Contents

Reports & Opinions

Reports

Dialogue

Features

Python

Reviews

Copy Dates

C Vu 15.4: July 7th

C Vu 15.5: September 7th

!!!! PRIZES for journal articles - see page 19 for details !!!!

Contact Information:

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

Advertising: Pete Goodliffe
Chris Lowe
ads@accu.org

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

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

Secretary: Alan Bellingham
01763 248259
secretary@accu.org

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

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

Membership fees and how to join:

Basic (C Vu only): £15

Full (C Vu and Overload): £25

Corporate: £80

Students: half normal rate

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

There are 6 journals of each type produced every year.

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

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

Any questions - just email membership@accu.org

Reports & Opinions

Editorial

This month's C Vu welcomes Ewan Milne as our new chair. Alan Griffiths has served in this capacity for a number of years, and has done fine work, overseeing considerable progress in the life of ACCU. So, thanks to Alan for getting ACCU to where it is today, and good luck to Ewan with taking it forward from here. Ewan's first "From the Chair" piece can be found after this editorial.

An organisation such as ACCU (with its new, freestanding name acknowledging the evident fact that ACCU is about much more than just C and C++) needs to combine progress with continuity. C Vu, too, works best when it combines contributions from new authors with material from regular contributors. Without his permission, I would like to thank Pete Goodliffe for being one of those regular contributors, with his column on "Professionalism in Programming" being right at the heart of the spirit of this journal.

Pete's "Professionalism" column in this issue mentions Martin Fowler's "Refactoring" book. This also received my recommendation, mirrored in Paul Grenyer's article published in a previous C Vu. The same books come up time and again, for good reason, and I am unashamedly using this opportunity to recommend it again. Some will dismiss this book for writing about simple things, but its approach of breaking a change to a program down into the smallest possible almost mechanical steps is a powerful way to allow us to make code better. Many books are written on how to design code from scratch, but most of the time most of us are changing existing code - indeed, some people with sympathies for the Extreme Programming school of thought claim "you're always doing maintenance". For that reason, it is often easier to apply the advice found in texts which discuss how to change our code and designs rather than how to come up with designs on the well understood but relatively rare occasions when we are given a clean slate. "Refactoring" is one book that addresses this niche. Please do tell me about others you have found - I am always happy to receive mail to `cvu@accu.org`.

Pete Goodliffe also talks about the role of simplicity in software design, and its limitations. Simplicity is so important that it is tempting to advocate making it a cornerstone of any course on software construction — and yet so elusive that we really don't know how to teach the idea. What counts as the simplest code to an experienced developer might be intimidating to a fledgling software engineer fresh out of college. It takes time measured in years to become truly comfortable with all of the idiomatic uses of a complicated programming language like C or C++. (Fans of simpler languages such as Java or Python can instead note that it takes years to become familiar with idiomatic uses of the libraries from those languages.) There is a balance to be struck between writing "neat" code (which might require less experienced developers to learn a new idea) and writing "excessively clever" code that can confuse or even mislead other programmers. If too many of your colleagues think that your code is "too cute" or "complicated", remember that the most important function of code is to be comprehensible to other developers. Most of the work will go in maintenance after all, and code that meets the original specification but cannot be changed as requirements evolve will have to be rewritten. Simple code is more flexible as well as more readable. Clever code should be reserved for (i) situations in which advanced techniques are necessarily involved and (ii) developers who are insecure about their real skill, and need to seek job security by writing code nobody else can maintain.

For those who have read Pete's articles in the past, and nodded sagely at the hard-earned experience he distils, I'd recommend digging up the old ones and re-reading them from your current perspective. Think about how your current role can use the same knowledge from different angles. If you're finding that you take on an increasing amount of leadership or management responsibility, take note of the comments Pete has made on how developers interact with team leads and managers - and make sure you are not turning into the pointy-haired manager you once mocked. (And the other way too: if your "boss" is a battle-hardened software developer, he or she will remember dealing with managers who "just didn't get it", and will probably appreciate a friendly word if he/she starts to forget where he/she came from.

An aside: how long is it before English adopts a gender-neutral form of he/she? I am no fan of excessive political correctness — those who know me will attest to that — but writing "he/she" tires quickly for writer and reader alike, attempting to use each gender equally often in an arbitrary way is a hack, and writing "the masculine pronoun he will mean either he or she" is an unbearable contradiction. Those wanting an interesting diversion from more common topics might like to investigate online, maybe starting with a look at the gender-neutral pronoun FAQ at `http://www.aetherlumina.com/gnp/faq.html`.

And now, if you will permit me, I'm going to talk a little about programming languages.

Recently I have been learning about C#. Previously I had understood it as a rip-off of Java, and read little more than that. Increasingly, though, I have been asked my opinions of C# from a technical point of view, and so it has been necessary for me to develop some more refined impressions. Previous conversations convinced me that C# might be a better programming language - considered as a programming language in isolation from its larger environment and commercial considerations - than Java, whilst still being similar in concept. So, I have dug a little deeper, and written some code, using the DotGNU implementation of the C# compiler and its associated runtime running on Mac OS X. (More of that another time.)

For those who are at least somewhat familiar with Java but haven't looked at C#, I would recommend taking the time to learn a little about the new kid on the block.

One very small point may be of interest. In C++, we learn that `+=` is a more primitive operation than binary `+`, and so idiomatic code tends to implement the binary addition operation in terms of `operator+=`. In C#, `+=` is a pure abbreviation, and hence if an `operator+` is defined for a type (recall: C# has operator overloading, whereas Java does not except for its `String` class), `operator+=` will come "for free". For a class type, it just won't be as efficient as an `operator+=` in C++, because it needs to create a new object. For the C# gurus out there, I'd be interested to hear if a smart C# implementation could avoid the creation of a new object if certain constraints were satisfied - such as the lhs being the only reference to its referent.

The appearance of a competitor in C# appears to have given new speed to the evolution of Sun Microsystems' Java programming language. Features slated for Java 1.5 include templates (though in a less powerful form that the template system of C++), enums (Java dropped these from C, but C# reintroduced them and now Java seems set to do the same), a foreach construct, and automatic boxing and unboxing (similar to that seen in C#).

> "The new language features all have one thing in common: they take some common idiom and provide linguistic support for it. In other words, they shift the responsibility for writing the boilerplate code from the programmer to the compiler."- Joshua Bloch, senior staff engineer, Sun Microsystems

C#, too, is acquiring templates. It is interesting to see in the evolution of the languages touted by some as C++-killers that contact with real programming problems causes them to trade off some of their initial simplicity for power. C++ is indeed far from perfect - but I'm reminded of an overused quote about people unable to remember the past. ("Condemned to misquote it," or something like that.)

All of this is of course for fun only; for programming projects in the commercial world it is rare to be able to choose based on the technical merits of a programming language only. Just among ourselves, though, it's sometimes fun to re-enact the great language wars, so long as we remember that it's no way to make the real decisions.

Remember to have fun programming,

James

From the Chair

Ewan Milne <chair@accu.org>

As this is my first regular "From The Chair" column in C Vu, I'd like to thank all of you who helped elect me to the post of ACCU Chair at the AGM — particularly everyone who failed to stand against me, thus ensuring that I swept to power unopposed.

As usual, the AGM took place during the ACCU conference, which was a great success once again. There are two people in particular I'd like to thank for all their hard work, not just in organizing this year's event, but for their efforts over the past few years. Francis Glassborow has been programme organizer since we started running conferences, and it is almost entirely due to his efforts that the programme has consistently reached such high standards. Not an easy person to replace, this task will now be shared by several others, including Kevlin Henney and me.

Another invaluable figure over the past 3 years has been Sue Bennett of Desktop Associates, our commercial organizer. Sue is now moving on to other things, and so sadly this partnership has come to an end with this year's event. We are considering alternatives for this role at the moment; hopefully I will be able to bring you news on this front soon. I'm reminded that Sue first approached us after reading in these very pages of our need for an organizer; so if there is anyone out there who would like to consider taking on the job, please do contact me. In the meantime, I'm sure everyone will agree that Sue has done a very smooth, efficient and fuss-free job — thanks again.

This shows how vital people are in the continuing success of the association. If you have any ideas that you want to share or develop, or if you have any feedback about our current activities, do feel free to contact me at chair@accu.org. I'd value any input on how we can take ACCU forward. Remember: the association is the sum of its members.

Membership Report

David Hodge <membership@accu.org>

Membership stands at 1107, 3 lower than our final total for last year, but there is a month and a half to go at the time of writing (mid May).

We have members in 39 countries other than UK, the top three being USA (120), Germany (33) and The Netherlands (20).

With this mailing you should get a copy of the latest handbook. If you spot any address errors then please let me know.

Renewal time will be along in a couple of months so if you would like to set up a standing order (dated August 15th) to make the renewal of your subscription easier then just send me a request by e-mail and I will send you the appropriate form.

Standards Report

Lois Goldthwaite <standards@accu.org>

In standards news, the main event was of course the meeting of the international C and C++ standards committees in connection with the ACCU conference the first week of April. The standards meetings were hosted by ACCU and sponsored by Microsoft, Intel, LDRA, and Hitek. The venue at The Holiday Inn in Oxford nicely accommodated all events, with plenty of room for the informal socialising which is so important. A number of ACCU members took advantage of the opportunity to meet with committee members and see first-hand how a standards committee works, and the BSI C++ panel has four new recruits.

The C++ committee launched the drive to C++0x by adopting the 2003 standard (the 1998 version plus the updates from Technical Corrigendum 1) as the current working paper. That means that as further corrections or more extensive changes are approved, they will be incorporated into the working paper following each meeting. Quite deliberately the committee established a period of stability following the issue of the original C++ standard, to allow compiler and library vendors time to implement the standard, so TC1 consists only of corrections where the original standard was found to be ambiguous, unclear, or contradictory. The objectives for C++0x are to make the language easier to learn and teach, to provide better support for generic programming and library creation, and to add significant new features to the standard library. Expect to see C++ evolving in these directions for the next several years.

The Technical Report on C++ Performance will soon be going out for formal vote by national standards bodies, as soon as, ahem, the editor completes the final proofreading duties. Meanwhile, you can download the final draft from http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1457.pdf. Please send comments to the editor.

ACCU is sponsoring Jon Jagger as a UK participant in the ECMA group working on standards for C# and CLI. They have been so impressed with his expertise and contributions that they have elected him convenor of the group!

SC22, the parent organisation of the international language committees, has organised a Linux Study Group to examine whether Linux is a suitable topic for a formal ISO standard. The LSG is meeting in London May 28-30. It is well known that the Linux community has a vigorous, albeit rather less formal, standards process outside the ISO umbrella, but one good reason for interest in an ISO-endorsed Linux standard is that a number of governmental and large corporate customers rely on formal international standards in their buying decisions. The ISO kitemark would increase their comfort level with a Linux solution.

There is already an SC22 Working Group tasked with maintaining the Posix standard (it's a joint effort with IEEE and the Open Group), and the BSI Posix panel is of the opinion that if SC22 does pursue work on Linux that a single body should have responsibility for both efforts. The UK panel has also approached members of the Linux/Open Source community for input in formulating the UK position. Please write to standards@accu.org if you want to participate.

The LSG met in London May 28-30, with delegations from eight countries plus representatives from Usenix, the Free Standards Group, the Linux Standard Base organisation, the Open Group, and the Linux Professional Institute. It was a good sign that members of the Linux Community – at least those who attended the meeting – could see the advantages to Linux users of attaining an ISO endorsement of the operating system. Yet to be determined is how delegates from national bodies will be able to cooperate with the Free Standards Group in the development and maintenance of Linux standards.

Conference Chair

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

Well this is my last report because I retired from organising the ACCU Conference programme at the end of the recent one.

It was the most successful of the seven conferences we have had. Indeed those attending the WG14 (C Standard) meeting that was going on alongside our event were amazed; several of them had thought that I was just talking up the event when I had told them about it. Perhaps a few more of them will be willing to speak in the future.

If you came you know it was a great event; if you are familiar with other conferences you will also know that ours is among the very best in the world. Many speakers come not just to speak but also to listen and that is a measure of the quality.

If you did not manage to come you missed a great gathering. I find it hard to believe you would not have enjoyed it as well as finding it informative.

Thanks to all who have made my tenure as Conference Chair a pleasure. As always after the event I had that warm glow which makes it so tempting to offer to stay on for another but I know in my heart that it is time that I handed the problems on to younger blood.

I hope that I will have the opportunity to meet with many of you in the coming years in a more relaxed state, where I can continue a conversation without having to rush off to solve yet another crisis.

At the time of writing we have more or less finalised the dates of the next Spring Conference as 14–17 April 2004. The venue will be the same as most seem to feel that it was the best we have managed so far and hotel staff are willing to address the problems to make it even better next time.

See you next year.

Dialogue

Student Code Critique Competition

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.

Student Code Critique 21

The problem

I'm creating a program that inputs three integers, and returns the sum, product, average, largest and smallest. Very simple, but I'm getting a bad return on one of my variables. Except for my "largest" variable, the others are returning the correct numbers. I'm hoping that someone can tell me where I'm going wrong.

```
#include <stdio.h>
int main(){
  int num1, num2, num3, sum, average
  int product, smallest, largest;
  puts("Input three different integers: ");
  scanf("%d, %d, %d",&num1,&num2,&num3);
        // read three integers
  sum = num1 + num2 + num3;
  average = ((num1 + num2 + num3)/ 3 );
  product = num1 * num2 * num3;
  if(num1 < num2 ) smallest = num1;
  if(num2 < num1 ) smallest = num2;
  if(num3 < smallest ) smallest = num3;
  if(num1 > num2 ) largest = num1;
  if(num2 > num1 ) largest = num2;
  if(num3 > largest ) largest = num3;
  printf("\nSum is %d",sum);
  printf("\nAverage is %d",average);
  printf("\nProduct is %d",product);
  printf("\nSmallest is %d",smallest);
  printf("\nLargest is %d\n",largest);
  return 0;
}
```

Critiques

From The Harpist

Please note that I am only doing this as a favour to Francis and not as a competition entry.

I am going to assume that this student has just started and is attempting one of those ridiculous exercises that thoughtless authors and instructors set. I assume that the student has not yet learnt about arrays.

Clearly the student has been taught that the structure of a program is:

Input
Compute
Output

This is the kind of brain deadening guideline that results in bad code. For example, it is because of this that the program declares three similar variables whose only difference is the order in which they acquire values from input. In fact almost all the student's problems are directly attributable to this 'design' decision. Experience teaches me that the instructor will probably be entirely happy with this aspect of the program and waste time on the implementation faults that arise because of it.

