry->set_value(0.0);
  og_result->set_text("");
  abv_result->set_text("");
}
```

```
// C++/glade/ogcalc-main.cc
#include <gtk/gtk.h>
#include <glade/glade.h>
#include "ogcalc.h"

// This main function merely instantiates the ogcalc
// class and displays it.
int main (int argc, char *argv[]) {
  Gtk::Main kit(argc, argv); // Initialise GTK+.
  ogcalc window;   // Create an ogcalc object.
  kit.run(window);
           // Show window; return when it's closed.
  return 0;
}
```

To build the source, do the following:

```
cd C++/glade
c++ 'pkg-config –cflags libglademm-2.0' -c ogcalc.cc
c++ 'pkg-config –cflags libglademm-2.0'
                                    -c ogcalc-main.cc
c++ 'pkg-config –libs libglademm-2.0' -o ogcalc
                               ogcalc.o ogcalc-main.o
```

Similarly, for the plain C++ version, which is not discussed in the tutorial:

```
cd C++/plain
c++ 'pkg-config –cflags gtkmm-2.0' -c ogcalc.cc
c++ 'pkg-config –cflags gtkmm-2.0' -c ogcalc-main.cc
c++ 'pkg-config –libs gtkmm-2.0' -o ogcalc ogcalc.o
                               ogcalc-main.o
```

Analysis

ogcalc.h

The header file declares the `ogcalc` class.

```
class ogcalc : public Gtk::Window
```

`ogcalc` is derived from `Gtk::Window`

```
virtual void on_button_clicked_calculate();
virtual void on_button_clicked_reset();
```

`on_button_clicked_calculate()` and `on_button_clicked_reset()` are the signal handling functions, as previously. However, they are now class member functions, taking no arguments.

```
Gtk::SpinButton* pg_entry;
Glib::RefPtr<Gnome::Glade::Xml> xml_interface;
```

The class data members include pointers to the objects needed by the callbacks (which can access the class members like normal class member functions). Note that `Gtk::SpinButton` is a native C++ class. It also includes a pointer to the XML interface description. `Glib::RefPtr` is a templated, reference-counted, "smart pointer" class, which will take care of destroying the pointed-to object when `ogcalc` is destroyed.

ogcalc.cc

The constructor `ogcalc::ogcalc()` takes care of creating the interface when the class is instantiated.

```
set_title("OG & ABV Calculator");
set_resizable(false);
```

The above code uses member functions of the `Gtk::Window` class. The global functions `gtk_window_set_title()` and `gtk_window_set_resizable()` were used previously.

```
xml_interface = Gnome::Glade::Xml::create(
                "ogcalc.glade", "ogcalc_main_vbox");
Gtk::VBox *main_vbox;
xml_interface->get_widget("ogcalc_main_vbox",
                          main_vbox);
add(*main_vbox);
```

The Glade interface is loaded using `Gnome::Glade::Xml::create()`, in a similar manner to the `GObject` example, and then the main `VBox` is added to the `ogcalc` object.

```
xml_interface->get_widget("pg_entry", pg_entry);
```

Individual widgets may be obtained from the widget tree using the static member function `Gnome::Glade::Xml::get_widget()`.

Because gtkmm uses `libsigc++` for signal handling, which uses class member functions as signal handlers (normal functions may also be used, too), the signals cannot be connected automatically, as in the previous example.

```
quit_button->signal_clicked().connect(
              SigC::slot(*this, &ogcalc::hide));
```

This complex-looking code can be broken into several parts.

```
SigC::slot(*this, &ogcalc::hide)
```

creates a `SigC::slot` (function object) which points to the `ogcalc::hide()` member function of this object.

```
quit_button->signal_clicked()
```

returns a `Glib::SignalProxy0` object (a signal taking no arguments). The `connect()` method of the signal proxy is used to connect `ogcalc::hide()` to the "clicked" signal of the `Gtk::Button`.

```
calculate_button->signal_clicked().connect(
        SigC::slot(*this,
            &ogcalc::on_button_clicked_calculate));
calculate_button->signal_clicked().connect(
        SigC::slot(*reset_button,
            &Gtk::Widget::grab_focus));
```

Here two signal handlers are connected to the same signal. When the "Calculate" button is clicked, `ogcalc::on_button_clicked_calculate()` is called first, followed by `Gtk::Widget::grab_focus()`.