Yes, the student should have used the largest available integer type for input (assuming that the question specified input). The student should have considered that an average would generally be of a floating-point type. There is also the issue that when computing the product, we might get overflow. I am going to ignore these issues because it is likely that the student simply does not have the tools to cope with such issues. For example how do you detect overflow when dealing with integer multiplication? Note that actually overflowing an integer type results in undefined behaviour. (The best solution I can come up with is to do all the computation in a `long double`, check that the result is within the range of a `long int` – expressed as a `long double` – and finally convert the result to a `long int` – assuming that an integer result is required. Such programming techniques are way beyond what we should expect from a student who does not yet know about arrays and loops.)

Anyway here is my solution with commentary:

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

int main() {
  long int number = 0;
  long int largest = LONG_MIN;
  long int smallest = LONG_MAX;
// The above is an idiom for use when you are
// going to compute a largest/smallest value,
// initialise to the smallest/largest possible
  long int sum = 0;
  long int product = 1;
  puts("WARNING: If you give inappropriate \n"
       "values to this program you\n"
       "will get silly results\n");
// Note the use of puts to supply pure text
// output and the use of quotes to deal with a
// multiline text statement.
  puts("Type in a whole number: ");
  scanf("%d", &number);
// scanf is a poor function for general use
// but with the Student's likely state of
// knowledge it is appropriate. However
// warnings should be given that this is not
// a desirable long term mechanism.
  if(number > largest) largest = number;
  if(number < smallest) smallest = number;
  sum += number;
  product *= number;
// The above give an opportunity to discuss
// C's compound assignments and the need to
// use initilaised variables

// The following is simply a copy and paste of
// the above input and processing code
  puts("Type in a whole number: ");
  scanf("%d", &number);
  if(number > largest) largest = number;
  if(number < smallest) smallest = number;
  sum += number;
  product *= number;
  puts("Type in a whole number: ");
  scanf("%d", &number);
  if(number > largest) largest = number;
  if(number < smallest) smallest = number;
  sum += number;
  product *= number;
// Now output the results:
  printf("\nThe sum is %d",sum);
  printf("\nThe mean is is %d",sum/3);
  printf("\nThe product is %d",product);
  printf("\nThe smallest is %d",smallest);
  printf("\nThe largest is %d\n",largest);
  return 0;
}
```

Please notice that this code is a very good way to introduce the concept of a for-loop as a way to avoid repetitious source code. Once the code has been rewritten with a for-loop it can be developed to handle more input, both by hardcoding the number of items and by obtaining the number of items interactively.

Unless we are going to re-use the input (for example by sorting it) there is no need for the student to know about arrays. Indeed I would contend that using an array for a question such as this one is overkill, not least because it requires the introduction of dynamic memory management if the question is generalised to user determined amounts of input.

However, were I mentoring students in C++ I would give any student who submitted such a pure C solution a very low grade. Why? Well I believe that students who head off in this direction are actually making their C++ progress much harder by being pre-occupied with low-level detail (such as the use of the address-of operator with `scanf()`).

Yes, it is important that a student who presented the original code should be helped to understand why the code fails to work as expected. But the most important lesson is that bad design leads to logic errors. In this case there is no inherent difference between the three inputs and yet the submitted program handles them as if they were somehow different.

What saddens me is the frequency with which such opportunities to develop a student's design insights are ignored by tutors. It should not be four times more difficult to write a program that handles four numbers rather than three, and yet that, basically, is the consequence of the original design. Even without knowing about a for-loop, my design can be extended with a single copy/paste operation. That is not to encourage such, but to demonstrate that the solution of the problem for three inputs should generalise to the solution for n inputs. If it does not, it is a poor solution.

Critique by Catriona O'Connell <catriona38@hotmail.com>

The problem domain is not well-defined. The mathematical meaning of integer and the C language definition of integer are different. Problems of overflow can be minimised by using the largest possible representation (`long`) instead of `int`. However the GNU C compiler defines `INT_MAX` and `LONG_MAX` to be the same value (2147483647), so this doesn't necessarily help. The likelihood of problems might be further reduced by using compiler extensions. For example the GNU C compiler supports "`long long int`" with a maximum value of 9223372036854775807. The only way to avoid problems would be to read the input as a string. It would then need to be converted and processed using multiple-precision (MP) arithmetic routines. This is not likely to be within the student's ability. Without using MP routines there is no portable way to determine if arithmetic overflow or underflow occurs. If this happens then the results are undefined.

The user is asked to enter three different integers, but no check is made for uniqueness. This is important for the comparisons that are made to determine the largest and smallest numbers. If `num1==num2` then smallest/largest will have initial value (0), so depending on the sign of `num3`, the returned value could be wrong.

In the `scanf()` format string it is probably a mistake to include the commas. This forces the user to separate the input integers with commas. Since `scanf()` is already smart enough to bypass whitespace it would be sufficient to specify that the numbers be space-separated. If commas are required, then the user should be told of this.

The only reason I can see to read in all three numbers at the same time is that it avoids the problem of excess characters being left in the input buffer. This would cause problems if `scanf()` were called multiple times. The student would either have to program in assignment to a string for the remainder of the buffer or be introduced to the delights of non-assigning `scanf()` calls such as

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

to clear the input buffer.

The student does not check the return code from `scanf()` to ensure that three integers have been assigned before using what he/she thinks is valid input. This is a major error. It is almost impossible to recover gracefully from bad input using `scanf()` but it is even worse to plough on regardless.

`scanf()` should not really be used for interactive user input. If you are going to use it then it should be restricted to reading well-defined structure input - preferably computer-generated output. When you have enough experience to know how to use `scanf()` safely, then you probably know enough to avoid using it. There are very few cases where `fgets()/sscanf()` cannot replace `scanf()`.

A better solution avoids the use of `scanf()` and instead uses a combination of `fgets()` to read a buffer of input and `sscanf()` to parse that buffer. `fgets()` has the advantage of `gets()` in that the buffer will not overflow, but input may be truncated if it is longer than the buffer. `sscanf()` works like `scanf()` except that it reads formatted data from a string instead of `stdin`.

The student's method for finding the largest and smallest values does not scale well. The sum, product, largest and smallest values can be calculated as running values, avoiding the need to have all three integers entered at the same time. The average is calculated as the sum divided by the number of integers entered. There is no need to recompute the sum for this calculation. The problem domain does not define if an integer average is acceptable or whether a `double`/`float` would be more appropriate. The variable `average` could be defined as `double` and the average calculated as `sum/3.0`.

The accompanying code is my attempt at a re-write using `fgets()`/`sscanf()` with return code checking.

```
#include <stdio.h>
#include <limits.h>
#define LOOP_COUNT 3
#define BUF_SIZE 80

int main() {
  long num, sum, average;
  long product, smallest, largest;
  char buffer[BUF_SIZE];
  int i;
  int gotint;
  sum = 0;
  product = 1;
  average = 0;
  largest = LONG_MIN;
  smallest = LONG_MAX;

  for(i=0; i<LOOP_COUNT; ++i) {
    gotint = 0;
    while(!gotint) {
      printf("Input integer %d :", i+1);
      if(fgets(buffer, BUF_SIZE, stdin)
                                    != NULL) {
        if(sscanf(buffer, "%ld", &num) == 1) {
          gotint = 1;
        }
        else {
          printf("Error in sscanf()\n");
        }
      }
      else {
        printf("Error in fgets()\n");
      }
    }
    sum += num;
    product *= num;
    if(num < smallest) smallest = num;
    if(num > largest) largest = num;
  }

  average = sum/LOOP_COUNT;
  printf("\nSum: %ld", sum);
  printf("\nPrd: %ld", product);
  printf("\nAvg: %ld", average);
  printf("\nSml: %ld", smallest);
  printf("\nLge: %ld", largest);

  return 0;
}
```

By Anon

The first thing I noticed was the amount of code duplication. If the number of integers was changed from three then a comparatively large amount of change would be required to update the code.

The student appears to have split the code into distinct parts, one for input, a second for processing and a third for output, which is laudable. To keep this structure means all the inputs must be held until the input is complete; the obvious choice would be to replace the individual `num1`, `num2`, `num3` variables with an array, but we have been told that arrays are not appropriate.

This suggests that the design should be changed to combine the input and processing stages so that each input is consumed and discarded in a loop, with the variables being reused each iteration and the results generated incrementally.

As for the results being wrong I initially thought of overflow in the addition and multiplication but we are told these are correct, it is in fact the largest value that is wrong. Looking again at the three lines calculating the largest value it appears it could go wrong if `num1==num2`, in which case neither of the first two lines will fire and largest will be uninitialised, the result of the comparison with `num3` is therefore random. The student may have covered this by prompting the user for different integers but this is a very weak check. The same fault should be true of the calculation of the smallest value as the code is very similar, but we are told this is ok. A possible explanation is that the test numbers are likely to be small, and that the random values in uninitialised variables are unlikely to be zero in all the most significant bytes.

I wonder if the programmer is used to `else` clauses in the statements. Rather than reverse the operands as in the first two lines of the smallest and largest calculations, they could instead write:

```
if(num1>num2);
largest=num2;
else
largest=num2;
```

This would also overcome the problem with not coping with the **`num1==num2`** case.

[Look again, this has another bug, and one that will not always be detectable by the compiler. Francis]

As I have already decided this should be compressed into a loop, there will only be a single condition updated each loop.

```
if(newvalue>largest)
  largest=newvalue;
```

This presents the problem of how to ensure the first input is always assigned to `largest`. Here are two possible suggestions, either add special case code for the first loop:

```
if(loop count == 0)
   largest=newvalue;
else {
   if(newvalue>largest)
      largest=newvalue;
}
```

Alternatively, and my preference, is to initialise `largest` with `INT_MIN` from `<limits.h>` This is guaranteed to be small as the smallest input so the condition will work as expected. `INT_MAX` can be used for `smallest`.

Similarly the sum can be initialised to zero and product to one and both updated on each loop.

Finally there is the input method, which dictates how the program is used. Currently the input is separate from the command line so the user types:

```
name of executable (return)
3 2 1 (return) (or some other 3 numbers).
```

Is this is what is desired? Or should it run

```
name of executable 3 2 1 (return)
```

in which case the numbers should be extracted from `argc` and `argv` rather than a `scanf`. The numbers could even be input one per line:

```
name of executable (return)
3 (return)
2 (return)
1 (return)
```

This obviously depends on the user requirements. Seeing as this is a student assignment the exact method is probably less important than understanding the alternatives – so I will leave it as it is.

Not having programmed in C for a few years I vaguely remember something about preferring `atoi` over `scanf` for `int`s, but I cannot remember the rationale for this so I will stick with `scanf`.

The code definitely does need some input validation, the code expects three inputs but there is no check that `scanf` received them. The concept of never trusting user input is fundamental. At a minimum the `scanf` return should be checked, this will be equal to the number of variables successfully assigned.

Putting all these changes together with a few smaller items such as separating the constant 3, we get

```
#include <stdio.h>
#include <limits.h>
#define NUM_INPUTS 3

int main(void) {
  int product = 1;
  int smallest = INT_MAX;
  int largest = INT_MIN;
  int sum =0;
  int count;
  printf( "Input %d integers", NUM_INPUTS );

  for(count = 0; count<NUM_INPUTS; ++count) {
    int num;
    if(scanf("%d", &num) != 1) {
      puts("error reading input");
      return 1;
    }
    product *= num;
    sum += num;
    if(num < smallest) smallest = num;
    if(num > largest) largest = num;
  }

  printf("\nSum is %d", sum);
  printf("\nProduct is %d", product);
  printf("\nSmallest is %d", smallest);
  printf("\nLargest is %d", largest);
  printf("\nAverage is %d\n", sum/NUM_INPUTS);

  return 0;
}
```

From Walter Milner `<w.w.milner@bham.ac.uk>`

This is a good attempt at a solution for which you deserve credit. However there are several details of your code which require comment, and the overall design could be improved.

Firstly the details. Your declaration:

```
int num1, num2, num3, sum, average
```

omits a trailing semi-colon, but since you are implying your code compiles, I assume it was there and has somehow got lost.

There are several issues raised by

```
puts("Input 3 different integers: ");
scanf("%d, %d, %d", &num1, &num2, &num3);
```

If you remember, we discussed in class the limitations of **`scanf`** for input in real code. The first point is that the commas in the format string "`%d, %d,`

%d" will need to match up with commas separating data values in what the user types in – in other words we are hoping the user enters something like:

```
1, 2, 3 <return>
```

If they omit the commas you get a result like:

```
Input 3 different integers:
1 2 3
Sum is -1717986919
Average is -572662306
Product is 687194768
Smallest is -858993460
Largest is 1
```

The use of a comma is made worse by the possibility that the user may separate thousand by using a comma, such as

```
1,000   2,000   3,000 <return>
```

These problems would be partly solved by changing the prompt to something like 'Enter three different integers separated by commas'. However users do not always obey instructions, and good code should cope with this. In other words, data input should be *validated*. For example suppose the user enters just 2 numbers, or non-integers, or numbers which are equal? Real code would check this, for example by inputting the data as a string, and then parsing it to check that it did consist of three different integers. We will look at this in detail in a later class.

The line

```
average=((num1+num2+num3)/3);
```

also contains several points. Firstly, the outermost pair of brackets is superfluous. Secondly you have just calculated

```
num1 + num2 + num3
```

so why do it again – in other words say

```
average = sum/3;
```

Thirdly we have a major point about *data type*. Suppose the user entered 1, 2 and 5. The average of these is 2.6667, which is not an integer. In other words average needs to be a floating point type, single or double. But even this is not enough, since

```
double average;
...
average=sum/3;
...
printf("\nAverage is %lf", average);
```

still gives 2.0000 for input of 1, 2 and 5. The problem is that `sum/3;` is the division of an integer by another integer, and the compiler uses integer division, which is 'guzinter', as in 5 guzinter 21 4 times remainder 1. If we want the compiler to generate code for floating point division, one way to do this is `average = sum/3.0;`

If we divide by 3.0 (a floating point constant) rather than 3 (integer) the compiler converts `sum` to floating point, then uses floating point division as we want. The difference between 3 and 3.0 is a small point (as it were) but an important one.

Another issue is the question of overflow. For example:

```
Input 3 different integers:
2000, 2001, 2002
Sum is 6003
Average is 2001.000000
Product is -577930592
Smallest is 2000
Largest is 2002
```

because on a system with 32 bit `int`s the product is larger than the largest integer which can be represented. This cannot be fixed by say using `long` instead of `int` – we could still input a value that was too large. We will see in a later class that `<limits.h>` contains a value for `INT_MAX` that we can use to deal with this.

Then we turn to the overall design – which in this case is not related to language-specific details. The overall plan of your code is

1) input the 3 values
2) process them to calculate the required values
3) output the results

The key question is about the storage of the input values – do we need to store all three values *at the same time*? Or could we deal with them *one at a time*, using this kind of plan:

```
for each value
  input it
  process it
```

Now some kinds of process require all values to be available. For example if we needed to find the standard deviation of some numbers, we need to calculate how far each is from the mean, so we need to first find the mean, then calculate the distance from the mean. Unless we input them all twice, this requires us to hold all values in memory at the same time. Similarly sorting them requires all values to be available at the same time.

But in this case by using an idea like 'the biggest one so far' we can do all the processing with only one value in memory at any one time. For example -

```
input first value
biggest so far = first value
for the rest
  if new value > biggest so far
    biggest so far = new value
```

We can do the equivalent for the smallest, the product, the total, and hence the average. Why bother? Because with this new design

1) It is trivially extensible beyond three values to any number
2) The storage requirement is small (one integer) and constant, independent of the number of values.

The first of these is probably the most important. Finding the largest of 3 values with the original approach required 3 ifs. Finding the largest of 4 requires a re-writing of the logic of the program. With the re-design, finding the maximum of 4, or 4000, is a trivial change.

A coding with this design follows. We change how many numbers to deal with by changing `howmany`, which could be done at run-time:

```
#include <stdio.h>

int main() {
  int num;
  const int howmany = 3;
  double average;
  int product, smallest, largest, sum;
  int index;
  printf("Enter first number: ");
  scanf("%d",&num);
  smallest = largest = sum = product = num;
  for(index = 1; index < howmany; index++) {
    printf("Enter next number: ");
    scanf("%d",&num);
    if(num>largest)
      largest = num;
    if(num<smallest)
      smallest = num;
    sum += num;
    product *= num;
  }

  average = sum/(double)howmany;
  printf("\nSum is %d",sum);
  printf("\nAverage is %lf", average);
  printf("\nProduct is %d", product);
  printf("\nSmallest is %d", smallest);
  printf("\nLargest is %d\n",largest);
  return 0;
}
```

The Winner of SCC 21

The editor's choice is: **Catriona O'Connell**. Please email francis@robinton.demon.co.uk to arrange for your prize.

Student Code Critique 22

(Submissions to francis@robinton.demon.co.uk by July 6th)

When critiquing this code please do not restrict yourself to explaining the writer's perceived problem but deal with all aspects of the code. There are some nasty problems as well as issues of good coding practice.

The Code

Please look at the following code that is just playing with container of objects.

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

class A {
public:
  A(int p1= 0) : x(p1) {
    cout<<"CTOR"<<endl;
  }
  A(const A& a) {
    x = a.x;
    cout<<"COPY CTOR"<<endl;
  }
  ~A() {
    cout << "DTOR: address = " << (long)this <<endl;
  }
  friend ostream&
    operator<<(ostream& out, const A& obj);
private:
  int x;
};
ostream& operator<< (ostream& out, const A& obj) {
  out << "x = "<<(obj.x)<<endl;
  return out;
}

int main() {
  vector<A> v;
// Be inefficient, not production code
  for(int i=0; i<3; i++)
    v.push_back(A(i));
// Lo behold, container of objects
  copy(v.begin(), v.end(),
       ostream_iterator<A>(cout, ""));
  cout << endl;
  A * pa = &(v.back());
// pa points to address of reference
// to the last element
  cout << (*pa);      // info
  cout << "pa = "
       << (long)pa << endl;  // address
  cout << "call pop_back()" <<endl;
  v.pop_back();              // Line A
// destructor WILL destroy last element
// Now pa points to deleted memory, should crash
  cout << (*pa);             // Line B
  cout << "pa = " << (long)pa << endl;  // address
  delete pa;                 // Line C
  return 0;
}
```

Lines A and C delete the same memory, which shouldn't be allowed. But program runs fine allowing dereferencing at line B. Why?

The Wall

Letters to the Editor

C++ Primer book review - thank you.

Dear Francis and the team at ACCU,

It has been quite some time since we initially looked at organising a reader survey for the forthcoming 4th edition of the C++ Primer by Stan Lippman, Barbara Moo and Josée from Addison Wesley.

The survey was very successful, and the editors and teams at Addison Wesley in the US are currently completing the analysis of the survey. We appreciated all your assistance and support in promoting this survey to your members. In our initial offer to the members of the ACCU, 5 members had the chance to win an Addison Wesley book of their choice for taking part in the survey. I am now pleased to announce the winners and their choice of book.

Nigel Rafferty who chose 'Advanced CORBA Programming with C++' by Michi Henning and Steve Vinoski

Paul Johnson who chose 'C++ Templates: The Complete Guide' by David Vandevoorde and Nicolai M. Josuttis

Alan Lenton who chose 'C++ Gotchas: Avoiding Common Problems in Coding and Design' by Stephen Dewhurst

David Sykes who chose 'Design Patterns: Elements of Reusable Object-Oriented Software' by Erich Gamma et al

John Kewley who chose 'Modern C++ Design' by Andrei Alexandrescu

The winners are now all receiving their books. I wondered if you'd announce these details in the next C Vu and whether you were interested in supporting this by reviewing these books along side the announcement? Thank you again for your support and please let me know if you're interested in reviewing these books in the next C Vu.

Kind Regards

Melanie Backe-Hansen

PR Executive - Professional Computing

Consider this an announcement; I will be contacting the recipients of these prizes to ask them for brief comments on their choice of books. — James

Sincerely Grateful!

Hi James,

I wanted tell you and at the same time let everyone that is a part of the ACCU know how humbly lucky I feel to be involved in such a dedicated group of people, who in many ways reach out to mentor and help so many people.

While reading through the Mentored Developer threads, a project was presented involving the study of C++ Programming Language 3rd Edition, written by Bjarne Stroustrup. I wanted to get involved as a student but knew unless I could get the book, I would have to observe the project rather than actively participating, and even then it would be difficult to reference most of the discussions. I was hoping I could get the book but my finances have been very low and could not afford even a used book.

Basically I've been unemployed and using benefits to attend school for 14 months and in January was forced to get whatever I could I'm presently working as a Security Guard for very low wages and barely surviving, I am supporting a wife and 3 children which prioritizes things.

Anyway, I wanted to post a recognition to the Mentored Developers and to the most honorable Mr. James Bannon to who sent me a copy of this book so that I may get involved, I am deeply moved and humbled by these actions and hope that I may do the same in the future.

Sincerely Grateful!

Michael J. Murphy

ACCU's success or otherwise is determined by the contributions of its members, and how they choose to form communities. Mr Bannon's generosity is just one example of the many people who work more or less visibly through ACCU to help each other. And to James Bannon: my apologies for singling you out. I am sure you intended only to find a good use for a spare book — but it is the very spirit of choosing to help each other that ACCU is based on, and I too am proud to be part of an organisation where so many people help each other (and are then surprised to discover that it is not always taken for granted). — James

Francis' Scribbles

by Francis Glassborow

Benign + Benign = Disaster

You will have to wait to the end to understand that heading. The story starts with one of the test students for my book who after three days of bashing his head against a brick wall finally emailed me with a problem that I could distil down to why this code will not compile:

```
#include <iostream>
using namespace std;
double distance(int, int);
int main() {
  int abc(0), xyz(0);
  cout << distance(abc, xyz);
}
```

Try it. I can fix the problem this time by using `::distance(abc, xyz)`. But how on earth would an inexperienced programmer know to try that? And how could anything in the `iostream` header conceivably cause a problem?

A little experience will lead you to guess that distance is used as a function name somewhere in the Standard C++ Library. The error messages generated (all five of them) will give you a clue that the `iterator` header is part of the problem. Presumably the `iostream` header includes that (and a little thought will show that that inclusion is almost necessary).

The instinctive moral drawn by not a few C++ experts is that that the above code just demonstrates the evils of using directives. Sorry, I beg to differ. Suppose that your code needs to use the `distance` function from the Standard C++ Library as well as a `distance` function from a third party library. You might be of the school of thought that prefers `using` declarations.

Try replacing the `using` directive in the above code with:

```
using std::cout;
using std::distance;
```

which is the style advocated by Andy Koenig in *Accelerated C++*. Now you get the same errors but the only way to fix them is to remove the `using std::distance;` declaration.

As an aside, please note the difference between `using` declarations (that effectively make the name concerned behave as if declared in the current scope) and `using` directives that simply make all the names in the specified namespace available for overloading and for use without qualification. A `using` directive is NOT equivalent to providing `using` declarations for all the names declared in the relevant `namespace`.

The Next Step

Faced with, to me, unnatural behaviour from all the compilers I had to hand (that for once all agreed with each other, Comeau, CodeWarrior, Borland and MinGW) meant that I had to try to find out what was happening.

First I looked up distance in my copy of Nico Josuttis *The C++ Standard Library*. Here I found:

```
template <typename InputIterator>
iterator_traits<InputIterator>::difference_type
  distance(InputIterator pos1,
           InputIterator pos2);
```

Well, the error messages mention `iterator_traits` so that looked as if I was moving forward. But why was it trying to instantiate `iterator_traits<int>` for a template function that could clearly never be called because an exactly matching non-template function declaration was already in scope and must always be preferred?

The answer to that question is simple: the compiler must generate the overload set first before determining a best match. That seems reasonable and avoiding special cases makes sense. But why is it trying to place a function it cannot instantiate into the overload set? Hang on to that question for a moment and have a look at this code:

```
template <typename T>
class x_traits;

template<typename T>
x_traits<T>::return_type foo(T, T);

int foo(int, int);
int main(){
  foo(1, 2);
}
```

Do you think that code should compile? It does but only after I remember to add `typename` before the return type of the `foo` function template, without a murmur and it selects the correct version of `foo()`. However notice that I have only declared `x_traits`, I have not defined it. So what happens when I add the definition? For example:

```
template <typename T>
class x_traits {
  typedef typename T::return_type return_type;
};
```

It depends where I place that definition. If I place it before `main()`, the code fails to compile. If after, everything is fine. In other words as long as the compiler cannot try to instantiate the return type while generating an overload set all will work as desired. However the moment we provide that extra information, the compiler grabs it and refuses to compile the code. Do you feel something is not quite right?

Archaeological Research

I am grateful to John Spicer of EDG for having dug away into the past to find an explanation of what is clearly silly. Here is what the C++ Standard has to say on the subject (and it took several people quite a lot of winnowing down to find that this was the problem):

> **14.8.3 Overload resolution**
>
> A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (14.8.2) and checking of any explicit template arguments (14.3) are performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction and checking succeeds, **the template-arguments (deduced and/or explicit) are used to instantiate a single function template specialization** which is added to the candidate functions set to be used in overload resolution. **If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template**. ...

I have emphasised two sections.

In 1996 clause 14 of what was to become the C++ Standard was being reworked editorially. One of the changes made was to systematically replace 'generate' with 'instantiate' and another was to replace 'template function' with 'function template specialization'. These were perfectly sensible editorial changes because 'generate' was almost always used as synonym for 'instantiate', and there is no such beast as a 'template function.'

Before that date the first piece of emphasized text read:

> the template-arguments (deduced and/or explicit) are used to generate a single template function

That is substantially more vague and would not have led implementors to try to actually instantiate the function rather than just use its signature. Those two substitutions were generally benign but together leads to the current position where many high quality compilers are rejecting the code I started with.

However the second piece of emphasised text leads to looking further to understand what it means to say that 'argument deduction fails.' This leads to14.8.2/4, which explicitly says:

> When all template arguments have been deduced, all uses of template parameters in non-deduced contexts are replaced with the corresponding deduced argument values. If the substitution results in an invalid type, as described above, type deduction fails.

The problem is that the failure occurs whilst the compiler is trying to instantiate a template because that is the only way that it can follow the requirements of 14.8.2. But why are compilers treating that failure as making the users source ill-formed? I do not think that the compilers are doing the right thing here. If I understand the text correctly, the very first error in trying to instantiate the return type should cause the compiler to silently remove the function template from the list of overload candidates.

That means that compiler implementors have to be more careful of the context in which a template instantiation is taking place.

Is This the End of the Story?

I think not. 14.8.2's requirements re generating an overload have some serious things to say about `export`.

In order to determine if argument deduction for a function template such as `std::distance` has succeeded the compiler has to see the relevant

template definitions. For example consider:

```
template <typename T>
class x_traits;

template<typename T>
x_traits<T>::return_type & foo(T, T);

int foo(int, long);
int main(){
  foo(1, 2);
}
```

Now `foo<int>` is the best match if it is allowed to be part of the overload set. However under the requirements of 14.8.2/4 it should not be considered if `x_traits<int>` cannot be instantiated. However the compiler cannot know the answer to that question until it can see the definition of `x_traits`.

Under the inclusion model for templates that will not happen (I think) but once we bring `export` into play all the bets are off.

Conclusion

This all started because a novice was faced with code that would not work. That happened because I, as an author, had checked my code, moved on, re-organised a library and did not check that the re-organisation had not broken anything. I knew in my own mind that it could not have done. In one sense I was right, because it should not have done but that is not really an excuse.

I guess that this was not the first time that the issue has arisen somewhere during the last five years. I guess that it has just been treated as a reason for avoiding `using` directives (and possibly declarations).

We all know that C++ templates are problematical and so we do not make enough effort to understand when things go wrong. I think we do have a real problem hiding within the novice's code not compiling. I think we have to address that problem. I do not think that it is going to prove at all easy.

I think that there is a subtle but serious break in the overload rules when function templates that have templated types dependant on their deduced type arguments. I would hate to be a compiler implementor faced with this problem but I think that we should not just dismiss it.

It is a fundamental precept of programming in C++ that the meaning of code should not silently change because we make a definition visible. In my opinion the current rules for overload resolution break that precept.

Problems

Problem 7 revisited

My thanks to James Holland for his contribution (should be elsewhere in this issue) of an article on a simple polygon fill).

Problem 8 report

Many of you know that C++ strings can contain embedded nulls. However how do you get them there? And what about C arrays of `char`?

Commentary

I wonder what you expected the results to be for the three mini-programs? If you are familiar with C I would expect you to have realised that the process of initialising an array from a string literal takes no notice of the exact `char`s that make up the literal. The compiler simply creates an array of sufficient size to copy all the characters from the literal including the final null terminator.

That means that the first two mini-programs have identical output.

```
int main() {
  char const * mess = "One\0Two\0Three";
  cout << mess << '\n';
  cout << mess+4 << '\n';
  cout << mess+8 << '\n';
}
```

And now repeat the exercise for:

```
int main() {
  char mess[] = "One\0Two\0Three";
  cout << mess << '\n';
  cout << mess+4 << '\n';
  cout << mess+8 << '\n';
}
```

However the rules are very different for `std::string` constructed from a string literal. The constructor for `std::string` that takes a single parameter of type `char const *` uses it as a null terminated array of `char`. That means it stops when it hits the first null.

I hope you did not try to run the program because it has undefined behaviour and, at least in theory, could have done bad things to your computer.

You need to use a two argument `std::string` constructor, the one that takes `char const *` and `size_t` arguments.

Please note carefully that while `std::string` handles embedded nulls internally it does not work well with external data that has embedded nulls.