```
cf_entry->signal_activate().connect(
        SigC::hide_return(SigC::slot(*this,
            &Gtk::Window::activate_default)));
```

`SigC::hide_return` is a special `SigC::slot` used to mask the boolean value returned by `activate_default()`. The slot created is incompatible with with the slot type required by the signal, and this "glues" them together.

In the `ogcalc::on_button_clicked_calculate()` member function,

```
double pg = pg_entry->get_value();
```

the member function `Gtk::SpinButton::get_value()` was previously used as `gtk_spin_button_get_value()`.

```
std::ostringstream output;
output.imbue(std::locale(""));
output << "<b>" << std::fixed
       << std::setprecision(2)
       << og << "</b>";
og_result->set_markup(
              Glib::locale_to_utf8(output.str()));
```

This code sets the result field text, using an output `stringstream` and Pango markup.

In the `ogcalc::on_button_clicked_reset()` member function,

```
pg_entry->set_value(0.0);
og_result->set_text("");
pg_entry->grab_focus();
```

class member functions are used to reset and clear the widgets as in previous examples.

ogcalc-main.cc

This file contains a very simple `main()` function.

```
Gtk::Main kit(argc, argv); // Initialise GTK+.
ogcalc window;
kit.run(window);
```

A `Gtk::Main` object is created, and then an `ogcalc` class, `window`, is instantiated. Finally, the interface is run, using `kit.run()`. This function will return when `window` is hidden, and then the program will exit.

Conclusion

Which method of programming one chooses is dependent on many different factors, such as:

- The languages one is familiar with.
- The size and nature of the program to be written.
- The need for long-term maintainability.
- The need for code reuse.

For simple programs, such as `C/plain/ogcalc`, there is no problem with writing in plain C, but as programs become more complex, Glade can greatly ease the effort needed to develop and maintain the code. The code reduction and de-uglification achieved through conversion to `Glade/libglade` is beneficial even for small programs, however, so I would recommend that Glade be used for all but the most trivial code.

[concluded at foot of next page]

Reviews

Bookcase

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

Francis Writes

It is curious the way things go in cycles. We have had very few C and C++ books in the past few issues and now we are inundated with them. Unfortunately, there is the usual high number of poor books for novices. In addition, the books from one publisher seem expensive in the UK and astronomical in the US (even when they are listed). Asking a student to pay a large amount for a mediocre book seems unreasonable. I wonder if the high headline prices are a mechanism to appear to give big discounts to academic bulk purchasers.

The relative costs of books in the UK and the US are becoming completely silly. Actually, after the US dollar has remained at around 1.8 to the pound sterling for over a year, a ratio of 2:1 for books sourced in the UK isn't too bad for US purchasers but ratios of 5:4 for books sourced in the US seems quite unreasonable.

Another issue I noticed recently is the weight of books. One or two publishers are using very heavy paper, which may be seriously affecting transport costs. We now have a ratio of over 2:1 in weight for books with the same page count. Quite apart from anything else, the average person does not want to carry around a textbook that weighs over 2kg. Yes, I just checked by weighing a couple.

Francis

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

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

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

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

C & C++

Computer Science A Structured Approach Using C++ by Behrouz A. Forouzan & Richard F. Gilberg (0-534-37480-8), Thomson, 1020pp @ £36-99 (2.0)
reviewed by Frank Antonsen

The aim of this book is to teach C++ as a first language. Often C++ is considered too complicated for this and Java is used instead. But although C++ is a very powerful language that allows you to do all sorts of complicated things, you can also do quite simple things in a simple and consistent manner.

So let me start by saying what I like about this book. First, it begins by using streams and manipulators to provide a nice looking output in the very first chapter. All too often, this is moved to an advanced chapter near the end of the book. Secondly, it shows you how to write your own I/O manipulator and handle binary files. There is an appendix on the STL, and they introduce templates relatively early on. Finally, each chapter concludes with a broader section on general software design techniques and methodology. However, alas, this also shows the limitations of the book. They cite very few books, which is not a problem, but all of them are from the seventies or eighties. Many things have happened since.

In fact, the authors spend a lot of time discussing structure diagrams and give much good advice on how to factor functions. However, they barely touch UML, and there is not a single word on when to factor classes. In other words, they teach is procedural programming, with just hint of object orientation. All the advice they give on writing functions remain valid of course, but it takes on quite a different meaning in the context of classes.

The first part of the book shows signs of being just a quick update of the original book (I don't know when that appeared). For instance, it warns against using the `bool` type, because not all compilers provide it. In 2004, if a compiler does not provide `bool`, do not use it! Fortunately, the authors ignore their own advice for most of the book so this is not a big problem. The same holds for their advice on where to define variables inside functions; the first chapters recommends putting all variable definitions at the top before any executable code, that is in the old-fashioned C-style. Again, the authors ignore their own advice in the rest of the book and comply with modern practice of defining a variable as close as possible to where it is used.

The book has 1000+ pages, so inevitably a number of typos must show up. There are a few places where the code is clearly wrong, but these are usually obvious and should not confuse the novice too much. What is more serious is the old-fashioned style. There is a long (actually quite good) discussion on pointers and the use of the `this` pointer, but no mention of `auto_ptr`. They use characters instead of standard strings until the end of the book, where a chapter on strings has been tagged on. Even though templates are introduced in chapter 13, they are not used in the remaining 4 chapters, not even in the final chapter where a linked list class is designed. Exceptions are only introduced at the very end; all previous code relies on the older style of having functions returning an invalid number or calling `exit` with an error-code.

[continued from previous page]

The C++ code using gtkmm is slightly more complex than the code using Glade. However, the benefits of type and signal safety, encapsulation of complexity and the ability to re-use code through the derivation of new widgets make gtkmm and `libglademm` an even better choice. Although it is possible to write perfectly good code in C, gtkmm gives the programmer security through compiler type checking that plain GTK+ cannot offer. In addition, improved code organisation is possible, because inheritance allows encapsulation.

GObject provides similar facilities to C++ in terms of providing classes, objects, inheritance, constructors and destructors etc., and is certainly very capable (it is, after all, the basis of the whole of GTK+!). The code using GObject is very similar to the corresponding C++ code in terms of its structure. However, C++ still provides facilities such as RAII (Resource Acquisition is Initialisation) and automatic destruction when an object goes out of scope that C cannot provide.

There is no "best solution" for everyone. Choose based on your own preferences and capabilities. In addition, Glade is not the solution for every problem. The author typically uses a mixture of custom widgets and Glade interfaces (and your custom widgets can contain Glade interfaces!). Really dynamic interfaces must be coded by hand, since Glade interfaces are not sufficiently flexible. Use what is best for each situation.

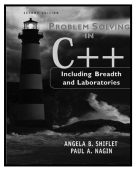
Roger Leigh

Further Reading

The GTK+ Tutorial, and the GTK+ documentation are highly recommended. These are available from `http://www.gtk.org/` The gtkmm documentation is available from `http://www.gtkmm.org` Unfortunately, some parts of these manuals are as yet incomplete. I hope that they will be fully documented in the future, since without good documentation, it will not be possible to write programs that take advantage of all the capabilities of GTK+ and gtkmm, without having to read the original source code. While there is nothing wrong with reading the source, having good documentation is essential for widespread adoption of GTK+.

Documentation and examples of GObject are scarce, but Mathieu Lacage has written an excellent tutorial which is available from `http://le-hacker.org/papers/gobject/`

It is really a pity. The authors do quite a good job at explaining things, but for a book published in 2004 it is far too old-fashioned. This becomes clearer as one reads the later chapters where most of what has happened to C++ since the late eighties has been quickly tagged on.



Problem Solving in C++ by Angela Shiflet & Paul Nagin (0-534-40005-1), Thomson, 1070pp @ £35-99 (no US price)
reviewed by Frank Antonsen

I have some problems with this book. On the one hand, it is written in an easy to follow language and contains valid points on how to structure programs. It also contains some interesting background information and exercises.

However, the book is more about C with I/O streams added than it is about C++. Even though classes are introduced early on (in chapter 3, on page 151), they are rarely used for anything but the most trivial encapsulation. I would be able to accept this in a beginner's book if there were not so many such oddities and omissions.

Let me give some examples.

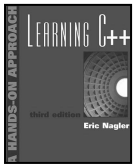
For the first 10 chapters, spanning two thirds of the book, the only character data type introduced is `char`. Then in chapter 11, the first 40-odd pages are used to discuss `char*`, before 8 pages are devoted to the standard `string` class. Why? The fundamental character data type is after all not the individual character, just like the basic numeric data type is not the digit. Ironically, the authors finish chapter 11 with saying that people should use `std::string` instead of the old C-strings!

Throughout the book the authors prefer to use the C-functions `exit` or even `assert` to terminate a program upon error. Why do they not use exceptions?

Chapter 10 introduces arrays, and finishes off by introducing the vector class from STL. They even give a (very short) introduction to templates, which they then never use. The final chapters of the book implement some basic data structures. This would be perfectly acceptable if it was a way of demonstrating how templates are used in STL by providing some simple implementation of the container. Instead they even fail to mention that the STL contains (far superior) implementations of these containers.

Furthermore, this is a book that goes to great lengths to teach safe programming methods. Why do they then almost invariably define global constants with `#define` instead of using `const int` or `string` etc, which would give them type safety as well?

In summary, the authors seem to possess only a rudimentary knowledge of modern C++. As I said above, this is really a book on C with C++'s iostream library added. Not recommended.



Learning C++ 3ed by Eric Nagler (0-534-38966-X), Thomson, 530pp @ £34-99 (no US price)
reviewed by Paul F. Johnson

I started reading this book and what initially struck me was how easy it was to read. It has plenty of good points (such as common problems and helpful hints) and the writing style is very much like Schildt, except without the number of mistakes.

I really have two problems with the book.

Firstly, the over use of acronyms. The author seems to like these and while they are initially explained, it becomes a pain trying to remember them. Sure we have the likes of FDDI and other such acronyms to recall, but the number of them in this book really will not help those wanting to learn.

Secondly, the use of Hungarian notation. I have never liked Hungarian notation and is widely discredited as a teaching method within education. Take for instance this variable name