Also note that if you wanted the same output as for the above you will have to work harder because the indexing that works fine for C-strings just produces a single character for C++ ones.

```
int main() {
  std::string mess("One\0Two\0Three");
  cout << mess << '\n';
  cout << mess[4] << '\n';
  cout << mess[8] << '\n';
}
```

Problem 9

Consider:

```
int foo(int i) { return ++i; }
int bar(int i) { return i; }
int main() {
  int i(21);
  cout << (foo(i) + bar(i++));
}
```

and:

```
int fooref(int &){return ++i;}
int barref(int const &){return i;}
int main() {
  int i(21);
  cout << (fooref(i) + barref(i++));
}
```

Now we all 'know' that passing by value and passing by `const &` are interchangeable and have the same semantics (as long as the called functions do not use `const_cast<>` or try to modify the parameters inside the function body). So comment on any differences between the above programs, and what output you should expect for each.

Cryptic Clues for Prizes

Well that was a free exercise as I have yet to receive a single response. However I will persevere in the hope that as time goes by some of you will get the hang of cryptic numerical clues and contribute some of your own.

The answer to 'A tailless Roman mile.' is 1049. The 'tailless' is a pretty standard cryptic hint to remove the last letter (or letters). Actually 'endless' would have been better in this case. 'Roman' is a hint that letters are to be used as in roman numbers. Put those together and we have MIL.

To extend what is available I have generalised the rule for evaluating letters as roman numbers by specifying for my own purposes that any time a small 'value' comes before a larger one it represents subtraction. This is applied strictly left to right. For example:

IVXDL becomes (50 + 500 –(10 –(5 –1))) = 544.

And here are a few more example clues:

It is not bitter in a roman bar. (MILD = 1451)

Almost what mountaineers would do on Vesuvius. (CLIMb = 849)

Roman evil has no beginning. (eVIL = 44)

A boundless Roman likeness. (sIMILe = 999 + 49 = 1048)

Now you understand the mechanism try to come up with a few clues for 'roman numbers' and send them in. However the prize this time goes to anyone who can decipher the 3-digit number given by:

It looks like a call for help.

Careful now, read the whole clue. It isn't 999 or 911 (those would be clued with some version of 'Call for help.'

Francis

Features

Professionalism in Programming #20

Software evolution or software revolution[1]?

Pete Goodliffe <pete@cthree.org>

Change in all things is sweet. Aristotle [1]

If only software grew like plants. You'd put the seed of an idea into some fertile programming soil, add a little water and keep the conditions just right. Maybe at first you'd have to do some work to tend it: put a little rod in to help the thing grow upright, and cover it to keep the birds off. In time a seedling would sprout and grow, and when the program plant was big enough you'd be able to release it to the world. If it needed a little extra functionality you'd just keep watering it, perhaps add some fertiliser, and it would continue to develop. The trunk would strengthen in order to support the new branches and the plant would keep in perfect balance. If it was growing in a direction you didn't like then a little pruning would soon set it straight.

Ah, if only it was an ideal world. Sadly it's not. Not by a long chalk.

The truth of the matter is that software *is* a live entity. OK, we're not quite to the point where it's sentient or organic, but it has a life of sorts. It goes from being conceived, develops steadily, then comes of age and is sent out into the big wide world to make its living, hopefully garnering respect, admiration, and ultimately fulfilment. It may continue to develop, perhaps to the point where it gains a bit of a middle age spread and no longer has the handsome looks of its youth. Over time it gets tired and old, and is eventually retired, put out to pasture in the digital knacker's yard where it can gracefully die.

That's an idealistic view of the lifetime of a piece of software, but it's reasonably accurate. We need to look at how we cultivate our programs, especially after the initial round of development is done and dusted. Unfortunately, programs require thoughtful tending and seldom receive the care and attention they really deserve. What can we do to prevent early death from a slowly spreading code cancer?

To answer this we'll work backwards. We'll take a look at the symptoms of bad code growth, explore how we grow our code, and using this determine some strategies to develop healthier software.

Software rots

Bad things happen to good code. No matter how well you start off, no matter how good your intentions are, no matter how pure your design and how clean the first release's implementation, time will warp and twist your masterpiece. Never underestimate the ability of code to acquire warts and blemishes during its life.

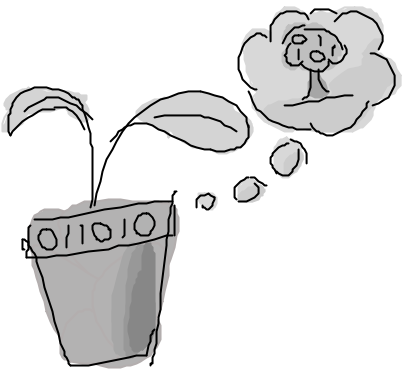
The 'maintenance' phase[2] of software development is always the longest, and overall where most of the effort goes – even if this effort is not scrunched into the compact focused ball that the initial design/development work is. We must explode the myth that software only develops during its initial stages of life. Boehm states as much as 40-80% of total development time is spent in maintenance [2].

During the initial development stages you can keep a firm grip on the code and work it around as much as you like, within the available time constraints. After it has been released you're generally more restricted. These restrictions may be practical:

- changes have to be minimised as much as possible to reduce their impact on the carefully tested code,
- APIs have been published, and once used by third parties have become much harder to modify, or
- the UI is known by users and can't be changed gratuitously.

These restrictions may be psychological, the developers' (erroneous?) preconceptions:

- it works this way, so we can't change it like that,
- it's too much work to revise the architecture at this late stage in the game, or
- it's not worth making this modification 'properly' now, the product won't be around for very much longer (as if).



The restriction may even be a simple lack of understanding – another programmer may not understand original author's mental model of the code, and this prevents them making the most appropriate modifications.

Despite all this, after a release code is never expected to stand still. No matter how well it was tested there will always be odd faults that crop up which will need fixing. Customers demand that new features are added. Requirements change under the development team's feet. Assumptions that were made during development prove to be incorrect in the Real World, and require adjustments.

The bottom line is that more code gets written after you think the project is completed. Where the fine line lies between 'maintaining an existing product' and working on new development of the 'next version' is a moot point. Whatever you call it, the original code base gets modified. Sometimes by the original author, often not.

And this is where the rot sets in. In fact, it's a damned-if-you-do-damned-if-you-don't scenario. If you never again touched the code, if you don't keep the program up to date with fixes and modifications, the program will degrade. In the worse case it will *stop working* as the platform

More metaphors for software construction

We've already examined the metaphor of *building* and looked at what it tells us about the software construction process [4]. In this article I've carelessly thrown around a few other metaphors. It's worth taking a quick look at these and see what they might imply about our methods of programming.

Growing software

This metaphor refers to how we *extend* our existing software, usually by adding new features. Bug fixing isn't really growth, it's more like tending to diseased parts of the system. Whilst we do indeed see our code growing as we add to it, programming is not the perfect analogue of a plant growing – we have far more control and influence over the growth process than we do with a seedling. Code is grown more like an oyster making a pearl, slowly, by the progressive addition of small extra parts.

I've heard of server farms, but the idea of a code farm is at once appealing and frightening.

Evolving software

Another common construction metaphor is the *evolution* of software. A lot of formal research has gone into this topic, but we're not going to get too academic here. We often start off with something like a single-celled code organism and gradually see it change and get adapted into a larger, more complex beast. This development is often an incremental process, so we see the software develop through a number of evolutionary stages. However, there are a few key differences to biological evolution. *We* are the ones deliberately making these changes; the software doesn't develop itself. Also we don't tend to develop many different revisions and employ natural selection to chose the best.

We do have the opportunity to iteratively improve the quality of our code, which mimics an evolutionary development in some respects. We can use experience gained from previous releases to adapt the code to its natural habitat, ensuring its long-term survival.

1 Well, I've seen a lot of revolting code in my time...

2 That is, work done after initial delivery which isn't considered a major new release.

it runs on changes, or the assumptions made become out of date. The 'Y2K' bug is a glorious example of this. Maybe the program will just putrefy as competing solutions develop more features, and gain more popularity. Untouched code slowly rots away.

However, if you do make extensions and fixes, the code doesn't only grow. It also rots. Fixing a fault usually sees the programmer introduce more faults as a side effect. Brooks found that as many a 40% of fixes introduced new faults [3]. The Programmer's Drinking Song (sung to the tune of '100 Bottles of Beer') written by a minstrel unknown, sums this up neatly:

99 little bugs in the code,
99 bugs in the code,
Fix one bug, compile it again,
101 little bugs in the code.
(Repeat until BUGS == 0)

It's not only newly inserted faults that can rot our code. Even functionally working modifications can cause code blight. Quick and dirty fixes pile on top of one another, putting nail after nail into the original neat design's coffin. The more you maintain carelessly and degrade the structure of code, the harder future maintenance gets. The plant analogy above is a good one. Many, many quick modifications don't make the trunk grow any stronger. The more heavy branches you add to the top of the system, the less stable the entire code base is. Eventually it totters over.

Does all this sound unduly pessimistic? Surely code won't 'rot' if you're careful? Adequate care does not seem to be taken in today's software industry. It's a culture thing. Programs have a habit of hanging around much longer than they were ever intended to. So many quick hacks live on, well past their expected life.

The warning signs

We should be on the look out for rotten code. Beware of the telltale signs of bad code, whether you're writing it now or stumbling across someone else's mess. Rot sets in with any change that leads to a lack of clarity, or makes the system more complex. Unnecessary complexity comes in many guises. Here are some of the flashing red lights and klaxon calls:

- Many separate bits of code crop up to do the same thing (e.g. to convert string formats, or display warning messages).
- It's no longer clear where to go to find a certain bit of functionality.
- A piece of data keeps getting converted between different type representations as it works its way through the system (e.g., display data is transferred between std::string, char*, unicode, UTF-8 and back again)
- APIs get 'blurred'; once neat interfaces are now too broad in scope, with new features being thoughtlessly added.
- Bits of private API leak out to be public, to allow other quick hacks to work. Private implementation member variables get exposed.
- New features are added with no documentation.
- There are functions with hulking great parameter lists.
- A function requires many parameters that it never uses, that just get passed through to a subordinate function.
- APIs change rapidly between code revisions.
- There are many big complex classes, or long functions.
- You find code that's too scary to even think about improving.
- Function names are misleading.
- The code is littered with incredibly complex functions, with many nested loops and special-case handlers.
- Complex module interconnections and dependencies mean that a small change in one place ripples out across the entire code. Think about the surface of a liquid; how viscous is your code? Does a small change in one secluded tributary disrupt something on the other side of the lake.
- The code compiles 'noisily', with many warnings generated.
- The code is littered with workarounds, fixes for symptoms not causes, which hide the real problem. The edges of the system get cluttered up with these rather than fixing the fault at the centre.

Many of these forms of rot are particularly visible in the code, and can be seen with a quick inspection or even the use of certain tools. However, there is a class of more subtle 'invisible' degradations that usually occur at a higher level than the syntactic gunk. Modifications that fudge the original code architecture or subtly work around normal program conventions are much harder to spot until you're immersed deeply in the system.

Code rot often sets in more voraciously when the original author of a piece of code leaves the company or project. Although code ownership is not necessarily a good thing, and is seldom written in stone in a company culture, the creator often takes a pride in their work and does housekeeping on their source files, even when other people make modifications. Once they've gone, this maintenance role slips away, and their files begin to rot more quickly.

Why do we make such a big mess of code? The answer is simple: complexity. A program is a huge collection of information organised on several levels: the overall architecture, its design in a particular language, the interface mechanisms to the outside world, the actual implementation of each little bit of code, and so on. That's a lot to understand before you start working with a chunk of code. There is seldom enough time to work out how a few lines actually work, let alone how they fit into the overall picture. We haven't yet learnt to manage this vast complexity.

How does code grow?

No code development really follows the classic model of: lock down all requirements, design completely, code completely, integrate, test, release. Unexpected modifications happen to an existing code base. New pieces get grafted in somehow. It's an incremental development cycle towards ever shifting goalposts.

In reality code growth happens by one of the following mechanisms, loosely ranked in order of disgust.

- **Luck**
  The most frightening way to make code, and not as uncommon as it should be. Code that grows by luck never had any design. It is modified without thinking. Its structure is down to happenstance, and the fact it works is attributable to miracle.
  Even if code was originally designed carefully, a lot of maintenance modifications can follow this happy-go-lucky approach. Hit-and-hope fixes may not be real solutions, they may just mask the immediate problem and make the real fix harder later on.

- **Accretion**
  We need a new feature added. Doing it properly would probably involve ripping up the interfaces between a few key modules and revising a lot of code. There's no time to do all this, and it would probably be too complicated for us, anyway. We'll just graft on another clump of code. It'll hang off one of the existing modules, well, perhaps a few of them, and use its own protocol to talk to them, rather than any existing mechanism. We'll have developed something demonstrable really quickly.
  Forget the fact that it's a monstrous kludge. Never mind that the performance will be awful. It's not important that the modules no longer have any clear roles and responsibilities. The system won't have a neat design any more, in fact it will look a lot more philosophical, and maintaining this in the future will be a nightmare. But we'll get this version out quickly, and we don't have any time to do the work properly now, anyway.
  Maybe later on we'll come back and do it properly…

- **Rewrite**
  What happens when you recognise that a bit of code is truly awful, not easily understood, or fragile and can't be extended? It needs a rewrite. If this is done based on what was learnt the first time around, it's usually much quicker to complete and of a much higher quality. However, rewrites rarely get done.
  Rewrites get riskier as you attack more at once. Rewriting a whole product is a different thing from rewriting a troublesome function or class.
  Good modularity and separation of concerns should mean you don't have to rewrite a whole system, just a module, keeping the original interface. If the interface sucks, or the reason you need a rewrite is that the system isn't modular enough anyway, then it's a different story.

- **Refactor**
  A formalised cousin of rewrite. If your code is mostly OK but bits of it need some work you can *refactor* these unpleasant parts. Refactoring

is a mechanism of making changes to a body of code in order to improve its internal structure, without changing its external behaviour. It improves the design so you can change it more easily in the future. It's never about performance improvement, just design enhancement. Not as drastic as a complete rewrite, refactoring is a series of gentle massages of what you already have.

In many ways this is a fancy name for particular kinds of improvement. Martin Fowler has formalised it and described a systematic improvement process documenting a number of small, understandable steps [5].

- **Design for growth**

  If you have some understanding of the ways your code will expand in the future, say some features have been deferred until the next release, you can carefully design the system so it's easy to make these future additions. Most of the time this doesn't make the job much harder.

  Even if you don't know the set of future additions, careful design can factor in an amount of room for growth. A good extensible system needs clearly defined interfaces and hinge points for new functionality to be plugged in. Be careful that this isn't an exercise in chasing the wind[3] though, trying to guess the future when you don't have a clue how the system will expand. Any extra design features come at the cost of complexity. If you correctly guess where this complexity is needed you win, if you guess incorrectly you'll make an unnecessarily complex system. This is the danger of over-design, and it's especially likely when design occurs by committee.