```
char const *ptr Ptr;
```

What exactly is it? Okay, to the seasoned reader, we would recognize it as a `char const` pointer called `Ptr`, but put yourself in the position of a new learner. It is hard enough coming to terms with the use of `const` without finding names such as above.

Those aside, the author has obviously put a lot of thought into the layout and approach to the book and it shows – the coverage of the C++ style of casting comes very early and classes and encapsulation are handled in a simple to understand approach.

My other criticism is that there are a lot of "will be covered in chapter xxx" when a simpler approach would be to have (say) `try`/`catch` when discussing memory allocation – the code is there and a line saying what the `try`/`catch` mechanism does and then refers to a later chapter. It certainly does not detract overly much, but for a new learner, it will not help.

While not as comprehensive as a Schildt book (in terms of coverage rather than accuracy – this book has far fewer mistakes than the average Schildt book), it is a far better book for a new-comer to C++ than a lot of books on the market. Recommended with reservations.

Developing Series 60 Applications A Guide for Symbian OS by Leigh Edwards et al. (0-321-22722-0), Addison-Wesley*, 748pp @ £37-99 (1.32)
reviewed by David Caabeiro

For those waiting for a definitive reference on Symbian C++ development for Series 60, this book fulfils all expectations. Series 60 is currently the best selling mobile platform, being deployed on devices from manufacturers such as Nokia, Siemens, Samsung, etc. It is difficult to find a topic not covered by this book, and given the lack of documentation provided by the SDK, it becomes a must-have in your bookshelf.

The book could be split into three parts. The first part comprises basic stuff such as building and deployment process, Symbian fundamental APIs and application framework (comparable to the MVC pattern). It is fundamental to understand these chapters to understand the rest of the book.

The second part refers to UI gadgets, starting with an explanation of basic controls, event handling, menus, etc. Following chapters provide description of dialogs, lists, notes, editors and many other system widgets.

Lastly, more advanced stuff, such as communications programming (sockets, TCP/IP, IrDA, Bluetooth, HTTP, messaging and telephony), multimedia framework, system engines and views, and finally testing and debugging.

Of course, no book covers all possible topics. The information you will find on some chapters (communications is an example) is the essential you will need to get started. For other advanced topics, such as client-server architecture, multithreading, etc. you will need to look for other material.

One of the things I liked most of this book is the quantity and quality of examples (which are available online) which feature working applications, so they are ready to build and run on your emulator and smartphone.

If you are on your first steps with Series 60 development get this book, you will not be disappointed. As I read somewhere, it might well be considered the "Charles Petzold" for Series 60 platform development.

C++ Programming: From Problem Analysis to Program Design by D.S. Malik (0-619-16042-X), Thomson, 1304pp + CD @ £34-00 (2.23)
reviewed by James Roberts

When I chose this book from the 'to be reviewed' list, I was expecting a book aimed at a reader who had mastered the basics of C++ (perhaps from the same authors C++ primer book), and was interested in progressing further.

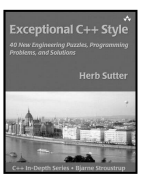
It turned out that this book is aimed at building up a student's knowledge of C++ from a start point of little or nothing.

The style is fairly wordy, and includes copious examples of completed code. Unfortunately, the author does not explain why design choices were taken, or what alternatives were not taken.

As a course book, perhaps it is reasonable to not include anything but the briefest descriptions of which compilers might be useful. However, for any other readers this is in my opinion rather important.

The main complaint I had with the book is the actual content. Why was there no mention of polymorphism (other than a passing definition)? Why were three chapters dedicated to the implementation of linked lists, queues and stacks, with no mention of the STL outside the appendices? A description of the concepts would have its place – but the full source code seems over the top.

There is apparently 'valuable testing software' included with the book. This seems to consist of a series of acrobat files mainly consisting of examination texts. I was unable to access the website, as I had no instructor id.



Exceptional C++ Style by Herb Sutter (0 201 76042 8), Addison-Wesley*, 325pp @ £30-99 (1.29)
reviewed by Pete Goodliffe

If you know Herb Sutter's writing then you will already be asking: is this **another** must have C++ book? Indeed it is. Herb has produced another exceptional (pun intended) tome. If you are a C++ programmer who is not familiar with Sutter's work then I suggest you get copies of Herb's previous books, work through them, and then get this one.

Sutter is a renowned C++ guru, chair of the ISO C++ standards committee, regular CUJ columnist, and conference speaker. He knows what he's talking about. As ever his latest book is well structured, readable, and authoritative.

It follows directly on from his two previous "Exceptional C++" books, and the story here is very much "business as usual". Presented in a question and answer format (which often works well, and sometimes seems very contrived), various individual topics are investigated in separate mini-articles. Some of the more thorny topics are split across several articles.

Sutter takes us on a journey through the latest wisdom on generic programming, exception safety, class design, resource management and optimisation. I was originally confused by the book's title "Exceptional C++ Style"; none of the items are really any more to do with programming style than his previous books.

However the last section, probably the best, does finally do some justice to the title. Sutter provides a number of case studies of Real World code, showing how to improve its coding style in light of modern C++ wisdom. This section alone will help less experienced C++ programmers to learn what industrial strength C++ coding is about.

The book is well cross-referenced (internally, with his earlier books, and with other major C++ books) and clearly laid out, with sound bite "guidelines" to distil the important information. It comes highly recommended for all practicing C++ programmers.

Due to lack of space not all book reviews could be printed in this issue. Reviews of the following books can be found on the website (`www.accu.org`*) and will be printed in the next issue if space permits.*

C++

Object-Oriented Programming: Using C++ for Engineering by Goran Svenk (0-7668-3894-3), Thomson, 506pp + CD @ £38-00 (1.92) reviewed by Mark Easterbrook

Ivor Horton's Beginning ANSI C++ 3ed by Ivor Horton (1-59059-227-1), Apress, 1090pp @ £37-50 (1.60) reviewed by Malcolm Pell

C++ Demystified by Jeff Kent (0-07-225370-3), McGraw Hill Osborne, 348pp @ £12-99 (1.54) reviewed by James Roberts

Programming in C, 3ed by Stephen G. Kochan (0-672-32666-3), Developer's Library*, 542pp @ £28-99 (1.38) reviewed by Giles Moran