  There is a school of thought, in Extreme Programming for example, that insists on the absolute simplest design that can possibly work in any given situation. This is *(or sometimes just appears to be — jad)* at odds with the design for growth mentality. Exactly how much design for growth you should employ can be a hard balance to strike.

The problem with writing code is that doing it well takes a long time and a lot of effort. You may make false starts down wrong design alleyways, and encounter flaws in the original plan that need piloting around, whilst putting up with huge product redefinitions en route. There is *never* enough time to accommodate all this, so we try to shoehorn as much as we can into the limited time available. Something has to give, and it's usually the purity of the code.

So code is shaped by design, yes, but also by the organisation that built it and its life history. A case in point: We have some particularly baroque user interface code. It works (mostly), but is pretty much unfathomable unless you devote a significant portion of your life to meditation at the temple of complex code. It's just an intense lump of intertwining logic with no discernible architecture. And it's like that for a reason.

The code was initially created as a simple one-off television UI for a single customer, and was only ever specified to be a minimal closed system. It used the simplest communication protocol possible with the main system, and was as lightweight as it could be. However, it was then sold as a feature to a second customer, who wanted it to look different. A skin feature was hacked on. Then it was sold to another customer in a different country. Internationalisation was hacked on. Then it was sold to another customer, who wanted some new UI facilities, so these were hacked in. This story continued. For a long time. Today this simple UI component is unrecognisable from its former self. It's pretty much unmaintainable; each addition has been a quick hack since the whole thing has always needed rewriting.

If the initial design had incorporated all these features the code would still be lean and logical. However, it would have been too much work up-front and the company probably would never have started the project. Pity the poor programmers who work with this.

Believe the impossible

Perhaps the reason we see so much bad code and so many quick hacks is the mistaken belief that it takes longer to do the job properly. When you factor in the time spent debugging, and the ease of making more modifications later on this proves to be a wrong assumption. You may be able to close a single fault report quickly by hacking out a fix, but it's not a complete solution. As professional programmers, we need to be aware of this, and take responsibility for what we do to code.

3 Ecclesiastes 2:11

In the corporate world, there is often a management expectation of quick fixes. It's reasonably easy to show a manager that a five tonne block of concrete stuck on top of a flimsily erected flagpole won't stay up for very long. It's harder to make them stand underneath the thing. And it's *much* harder to get the same message across when we're talking about software. They just don't get it. As far as most managers are concerned programmers are magicians. They practice dark mystical arts and have limitless powers. Managers just tell them what to do and when to do it by, and it will happen, however many all-nighters they need to pull.

And, being gifted and dedicated, sometimes we come through on this assumption. Doing so can actually make matters worse as management will now expect that this tactic will always work, and that it's *our* fault when it doesn't. Sadly, there comes a time when that kind of hacked up software just cannot be made to expand any more, when it really just wants to keel over and find its final resting place in a quiet dark corner somewhere. Management will not be happy[4].

Code growth is easier if the culture of a company is to develop software in small iterative steps. This way evolution is almost built into the design strategy, and rewriting some code parts to accommodate change is implied. The alternative, when you have to attack a monolithic code edifice with a small pickaxe in twenty seconds flat, is nowhere near as reasonable.

What can we do about this?

Now that we've identified some of the problems of working with an evolving code base, how do we manage the mess? What strategies can we adopt to avoid some of this?

The first and most important thing is to have recognised the problem. Too many programmers hack away without thinking about what they're doing to the quality of their code. As long as they silence the users' screams in the shortest time possible they don't care what state they leave the code in. Someone else can deal with it next time.

Writing new code

Before we think about working with existing code, there are a few considerations for the creation of new code that will greatly aid later maintenance. Extension and malleability need to be designed in, but as we've seen, not at the expense of complexity. Modern component/object based paradigms promise greater reuse and extensibility. They do give us clear interface points between code modules. However, if the interfaces don't support later extensions then something has to budge. Think very carefully about your system interfaces as you create them. It's hard to design for change, so don't necessarily try to support kitchen-sink functionality.

Simple things like writing neat, clear code that can easily be understood and worked with, accompanied by good documentation and well-defined and clearly documented APIs are key. Consider using literate programming tools to document interfaces. This increases the chance that the documentation will be updated when the programmatic interface evolves.

Modularity and information hiding are the cornerstones of modern software engineering. Try to isolate any likely changes to a small part of the system, making your system more viscous and therefore stable under change.

Consider the interconnection of modules, and try not to make every module link to every other module. Inter-module dependencies can take several forms: making function calls, getting notifications, using header files, opening network connections, and so on. It's advisable to avoid having one central module that every other module depends on, since a single change there will affect every other module in the system.

KISS: Keep It Simple, Stupid. Don't over-complicate, don't over-engineer. Optimise an algorithm only when you *know* that there are performance issues, not just because you think you know a good way to make it run faster. Simplicity is nearly always more desirable than performance, and it certainly makes later maintenance easier.

Maintenance of existing code

There is a difference between maintaining good code and maintaining bad code. With the former, you must carefully preserve the integrity of the

4 Of course this is a gross generalisation, but not too inaccurate. Many managers used to be programmers themselves, and know the tensions they must balance. A good manager should listen to the programmers' objections. A good programmer will make their boss listen. Too often, neither happens. Software suffers..

design and ensure you don't introduce anything that is out of place with the system as a whole. With the latter you have to do your best not to make the mess any worse, and if at all possible try to improve things on your way through. If you can't get as far as a rewrite of the offending sections, a little refactoring can go a long way.

Before you even begin to touch any code, a couple of organisational issues should be considered:

- Prioritise any changes that are needed. Balance importance against complexity of work and decide what should get done first. What early changes will impact later work?
- Only change what's necessary. *If it ain't broke, don't fix it.* Don't gratuitously 'improve' bits of code because you think they need it – only make the changes that are really required.
- Keep an eye on how many modifications are being made at once. Certainly making several parallel modifications *yourself* is either incredibly clever or completely foolish, most likely the latter. Do one thing at a time. Carefully. If several people are working on the code at once, be aware of what's changing. There is a danger of too many separate hacks causing odd conflicts. Methodical change by a single developer gives visibility of where the code is being stretched, and where most care is needed. Several simultaneous modifications might make the code pull thin without anyone understanding or noticing.
- Just as the initial code should be reviewed during its development, subsequent changes should also be reviewed. Organise formal reviews, and try to include the original code reviewers. It's very easy to introduce subtle new bugs with small code extensions; reviews will prevent many of these kinds of error.

As well as guarding against the warning signs we saw in a previous section, here are some practical suggestions to help you when working with existing code:

- When you come to make modifications, quickly inspect the code to get a feel for its quality. This is surprisingly easy to do, and you can rapidly get a feel for how easy the code is going to be to work with. Collate all the documentation so you know what's available, then start to digest it. You may find it helpful to use tools to visualise the code. A picture conveys a thousand words, and perhaps several thousands of code statements. Use metrics to gauge the quality of the code. This will make you wary of the places that hidden gotchas could be lurking.
- If you are fixing a fault, do you really understand the cause? If you can write test code to trigger it then this will prove that you have made the fix and help you verify it in all conditions. Once you have made a successful fix, look around the related parts of the system for similar faults. This overlooked step can make a big difference. Many problems hang around in packs, and it's much easier to defeat them in one crushing blow than slowly chip away at them as they each manifest themselves.
- When maintaining any code, retain the programming style of the source files you are working with, even it it's not your style or the house style. A file with code in several formats is confusing and hard to work with. Apply tidy ups as you go if they're not too gratuitous, but be aware that source code diffs across versions will be much harder if you do so.
- Before you modify a particular file or module of code there are a few key things to understand: where the code sits in the whole system, what interdependencies it has (so you know what other components might be affected by a change), what assumptions were made when the code was created (hopefully documented in the various related specifications), and the history of modifications that have already been made.
- Use tests to check you've not broken anything. Exhaustive regression testing is the only real way to be secure about the changes you've made. This is a key point, and often carelessly overlooked. Ensure you have an adequate test suite, and run it regularly.
- Adopt the correct attitude. Avoid that 'just one more hack' mentality. Don't dismiss code, thinking that in the future it will be thrown away or rewritten. I've been there. It won't. No, don't argue: it won't.
- Learn when you need to do some redesign work. Don't be afraid to redo something if necessary. For 'legacy' code this may be considered uneconomical. Sadly, it's legacy code that makes cash and is unlikely to be phased out.
- Try not to introduce extra dependencies with newly added code. An increase in coupling makes code more complicated and harder to change.
- If you make a wrong change, back it out quickly. Don't litter code with unnecessary dead wood.

As professional programmers, we should naturally shy away from the pressure to do a quick bodge job. Sadly we don't work in ivory towers, and sometimes compromise is required; it's not always a commercial reality to complete a task in the theologically 'correct' way. It's unprofessional to flatly refuse to extend code in a distasteful manner. My experience with the TV UI is a good example of why. This explains why so much code is brittle, flaky and dangerous. But it also explains why there's any code out there at all. If there wasn't the commercial drive to get it shipped, programmers would spend forever tweaking code to get it just right, writing and rewriting, by which time the company would have totally collapsed around them.

Conclusion

I'm not sure whether I agree with Aristotle. Change can be a right pain in the rear end. We should manage code changes carefully. That way we can evolve our good programs into something greater, rather than degrade them into an unstable mess.

Perhaps not understanding how to maintain software well and expand it correctly is one of the reasons software development is not yet a true engineering discipline like mechanical or structural engineering.

Pete Goodliffe

References

[1] Aristotle (384-322 B.C.). *Rhetoric.* 350 B.C. Book 1, Chapter 11, Section 20.

[2] Barry Boehm. "Software Engineering." In: *IEE Transactions on Computers*, Volume C-25, No 12. Pages: 1226-1241. Available from: `http://www.computer.org/tc/`

[3] Frederick P. Brookes. *The Mythical Man Month, Anniversary Edition.* Addison-Wesley, 1995. ISBN: 0201835959.

[4] Pete Goodliffe. "Professionalism in programming #17: The code that Jack built." *C Vu, Volume 14, No 6.* ISSN: 1354-3164.

[5] Martin Fowler, et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN: 0201485672.

Intrusive Identifiers

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

I just came across a potentially nasty subtlety in the GNU macro preprocessor M4 (version 1.4).

The words "`format`", "`divert`" and "`shift`" are all recognised as built-in macros, even without any parameters. So if your text happens to contain "`format`", "`divert`" or "`shift`", these words will disappear and strange things can happen.

You can of course work around the problem by undefining them if you don't want to use them:

```
undefine('format')
undefine('divert')
undefine('shift')
```

but I really think that there should be a big, obvious warning in the documentation (at the very least), and I've filed a bug report accordingly.

Lessons for more general programming? Be careful when putting identifiers into somebody else's namespace, especially when they're not likely to know all the identifiers you use when they write their code or text. This is particularly the case with C and C++ macros, which are substituted "blindly" without regard to scope or syntax.

Silas S Brown

Begin C++ Discussions at ACCU Spring Conference 2003

Paul Grenyer

The Begin C++ project [1] is by far the most popular of the ACCU Mentored Developers [2] projects. Unfortunately it has a very high rate of attrition. The project was created to provide support for ACCU members seeking to learn C++ through a mentored team based process. It is aimed at both novice to intermediate developers (students) and more experienced developers (mentors). The Begin C++ project is based around the book Accelerated C++ [3] by Andrew Koenig and Barbara Moo. As well as being a book of outstanding quality, the chapters and exercises also provide a clear structure for the project. The students read through the book at more or less their own pace and are able to post any questions they have to the project list [4] or direct to their mentors. As they complete each chapter the students are encouraged to submit answers to the end of chapter exercises to their mentors.

Many of the mentors have commented that most of their students start off well with regular communication and submission of exercises, but then disappear around chapter 3 or chapter 4. During the 2003 ACCU Spring Conference a number of 'panels' were held, which discussed Mentored Developers related subjects such as distance learning and mentoring. There were also two 'bird of a feather' (BoF) meetings held where the Begin C++ project specifically was discussed.

There were a number of specific ideas and suggestions for the Begin C++ project which came out of the discussions at the BoF's. They mostly originated from Alan Lenton and Mark Easterbrook.

- **More upfront**
  It was generally felt that mentors do not have a good enough idea of the goals of their students or their reasons for taking part in the project. To address this it was suggested that before accepting students into the project they should be asked to explain their background and interest in learning C++. Establishing the goal of the student could help the mentor to provide direction and help the student to attain the goal. Also, if the goal is achieved before the book is completed, the student would not appear to disappear as both student and mentor would be aware.
- **Prerequisites**
  Currently there are no prerequisites for joining the Begin C++ project other than being an ACCU member and owning a copy of Accelerated C++. It was suggested that some relatively simple prerequisites could be introduced. These would be along the lines of installing and learning to use a compiler (simple set-up instructions for major platforms and major compilers would of course be made available) and completing the first couple of Accelerated C++ exercises. This would make sure that the student had the ability to get to grips with a real compiler and would be able to test their exercise answers (the inaccuracies in some exercise submissions have suggested that students haven't tried to compile their solutions). The initial exercise solutions would allow the mentor to gauge the student's natural ability. Some students may need to start at a lower level than that presented by Accelerated C++. Other students may need to be encouraged to work through it quickly and then move on to more advanced topics.
- **Regular IRC Meetings**
  It was suggested that groups of students and mentors should get involved in regular (weekly or monthly was suggested) interactive meetings via the internet using something like IRC (Internet Relay Chat [6]). The meetings could concentrate on a predetermined C++ topic or give students the opportunity to raise problems they are having and get real-time answers. Mentors would share the responsibility of chairing meetings in rotation reducing the individual commitment. All students and mentors would be free and encouraged to attend as many meetings as they wished.
- **Anonymous Posting & Mentor Reposting**
  Many students are afraid to post questions they have to the Begin C++ list in case they are ridiculed by other members of the list. No amount of explaining that this will not happen, as everyone is there to learn, seems to change their mind or increase the overall volume of posting. The posting of questions to the list is very important as this helps other students learn. A couple of suggestions have been made to try and tackle this. One is the ability for anyone to post anonymously. Alternatively a student could first post to their mentor and then the mentor could post the question to the list on the student's behalf.
- **Monthly Begin C++ Newsletter**
  It was suggested that a monthly Begin C++ newsletter would help to increase communication within the project. The newsletter would probably be in email form and consist of descriptions of student's experiences and progress as well as some worked exercise examples. New members of the project could also be introduced.