C Programming for the Absolute Beginner by Michael Vine (1-931841-52-7), Premier Press, 240pp + CD @ £21-99 (1.36) reviewed by Thomas Padron-McCarthy

.NET Programming

ADO.NET in a Nutshell by B Hamilton & M MacDonald (0-596-00361-7), O'Reilly, 600pp + CD @ £31-95 (1.41) reviewed by Mick Spence

Programming .NET Components by Juval Lowy (0-596-00347-1), O'Reilly, 458pp @ £28-50 (1.40) reviewed by Paul Usowicz

Teach Yourself Visual Studio .NET 2003 by Jason Beres (0-672-32421-0), SAMS, 666pp @ £28-99 (1.38) reviewed by Griff Phillips

C# & Java

J2EE and XML Development by Kurt Gabrick & David Weiss (1 930110 30 8), Manning, 274pp @ £35-99 (1.11) reviewed by Alistair McDonald

JUnit Recipes by J.B.Rainsberger (1-932394-23-0), Manning, 720pp @$49.95 reviewed by Anthony Williams

Learn Java in a Weekend by Joseph P Russell (1-931841-60-8), Premier Press, 482pp @ £21-00 (1.43) reviewed by Paul F. Johnson

Professional Java Tools for Extreme Programming by Richard Hightower et al. (0-7645-5617-7), Wrox, 732pp @ £29-99 (1.50) reviewed by Jim Hague

Embedded and Real Time

Real-Time Java Platform Programming by Peter Dibble (0 13 028261 8), Prentice Hall, 332pp @ £39-99 (1.25) reviewed by Alan Barclay

Database Programming

Access VBA for the Absolute Beginner by Michael Vine (1-59200-039-8), Prima Tech, 328pp + CD @ £21-99 (1.36) reviewed by Richard Knight

Information Architecture with XML by Peter Brown (0-471-48679-5), Wiley, 324pp @ £27-50 (1.81) reviewed by Christopher Hill

Games Programming

Beginner's Guide to DarkBasic Game Programming by Jonathan S. Harbour (1-59200-009-6), Prima Tech, 711pp + CD @ £37-99 (1.58) reviewed by Mark Green

Methodologies & Testing

Agile Development in the Large by Jutta Eckstein (0-932633-57-9), Dorset House, 216pp @ $33.95 reviewed by Alan Griffiths

CMMI Distilled 2ed by Dennis M. Ahem et al (0-321-18613-3), Addison-Wesley, 310pp @ £22-99 (1.30) reviewed by Greg Billington

The OPEN Process Framework by Donald Firesmith & Brian Henderson-Sellers (0 201 67510 2), Addison-Wesley, 330pp @ £29-99 (1.50) reviewed by Matt Pape

Real World Software Configuration Management by Sean Kenefick (0-59059-065-1), Apress, 439pp @ £35-50 (1.41) reviewed by Derek Graham

Official Eclipse 3.0 FAQs by John Arthorne, Chris Laffra (0-321-26838-5), Addison-Wesley*, 385pp @ £26-99 (1.30) reviewed by Silas Brown

The Web & Networking

Pro Apache 3ed by Peter Wainwright (1-59059-3006), Apress, 880pp @ £31-50 (1.59) reviewed by Alan Barclay

The Definitive Guide to Linux Network Programming by Keir Davis et al. (1-59059-322-7), Apress, 375pp @ £31-50 (1.59) reviewed by Alyn Scott

General Programming

Unix & Shell Programming by B Forouzan & R Gilberg (0-534-39155-9), Thomson,pp @ £35-00 (no US price) reviewed by Paul F. Johnson

About Face 2.0; The Essentials of Interaction Design by Alan Cooper and Robert Reimann (0-7645-2641-3), Wiley, 540pp @ £24-50 (1.43) reviewed by Christopher Hill

Refactoring to Patterns by Joshua Kerievsky (0-321-21335-1), Addison-Wesley*, 367pp @ £37-99 (1.32) reviewed by Anthony Williams

Succeeding With Open Source by Bernard Golden (0-321-26853-9), Addison-Wesley*, 242pp @ £30-99 (1.29) reviewed by Mike Pentney

Holub on Pattems by Allen Holub (1-59059-388-X), Apress*, 412pp @ £31-50 (1.59) reviewed by Alan Lenton

VB for the Absolute Beginner by Michael Vine (0-7615-3553-5), Prima Tech, 342pp + CD @ £21-99 (1.36) reviewed by Richard Knight

Non-Programming

Linux in Easy Steps by Mike McGrath (1-84078-275-7) Computer Steps, @ £10.99 reviewed by Paul F. Johnson

Fearless Change by Mary Lynn Manns, PhD & Linda Rising, PhD (0-201-74157-1), Addison-Wesley, 273pp @ £22-99 (1.09) reviewed by Francis Glassborow