I attended the panel discussion on Distance Learning. The panel itself consisted of Francis Glassborrow and Alan Lenton. The purpose of the panel in such a session is to encourage the audience to get involved. The was a lot of very interesting debate and a great deal of audience participation. Although the subject of discussion was not specifically about the Begin C++ project or even about the Mentored Developers there was a lot of specific discussion about both. The following points were identified as needing improvement:

- **Feedback**
  It has been suggested that a 'Contract' should be drawn up between students and mentors so that they both know what is expected of them. This contract would include the minimum required communication between the mentor and the student. For example the student might be asked to send an email once a week to their mentor just to let them know they are still alive and still interested in the project. If the email is not received for a few weeks the mentor should contact the student. If no reply is received the student should be assumed to have stopped participating in the project and the mentor seek a new student. This should increase feedback.
- **Motivation & Reward**
  It is often difficult to motivate someone, especially from a distance. Hopefully the goals identified during the student and mentor's initial discussions can go a significant way towards providing motivation. However, this is certainly an area that needs more looking into.
  It has been suggested that a reward system should be introduced to the Begin C++ project. This has ranged from providing a certificate of completion to the ACCU refunding the cost of Accelerated C++. I think these are both good suggestions, but the refund of the cost of the book should carry the proviso that the ex-student become a mentor for the next batch of students.
- **A Sense of Community**
  Hopefully a sense of community will develop as communication increases on the Begin C++ list. Regular IRC meetings, as mentioned above, should help people learn who each other is.
- **Structure**
  I always thought that Accelerated C++ gave the Begin C++ project all the structure it should need. However, this appears not to be the case. This is an area that needs further investigation and discussion.

I believe we now have a better idea of what is wrong with the Begin C++ project and why people are dropping out. I also believe that using the methods suggested above we can improve the Begin C++ group. Luckily all the suggestions can be implemented concurrently, retained if they work and easily dropped and replaced by others if they fail. However, there is still a long way to go...

I have made this article available on my website [7] and I will be encouraging the Mentored Developers and specifically the members of the Begin C++ project to read it and discuss the way forward on list. I would be interested in hearing anybody's comments and volunteers willing to get involved will be greatly appreciated.

Paul Grenyer

References

[1] Begin C++ project:
`http://www.accu.org/begincpp/public/`
[2] ACCU Mentored Developers: `http://www.pjgrenyer.fsnet.co.uk/accu/mdevelopers/`
[3] Accelerated C++: `http://www.acceleratedcpp.com/`
[4] begin-cpp list:
`http://www.accu.org/mail/public/lists.htm`
[5] ACCU Spring Conference:
`http://www.accuconference.co.uk`
[6] Internet Relay Chat (IRC): `http://www.irc.org/`
[7] This article: `http://www.paulgrenyer.co.uk/articles/begin_cpp_discussions2003.htm`

Learning C++: A Student's Perspective

James Gale

Since leaving university I have been presented with some interesting challenges: up to now the most rewarding was embedded C. I have been on a steep learning curve since starting that project and I very much wanted to maintain it. Before starting my latest project my C++ was at best limited to C++ style C and maybe using a few MFC classes scattered here and there.

I had expressed an interest in learning C++ to my colleagues. With some advice I decided I would make a start in my spare time. However I got a lucky break when I was given a C++ project. So I started with the advice given to me by my company mentor which was to read "Accelerated C++: Practical Programming by Example" by Koenig and Moo. My mentor also advised me to purchase other books that included "The C++ Programming Language" by Bjarne Stroustrup and the "Effective" series by Scott Meyers.

Before I had ever really gotten into C++ I was under the illusion that C++ was a difficult language which would take a great amount of time to learn. However as soon as I started to read Koenig & Moo these ideas where shattered. I had always believed C++ was an extension of C. I was simply wrong! I have since come to realise that C++, although based on C, is a language in its own right.

My first great revelation came as I read about vectors. I realised that this was a language which gave the flexibility I sometimes wanted or needed by encapsulating everything in a set of standard classes. Indeed I saw that C++ was a very good language.

So how did I do? Well let's start with the standard library, I had no idea that the standard library even existed let along the containers and algorithms that it contained. I found everything simply fitted in to place — although sometimes with a bit of a push! I was learning more about the language and was really getting to grips with it. However I had only copied a few of the examples from the book. I had yet to write any code of my own. One of the greatest things that the Koenig & Moo book gave me was the confidence to write the code which I was to produce.

I had written some classes before this project but never before with the kind of understanding I was starting to develop. The first code I was going to write was some classes which would be the main base for the first part of the software. I used many of the new containers and techniques I had learned. Indeed things were going very well and there were parts of the language I was really starting to like, such as variable references.

My first real challenge came when I wanted to remove elements from a list. My first attempt ended with me using an iterator in a `for` loop and removing the elements that matched the element which I wanted to remove. The only problem with this is that the iterator would then be incremented after the remove and meant that I missed elements. The answer was to use the list container's version of `remove_if`. My mentor told me I needed to write a functor predicate class. My response was quite clear. And one of them is...?

It was time to make my first use of Stroustrup's book. I looked up the information in the index and then I began to understand. After reading the book I wrote the class. It worked and I was very pleased with myself. Well since then I have used this feature a few times. I was also very impressed with the book as well.

So how did I come to join ACCU? Well that was at the recommendation of my mentor who showed me the site and lent me some of the magazines. I had a look through them and was impressed. I quickly realised that there are plenty of books on C++ and some of them are rubbish, but ACCU did book reviews so I could weed out the rubbish and pick the roses. In addition to this there were articles from which in time I could start learning more.

I wanted to write this article to show anyone like me, who happens to know someone with a copy of C Vu that beginning on the path to C++ enlightenment is not as difficult as it may first seem. Indeed with the right tools the path can be made easier.

So now I am 2 months in to C++ and I still have a lot to learn about the language and the use of the standard library. I have joined the accu-mentored-developers project "Begin C++". The aim of this was to concrete my current knowledge of the language and to complete the exercises in the book itself. Once that is finished then I move on to the Effective books and in time more books. For now the road is long, but instead of walking I now have a car.

I will leave you with a final thought. C++ is a full and rich language which can be challenging and rewarding but most of all the best reward is starting down the path of learning the language and although that path never ends it will be a fantastic voyage.

James Gale

Members' experiences

TCC (Tiny C Compiler)

Silas S Brown

This is a very fast, small and new C compiler for Unix (e.g. Linux), available from `http://fabrice.bellard.free.fr/tcc/`. It compiles code much more quickly than the GNU compiler, and it can be used as a script interpreter with C as the scripting language (using the Unix idiom of starting scripts with "`#!`"). Because TCC compiles very rapidly, there is hardly any compiler overhead for small programs, and the ability to directly execute the source file without having to compile it first can be surprisingly convenient.

TCC supports modern C (for example, you can declare variables just before they are used, rather than having to put the declarations at the start of the block); using it to hack out scripts is easier than one might imagine, especially for people who have sometimes used C++ compilers in C-like ways just to avoid having to do such things as forward declarations. The advantages and disadvantages of C as a scripting language can of course be debated (there are higher level scripting languages, such as Python, which have their own advantages), but if you have a small task that you know exactly how to do in C, then it might be quicker just to write the C (as long as it doesn't get too messy or unsafe).

Of course, TCC can also be used to compile larger projects, in which case it will save a lot of compilation time. Occasionally it will expose a bug in your Unix distribution, as sometimes the provided standard libraries have only been thoroughly tested with the provided compiler (such as GCC); if you use another compiler then you might trip things up. For example, I found (and reported) that GNU `libc6` is missing the `alloca()` function; the GNU compiler replaces it with a built-in function. The header file does say that for non-GNU compilers `alloca()` is defined externally, but it wasn't included in my version of the library.

I wouldn't yet want to use TCC to compile such things as the Linux kernel; I'd rather use the same compiler that the kernel developers themselves use, just in case.

I also tried LWC, `http://students.ceid.upatras.gr/~sxanth/lwc/`, which compiles a subset of C++ into C (and is linked to from the TCC website). The version of LWC at the time of writing is 0.5, and it couldn't yet compile any of my C++ code.

Silas S Brown

Prizes! Prizes! Prizes!

This year the ACCU is offering awards for published articles in a number of categories. These are:

- Best C Vu article
- Best Overload article
- Best article by a new writer

The awards will be announced at next year's AGM and the prize will be an ACCU T-shirt and untold acclaim. Never written an article before? Get going!

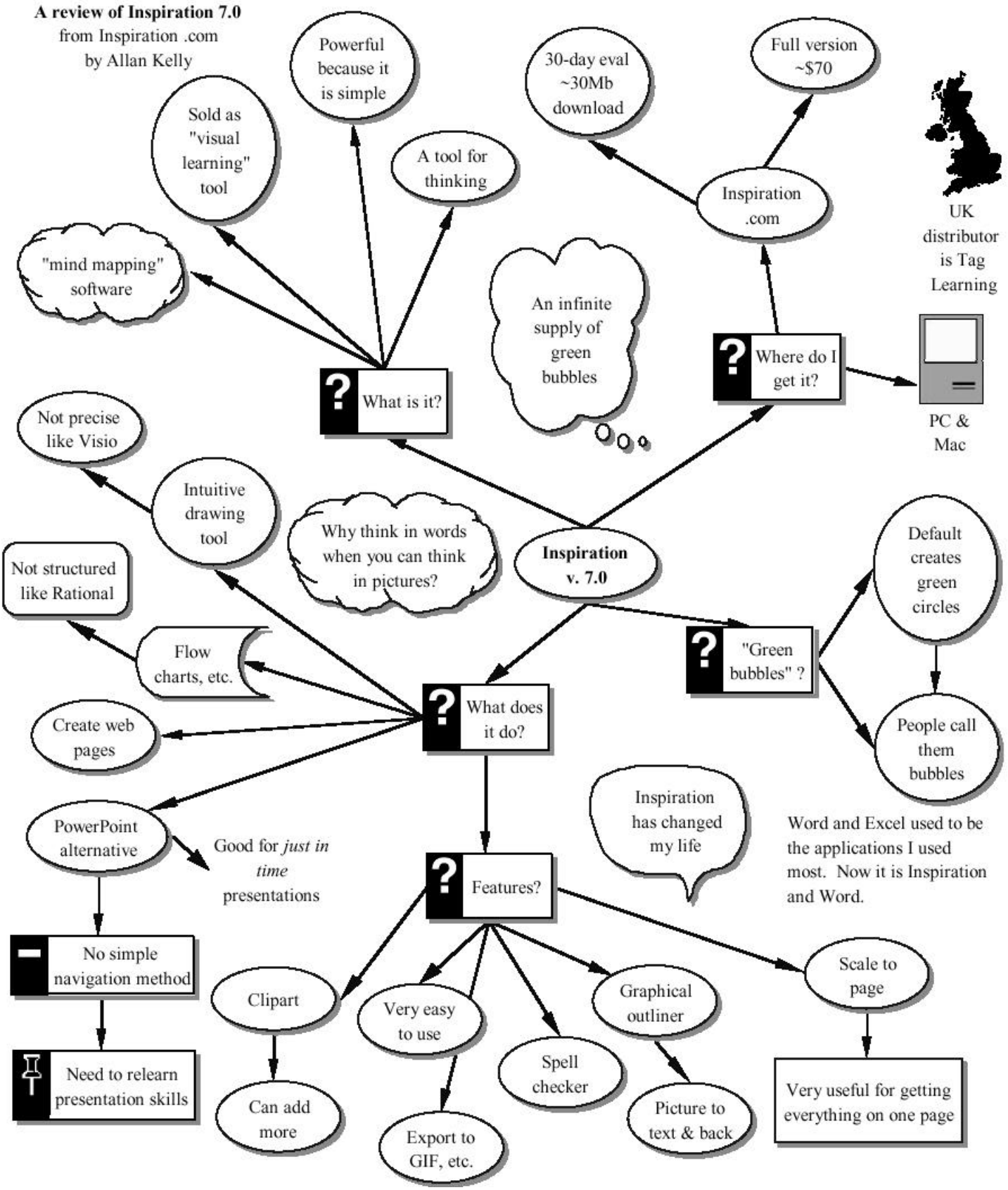
How to submit

You can send submissions by email to `editor@accu.org`. Plain text is perfectly acceptable; there is a Word document template you may wish to use if you want to retain formatting. That's all there is to it – get writing.

Members' experiences

Inspiration 7.0

Allan Kelly



Quick Guide to MinGW (GCC for Windows)

Paul Grenyer

What is MinGW?

According to the MinGW website [1], MinGW is a collection of freely available and freely distributable Windows specific header files and import libraries combined with GNU toolsets that allow one to produce native Windows programs that do not rely on any 3rd-party DLLs. Unlike other ports of GCC to Windows, the runtime libraries are not distributed using Gnu's General Public License (GPL). The full license for MinGW can be found on the License page [2] of the website.

So, in simple terms MinGW is GCC [3] for Windows complete with a Windows API.

Getting and Installing MinGW

Full information about the various components of MinGW and links to the packages are available from the MinGW download page [4]. There are a number of different components, but only one is needed to install a basic MinGW and get started. The current version of MinGW (at the time of writing) is Version 2.0.0, although it is actually version 3.2 of GCC. The Windows installation package is called `MinGW-2.0.0-3.exe`. Download this file, run it and follow the instructions to install MinGW.

To use MinGW from the command line, the path to the MinGW BIN folder must be added to Windows Path environment variable. To do this on Windows 2000 go to the Control Panel and double click the System Icon. Then click the Advanced tab and click the Environment Variables button. Select 'path' in the User Variables for ...' list box and click Edit. This brings up the 'Edit User Variable' dialog box.

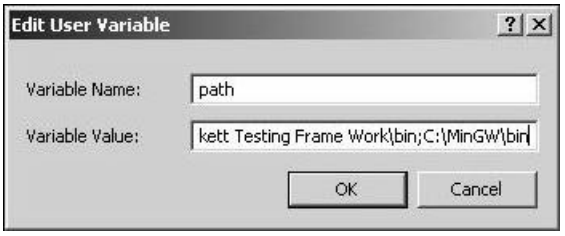


Figure 1: The Edit User Variable dialog box

Move the cursor to the end of the 'Variable Value' edit box, type a semicolon and add the path the the MinGW bin directory. The default path is `C:\MinGW\bin`. Then click Ok. To test MinGW open a command prompt, type the following and press return:

```
g++ –version
```

This should give something similar to the following output:

```
g++ (GCC) 3.2 (mingw special 20020817-1)
Copyright (C) 2002 Free Software Foundation,
Inc.
This is free software; see the source for
copying conditions. There is NO
warranty; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.
```

Hello World with MinGW

The classic Hello World program is very easy with MinGW. Simply create a file called something like `HelloWorld.cpp` and write the Standard C++ Hello World program into it:

```
// HelloWorld.cpp
#include <iostream>
#include <cstdlib>

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

To compile the Hello World program, type the following at the command line from the directory which contains `HelloWorld.cpp`:

```
g++ -o HelloWorld HelloWorld.cpp
```

This should create an executable called `HelloWorld.exe`. Running the executable should display the "Hello, World!" message.

Simple Make Files with MinGW

MinGW has a modified version of GNU Make [5] called `mingw32-make.exe`. The MinGW port of Make operates more as Make was intended to operate and gives less headaches during execution than the 'native' MSVCRT dependent version. For more detailed information on using Make see the GNU Make Manual [6].

To create a simple make file for the Hello World example, create a file called `makefile`, and enter the following rules:

```
# HelloWorld Makefile
HelloWorld : HelloWorld.o
  g++ -o HelloWorld.exe HelloWorld.o
HelloWorld.o : HelloWorld.cpp
  g++ -c HelloWorld.cpp
clean:
  del *.exe *.o
```

To build `HelloWorld.exe` executable enter the following at the command line and press return:

```
mingw32-make
```

To clean the project, or in other words remove the object files and executables, type the following at the command line and press return:

```
mingw32-make clean
```

Of course a make file for a single source file program such as Hello World is overkill. The above example is intended to be a starting point for larger projects.

The source code and make file [7] used in this article is available to download from my website.

Paul Grenyer

References

[1] MinGW Website:
`http://www.mingw.org`
[2] MinGW License:
`http://www.mingw.org/licensing.shtml`
[3] GNU GCC:
`http://gcc.gnu.org/`
[4] MinGW Download Page:
`http://www.mingw.org/download.shtml`
[5] GNU Make:
`http://www.gnu.org/software/make/make.html`
[6] GNU Make Manual:
`http://www.gnu.org/manual/make/html_node/make_toc.html`
[7] Source code and make file:
`http://www.paulgrenyer.co.uk/download/Quick_Guide_MinGW.zip`

Thanks

Phil Nash.

Python Section

A Python project (2)

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

This article continues my description of the Python code in a small project designed to help increase one's vocabulary in a foreign language by playing audio samples.

Some readers might have noticed that the `__init__` method (i.e. the constructor) of the class `Lesson` set two variables `self.newWords` and `self.oldWords` which were not explained in the text. This was an oversight on my part due to having rushed the article to publication (to fill a gap). The original code kept track of the number of new and old words added to each lesson so that it could give the user a textual report, but in the interests of brevity I deleted this functionality (among other things) when writing the article. But I overlooked the initialisation of those two variables, which were now no longer needed.

To continue from where we left off last time, we need to construct some `GluedEventLists` for the `Lesson`. Each list will correspond to the revision of one word at increasing intervals as previously described; each time the word is revised, it will be prompted for and then repeated one or more times, with appropriate delay to allow the user to anticipate the response. (In other words, the computer should play recordings such as "please say", word in English, pause, word in Chinese; if the word is relatively new then this might be repeated.) These clusters of "anticipation events" will be separated with "glue" so that the whole sequence can be added to the `Lesson` and interspersed with other sequences (but not interspersed in such a way that the individual anticipation events are interfered with).

Here is a function that returns the name of a randomly-chosen prompt (such as "please say"), along with an integer specifying whether the prompt should preceed (`0`) or follow (`1`) the first-language word. The function is hard-coded so that the first time a word is introduced it will say "listen and repeat" and the second time it says "say again"; subsequent occurrences are chosen at random from the rest of the list.

```
  def randomInstruction(numTimesBefore):
      if not numTimesBefore: return ("repeatAfterMe.wav",0)
      if numTimesBefore==1: return ("sayAgain.wav",1)
      return random.choice([
          ("whatSay.wav",0),
          ("pleaseSay.wav",1),
          ("nowPleaseSay.wav",1),
          ("tryToSay.wav",1),
          ])
```

Now for a function that returns a single item in an "anticipation sequence", given the first-language word (`promptFile`) and second-language word (`zhFile`) and the number of previous repetitions of that word. The result is returned as a `CompositeEvent` made up of all appropriate recordings and pauses; such `CompositeEvent`s will not be interleaved with others, but the `Glue` that intersperses them will be. Note that the variable `numTimesBefore` means the number of times that the word has ever been repeated in the entire "course" (not just the current `Lesson`); the counts persist between sessions. The hard-coding of number of repetitions is arbitrary and experimental (you can sometimes get away with that sort of thing in Python scripts).

```
  def anticipation(promptFile,zhFile,numTimesBefore=0):
      instruction, instrIsPrefix = randomInstruction(numTimesBefore)
      instruction = promptsDirectory+os.sep+instruction
      zhFile=samplesDirectory+os.sep+zhFile
      promptFile=samplesDirectory+os.sep+promptFile
      secondPause = 1+WavEvent(zhFile).length
      if not numTimesBefore: anticipatePause = 1
      else: anticipatePause = secondPause
      if numTimesBefore == 1: numRepeat = 3
      elif numTimesBefore < 5: numRepeat = 2
      elif numTimesBefore < 10: numRepeat=random.choice([1,2])
      else: numRepeat = 1
      pauseAfter = random.choice([1,2,3])
      # Now ready to go
      list = []
      if instrIsPrefix: list.append(WavEvent(instruction))
      list.append(WavEvent(promptFile))
      if not instrIsPrefix: list.append(WavEvent(instruction))
      for i in range(numRepeat):
          list.append(Event(anticipatePause))
          list.append(WavEvent(zhFile))
          anticipatePause = secondPause
      list.append(Event(pauseAfter))
      return CompositeEvent(list)
```

Now that this has been defined, we can define another function that builds an "anticipation sequence" from a list of such events separated by `Glue`. In this case the number of previous repetitions will be incremented with each item; the function to construct a list is passed a range of values (start value and end value) for this, and will generate the list with an appropriate number of items to fill this range. Thus we can use the function to continue where a previous lesson left off. Again, a certain amount of guesswork is involved with the numbers; I've found these work reasonably well if you are already familiar with the second language.

```
  def anticipationSequence(promptFile,zhFile,start,to):
      sequence = []
      sequence.append(GluedEvent(initialGlue(),anticipation(promptFile,zhFile,start)))
      for i in range(start+1,to):
          sequence.append(GluedEvent(glueBefore(i),anticipation(promptFile,zhFile,i)))
      return sequence
```

```
  def glueBefore(num):
      if num==0: return initialGlue()
      elif num==1: return Glue(15,15)
      elif num==2: return Glue(45,15)
      elif num==3: return Glue(130,30)
      elif num==4: return Glue(500,60)
      else: return Glue(500,150+3*(num-5))
```

`initialGlue()` will return some glue that is arbitrarily stretchable, to separate the first event from the beginning of the lesson (since the word can be introduced anywhere in the lesson). Other calls to `Glue`'s constructor give it the ideal length of glue and the stretchability (the "+/-" of the initial length) as arguments. Here is the implementation of `initialGlue()`:

```
  def initialGlue(): return Glue(0,maxLenOfLesson)
```

Now we will have a class `ProgressDatabase` that keeps track of the user's "progress" (the total number of times each word has been repeated in previous sessions) and generates new lessons accordingly. Its main member data is a list of words and repetitions; a list is used rather than a dictionary because that way we can sort it as required. The list can easily be saved and loaded to a text file in Python syntax by using Python's provided functionality:

```
  class ProgressDatabase:
      def __init__(self):
          self.data = []
          try:
              f = open(progressFile)
              self.data = eval(f.read())
              f.close()
          except IOError: pass
          except SyntaxError: pass # maybe /dev/null
          mergeProgress(self.data,scanSamples())
      def save(self):
          f = open(progressFile,'w')
          f.write(progressFileHeader)
          pprint.PrettyPrinter(indent=2,width=60,stream=f).pprint(self.data)
          f.close()
```

Once the data has been loaded, it is merged with the result of `scanSamples`, a function to scan a sound-samples directory for matching files in the first and second language. This means that new words can be added to the vocabulary merely by putting them in the right directory, without having to take any special action to tell the program about them. Here is an implementation of `scanSamples`, along with a companion function `isDirectory` that tries to determine in a platform-independent way whether or not a particular file is a directory, which is used for recursing subdirectories.

```
  def isDirectory(directory):
      oldDir = os.getcwd()
      try:
          os.chdir(directory)
          ret = 1
      except OSError:
          ret = 0
      os.chdir(oldDir)
      return ret

  def scanSamples(directory=samplesDirectory):
      retVal = []
      ls = os.listdir(directory)
      firstLangSuffix = "_"+firstLanguage+"."
      secLangSuffix = "_"+secondLanguage+"."
      for file in ls:
          if isDirectory(directory+os.sep+file):
              for i,j,k in scanSamples(directory+os.sep+file):
                  retVal.append((i,file+os.sep+j,file+os.sep+k))
          elif file.find(firstLangSuffix)>=0:
              file2 = file.replace(firstLangSuffix,secLangSuffix)
              if file2 in ls:
                  retVal.append((0,file,file2))
      return retVal
```

The `mergeProgress` function merges a progress database with a samples scan, to pick up any new samples that were added since last time the program saved its state:

```
  def mergeProgress(progList,scan):
      for (_,j,k) in scan:
          found=0
          for (i2,j2,k2) in progList:
              if j==j2:
                  found=1
                  break
          if not found: progList.append((0,j,k))
      return progList
```

`ProgressDatabase`'s method to create a `Lesson` involves a little more "scripting" (experimental / arbitrary coding). First it tries to add some recently-learned old words, then new words, then some older words and so on. Each group of words is handled by a service routine that tries to add words according to constraints; for example, each new word should be repeated at least `newWordsTryAtLeast` times (initially,

`newInitialNumToTry` repetitions should be tried; if this cannot be fitted in, one repetition less should be tried and so on down to `newWordsTryAtLeast`). The values of the global variables referred to will be open to tinkering later.

```
  def makeLesson(self):
      self.l = Lesson()
      self.data.sort() ; jitter(self.data)
      self.exclude = {}
      # First priority: Recently-learned old words
      # (But not too many - want room for new words)
      self.addToLesson(1,knownThreshold,1,recentInitialNumToTry,maxReviseBeforeNewWords)
      # Now some new words
      self.addToLesson(0,0,newWordsTryAtLeast,newInitialNumToTry,maxNewWords)
      # Now some more recently-learned old words
      self.addToLesson(1,knownThreshold,1,recentInitialNumToTry,0)
      self.addToLesson(knownThreshold,reallyKnownThreshold,1,recentInitialNumToTry,0)
      # Finally, fill in the gaps with ancient stuff (1 try only of each)
      self.addToLesson(reallyKnownThreshold,-1,1,1,0)
      l = self.l ; del self.l
      assert l.events,"Didn't manage to put anything in the lesson"
      return l
  def addToLesson(self,minTimesDone=0,maxTimesDone=-1,minNumToTry=0, \
                  maxNumToTry=0,maxNumToAdd=0):
      numberAdded = 0
      numToTry = maxNumToTry
      while numToTry >= minNumToTry:
          managed = 0
          for i in range(len(self.data)):
              if maxNumToAdd and numberAdded >= maxNumToAdd: break # too many
              if self.exclude.has_key(i): continue # already had it
              (timesDone,promptFile,zhFile)=self.data[i]
              if timesDone < minTimesDone or (maxTimesDone>=0 and timesDone > maxTimesDone):
  continue # out of range this time
              if timesDone >= knownThreshold: thisNumToTry = min(random.choice([2,3,4]),numToTry)
              else: thisNumToTry = numToTry
              if timesDone >= randomDropThreshold \
                 and random.random() <= calcDropLevel(timesDone):
                  # dropping it at random
                  self.exclude[i] = 1 # pretend we've done it
                  continue
              try:
                  self.l.addSequence(anticipationSequence(promptFile,zhFile,timesDone, \
                                                          timesDone+thisNumToTry))
                  managed = 1
                  numberAdded = numberAdded + 1
                  self.exclude[i] = 1
                  # Keep a count
                  if not timesDone: self.l.newWords=self.l.newWords + 1
                  else: self.l.oldWords=self.l.oldWords+1
                  self.data[i]=(timesDone+thisNumToTry,promptFile,zhFile)
              except StretchedTooFar:
                  pass
              except IOError:
                  # maybe this file isn't accessible at the moment; keep the progress data though
                  self.exclude[i] = 1 # save trouble
          if not managed:
              numToTry = numToTry - 1
          firstPass = 0
      return numberAdded
```

That code referred to a function `calcDropLevel` which calculates the probability that an old word should be completely omitted from a lesson; this increases with the number of previous repetitions of the word, so as to avoid monotony. Here is one possible implementation:

```
  def calcDropLevel(timesDone):
      # assume timesDone > randomDropThreshold
      if timesDone > randomDropThreshold2:
          return randomDropLevel2
      # or linear interpolation between the two thresholds
      return dropLevelK * timesDone + dropLevelC
  try:
      dropLevelK = (randomDropLevel2-randomDropLevel)/(randomDropThreshold2-randomDropThreshold)
      dropLevelC = randomDropLevel-dropLevelK*randomDropThreshold
  except ZeroDivisionError: # thresholds are the same
      dropLevelK = 0
      dropLevelC = randomDropLevel
```

The constants will be defined later. Also there is a function `jitter()` which "jitters" (slightly shuffles) the elements of a list, again to avoid monotony:

```
def jitter(list):
    # Assumes item is a tuple and item[0] might be ==
    # Doesn't touch "new" words (tries==0)
    swappedLast = 0
    for i in range(len(list)-1):
        if list[i][0] and ((list[i][0] == list[i+1][0] and random.choice([1,2])==1) or \
           or (not list[i][0] == list[i+1][0] \
               and random.choice([1,2,3,4,5,6])==1 \
               and not swappedLast)):
            x = list[i]
            del list[i]
            list.insert(i+1,x)
            swappedLast = 1
        else: swappedLast = 0
```

As another method of avoiding monotony, the length of long glue is randomly adjusted before checking for collisions (this was in the previous issue's code but was not explained, sorry).

All that remains, apart from defining the constants, is to write a main program. We'll put it in a function, and only call the function if this Python module is the main module; that way it can also be used as a library module in which case it will not execute its `main()` when imported.

```
def main():
    dbase = ProgressDatabase()
    soFar = dbase.message()
    lesson = dbase.makeLesson()
    firstTime = 1
    while 1:
        lesson.play()
        if firstTime:
            dbase.save()
            firstTime = 0
        if not getYN("Hear this lesson again?"): break
def getYN(msg):
    ans=None
    while not ans=='y' and not ans=='n':
        ans = raw_input("%s (y/n): " % (msg,))
    if ans=='y': return 1
    return 0
if __name__=="__main__":
    main()
```

Of course, in a production release `getYN` and so forth should be more internationalised.

The initialisation of the constants needs to go before `main()` is called; I think it's a good idea to put them near the top of the script for easy access.

```
samplesDirectory = "samples"
promptsDirectory = "prompts"
firstLanguage = "en"
secondLanguage = "zh"
maxLenOfLesson = 30*60 # 30 minutes
maxNewWords = 5
maxReviseBeforeNewWords = 3
newInitialNumToTry = 5
recentInitialNumToTry = 3
newWordsTryAtLeast = 3
knownThreshold = 5
reallyKnownThreshold = 10
randomAdjustmentThreshold = 500
randomDropThreshold = 14
randomDropLevel = 0.67
randomDropThreshold2 = 35
randomDropLevel2 = 0.97
progressFile = "progress.txt"
progressFileHeader = """# -*- mode: python -*-
# Do not add more comments - this file will be overwritten\n"""
```

For convenience, I also put the following code to support overriding the defaults in a different module called `override.py`, if it exists. The code should go before any of the variables are actually used; note that the default values of function and method parameters are evaluated at parse time, so this code should really go before any of the other code (but after the definition of the defaults).

```
try:
    from override import *
except ImportError: pass
```

Finally, to complete the script a few more imports are needed toward the beginning:

```
import time,sched,sndhdr,sys,os,random,math,pprint
if sys.platform.find("win")>=0: import winsound
else: winsound=None
```

Silas S Brown

Practical Uses for Python in C++ Series: Part 2

Generating Lists for C++ in Python

Paul Grenyer

In this, the second article in my Practical Uses for Python in C++ Series, I explore and provide examples of using Python [1] to generate lists for C++. The particular example I am going to use is the loading of country names into a Combo Box (see figure 1). Most of my GUI development experience has been with the Microsoft Visual C++ (MSVC) Graphical User Interface (GUI) libraries. For this article I decided I would try out a new cross-platform GUI framework library, with the idea that it may help the article appeal to more people. I initially looked at Qt [2] but found various problems with it. I then tried WxWindows [3] and found it much easier to work with, after the initial set-up and compilation.

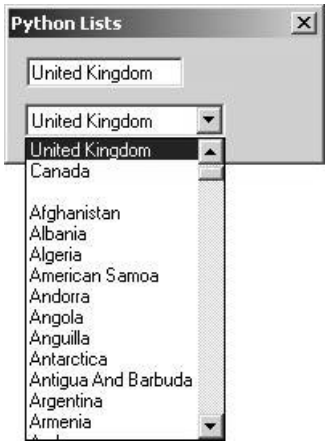


Figure 1: Dialog with Countries Combo Box

At the time of writing the latest stable release of WxWindows is 2.4.0 and can be downloaded from the WxWindows Download [4] page. There are versions available for Windows, Unix (including Linux), MacOS and OS/2. Full installation and build instructions come with each package and are beyond the scope of this article. I would also recommend the wizard [5] for MSVC and Windows users.

Before writing the necessary Python script we need a test application with a Combo Box. This is quite easy with WxWindows and only requires two new classes – an application class and a dialog class. The dialog class holds the combo box and is defined as follows:

```
// PythonListsDialog.h (include guards omitted)
#include "wx/dialog.h"
#include "wx/combobox.h"
#include "wx/textctrl.h"
class PythonListsDialog : public wxDialog {
public:
  PythonListsDialog(wxWindow *parent);
private:
  virtual void InitDialog();
  void OnComboTextChanged(wxCommandEvent &event);
private:
  wxComboBox *m_pCombo;
  wxTextCtrl *m_pText;
private:
  DECLARE_EVENT_TABLE()
};
```

`PythonListsDialog` is implemented in the following way:

```
// PythonListsDialog.cpp
#include "PythonListsDialog.h"
#include "wx/icon.h"
#include "wx/msgdlg.h"
#include <memory>
const wxWindowID comboId = 1000;
const wxWindowID textId = 1001;
BEGIN_EVENT_TABLE(PythonListsDialog, wxDialog)
  EVT_TEXT(comboId,
        PythonListsDialog::OnComboTextChanged)
END_EVENT_TABLE()
PythonListsDialog::PythonListsDialog(wxWindow
                                     *parent) :
      wxDialog(parent, -1, _T("Python Lists"),
      wxDefaultPosition, wxSize(200,100),
      wxDEFAULT_DIALOG_STYLE),
      m_pCombo(0),
      m_pText(0) {}
void PythonListsDialog::InitDialog() {
  try {
    wxString comboContents[] = {""};
    m_pCombo = new wxComboBox(this,
               comboId,
               "",
               wxPoint(10,40),
               wxDefaultSize,
               0,
               comboContents,
               wxCB_DROPDOWN | wxCB_READONLY);
    // Extra check for implementations
    // where new does not throw.
    if (m_pCombo == 0) {
      throw std::bad_alloc("Failed to create
                           combo box: new failed.");
    }
    m_pText = new wxTextCtrl(this,
                      textId,
                      "",
                      wxPoint(10,10),
                      wxDefaultSize,
                      wxTE_READONLY);
    // Extra check for implementations
    // where new does not throw.
    if (m_pText == 0) {
      throw std::bad_alloc("Failed to create
                            text box: new failed.");
    }
  }
  catch(const std::bad_alloc& e) {
    wxMessageBox(e.what(),
         wxString("Dialog Error"),wxOK,this);
  }
}
void PythonListsDialog::OnComboTextChanged(
                      wxCommandEvent &event) {
  m_pText->SetValue(event.GetString());
}
```

The second class is the application class and is defined as follows:

```
// PythonListsApp.h (include guards omitted)
#include "wx/wxprec.h"
class PythonListsApp : public wxApp {
private:
  virtual bool OnInit();
};
DECLARE_APP(PythonListsApp)
```

The application class implements its single override in the following way:

```
// PythonListsApp.cpp
#include "PythonListsApp.h"
#include "PythonListsDialog.h"
IMPLEMENT_APP(PythonListsApp)
bool PythonListsApp::OnInit() {
  // This will be deleted by the framework.
  PythonListsDialog* pDialog =
                            new PythonListsDialog(0);
  pDialog->ShowModal();
  return false;
}
```

Now that we have an application with a Combo Box we need to fill it with country names. The `wxComboBox` method for adding strings to the combo box is `Append()`. Therefore the code to add countries would look something like this:

```
m_pCombo->Append("Afghanistan");
m_pCombo->Append("Albania");
m_pCombo->Append("Algeria");
....
m_pCombo->Append("Zimbabwe");
```

This would take some time to format and enter into the code by hand and if the list ever changed significantly (unlikely for countries, of course) it

would need doing again. This is where Python comes in handy. If you have a list of countries in a text file, Python can be used to generate a function which will load all the countries into the combo box.

Start with a header file that declares the function which will load the countries:

```
// LoadCountries.h
#ifndef LOADCOUNTRIES_H
#define LOADCOUNTRIES_H
#include "wx/combobox.h"
void LoadCountries(wxComboBox* pComboBox);
#endif // LOADCOUNTRIES_H
```

Then create a template for the cpp file. Call it something like '`LoadCountries_template.cpp`':

```
// LoadCountries.cpp
#include "LoadCountries.h"
void LoadCountries(wxComboBox* pComboBox) {
  if (pComboBox != 0) {
    // Countries Here
  }
}
```

The template file will be used to generate C++ along with the text file containing the list of countries. The comment:

```
// Countries Here
```

is used to tell the Python script where to write the countries in the generated C++ file.

Now it's time to write a Python script. Assuming that the list of countries is in a flat text file called '`countries.txt`' and the C++ template is in a file called '`LoadCountries_template.cpp`', the first thing we want the script to do is take the two input files and an output file as command line arguments:

```
# CountriesList.py
import sys
import string
print "\nCountriesList.py\n"
# Check Command Line Parameters
if len(sys.argv) < 4:
  print "CountriesList.py requires three filenames \
                       as command line parameters: "
  "Countries, Template, OutputFile"
else:
  print "Countries Filename:\t" + sys.argv[1]
  print "Template Filename:\t" + sys.argv[2]
  print "Output Filename:\t" + sys.argv[3] + "\n"
```

Then open, read in and store the list of countries and the template:

```
#Open and store Countries
CountriesFile = open(sys.argv[1],"r")
CountriesStore = CountriesFile.readlines()
CountriesFile.close
#Open and store Template
TemplateFile = open(sys.argv[2],"r")
TemplateStore = TemplateFile.readlines()
TemplateFile.close
```

This list of countries I have is all in upper case and each country is followed by its abbreviation. For example:

```
AFGHANISTAN AF
ALBANIA AL
ALGERIA DZ
AMERICAN SAMOA AS
```

Therefore as the Python script writes the C++ it not only needs to be able to spot the "`// Countries Here`" comment, it must also case the countries and remove the abbreviation. Also, each country must be wrapped in quotes (") as well as the `Append` method.

The "`// Countries Here`" comment must be detected during the writing of the output file. This can be done as follows:

```
#Write Output File
OutputFile = open(sys.argv[3],"w")
for templateline in TemplateStore:
  if templateline == "// Countries Here\n":
    # write countries here
    print "Countries Comment Found.\n"
  else:
    OutputFile.write(templateline)
OutputFile.close()
print sys.argv[3] + " written Ok."
```

Finally the countries must be formatted and written to the output file. Python makes this easy:

```
# loop through countries
for line in CountriesStore:
  # Remove country abbreviation
  line = line [:line.rfind(" ")]
  # Case country
  line = string.capwords(line)
  # Write combo box loading country code to
  # output file
  OutputFile.write('\t\tpComboBox->Append("' + \
                                    line + '");\n')
```

The complete Python script looks like this:

```
# CountriesList.py
import sys
import string
print "\nCountriesList.py\n"
# Check Command Line Parameters
if len(sys.argv) < 4:
  print "CountriesList.py requires three filenames \
                       as command line parameters: "
  "Countries, Template, OutputFile"
else:
  print "Countries Filename:\t" + sys.argv[1]
  print "Template Filename:\t" + sys.argv[2]
  print "Output Filename:\t" + sys.argv[3] + "\n"
  #Open and store Countries
  CountriesFile = open(sys.argv[1],"r")
  CountriesStore = CountriesFile.readlines()
  CountriesFile.close
  #Open and store Template
  TemplateFile = open(sys.argv[2],"r")
  TemplateStore = TemplateFile.readlines()
  TemplateFile.close
  #Write Output File
  OutputFile = open(sys.argv[3],"w")
  for templateline in TemplateStore:
    if templateline == "// Countries Here\n":
      # loop through countries
      for line in CountriesStore:
        # Remove country abbreviation
        line = line [:line.rfind(" ")]
        # Case country
        line = string.capwords(line)
        # Write combo box loading country code
        # to output file
        OutputFile.write('\t\tpComboBox->Append("' + \
                                    line + '");\n')
      else:
        OutputFile.write(templateline)
  OutputFile.close()
  print sys.argv[3] + " written Ok."
```

To generate the `LoadCountries.cpp` file type the following at the command line, using the appropriate file paths:

```
python countriesList.py countries.txt
LoadCountries_template.cpp LoadCountries.cpp
```

[concluded at foot of next page]

Reviews

Bookcase

Collated by Michael Minihane
<michaelm@pobox.co.uk>

Francis Glassborow writes:

I have over 250 titles waiting for reviewers. Actually when we consider that we have over 3000 published reviews, that is not bad. However authors, publishers and potential readers all appreciate timely reviews so more hands will lighten the load.

You will also note that there is no C & C++ heading in this Bookcase. That isn't because there are no books being published on that subject but because those we had reviews for in this issue were system specific and so got classified elsewhere.

Many of you know that I try to trap out the really bad books on C and C++ and carry out the burdensome task of reviewing them myself. I could really do with someone who can authoritatively identify bad books on Java and C# and write honest reviews of them. Note that it is rarely necessary to read all (or even most of) a bad book in order to identify it as such, but it is necessary to be very clear in your own mind that it is actually bad rather than one whose style you dislike.

For example, typos damage a book but do not make it a bad one. However technical errors central to the subject of the book are very damaging. We have to distinguish technical errors that are central from those that are incidental. Errors in C++ found in Design Patterns are irritating but they are not central to the subject. Errors in C++ in a book purporting to teach C++ are central and would be a reason for warning readers to keep away from it.

Now to the conundrum of the month. Why does Amazon.com list used books that cost more than the same (in print) titles when bought new? In one case I noticed used copies cost 50% more than new ones.

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`
The PC Bookshop (020 7831 0022)
`orders@pcbooks.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 1.4 or better as appropriate (in a context where the true rate hovers around 1.5). I consider any rate below 1.25 as being sufficiently poor to merit complaint to the publisher.

Java & C#

.NET Development for Java Programmers by Paul Gibbons (1 59059 038 4), Apress, 386pp @ $49.95 (no UK price)
reviewed by Duncan Kimpton

On opening the book I was pleasantly surprised by the layout – plenty of white space and short paragraphs interspersed with screen shots.

The book, in my opinion erroneously, assumes that you will have a copy of VS.Net to hand. That is not at all certain when looking into the possibility of a platform transition.

C# is covered quickly and concisely, this is a great way of learning a new language, but it doesn't go into enough depth.

Winforms is covered lightly, but lacking any meaningful detail, this is largely worthless.

Coverage of web services and ASP.Net is much better, going into considerable depth. This is, however, at the expense of a heavier writing style.

Most aspects of .Net are given some coverage, with the notable exception of file I/O.

Overall Gibbons provides a good overview of the transition from Java to .Net for web developers, but not for application developers.

Don't expect this book to do anything more than get you started on the path to .Net development; the grandiose claim on the back cover that '*By the end of .Net Development for Java Programmers a professional Java developer should be capable of tackling a real world software project in .Net using C#.*' is way over exaggerated. Expect to buy other complementary reference works too.

Unfortunately I am unlikely to return to this book for a second reading. This book is over priced and not recommended.

Java Web Services Programming by Rashim Mogha, V V Preetham (0-7645-4952-9), Wiley, 555pp @ £37-50 (1.33)
reviewed by Emma Willis

Web Services are self-contained modular applications that can be defined, published and accessed across the web.

This book claims to teach you, amongst other things how to 'understand the new web services model', 'design a web service using Java architecture' and 'implement web services using Java's XML APIs'.

It deals exclusively with the Java tools available to the Web Services programmer, as these are handily packaged up into Sun's Java Web Services Developer Pack (Java WSDP).

Then add `LoadCountries.cpp` and `LoadCountries.h` to the WxWindows project. Add "`#include "LoadCountries.h"`" and "`LoadCountries(m_pCombo);`" to `PythonListsDialog.h` and rebuild:

```
// PythonListsDialog.h
#include "PythonListsDialog.h"
#include "wx\icon.h"
#include "wx\msgdlg.h"
#include "LoadCountries.h"
#include <memory>
...
void PythonListsDialog::InitDialog() {
  try {
    wxString comboContents[] = {""};
    m_pCombo = new wxComboBox(this,
             comboId, "",
             wxPoint(10,40),
             wxDefaultSize, 0,
             comboContents,
             wxCB_DROPDOWN | wxCB_READONLY);
    LoadCountries(m_pCombo);
  ...
```

Running the executable created by rebuilding the WxWindows project will display a small dialog box with a Combo Box of countries. Selecting a country from the Combo Box displays its edit box above it (see figure 1). The full `CountriesList.py` script and a Microsoft Visual C++ 6.0 test application[6] are available for download from my website.

Paul Grenyer

References

[1] Python: `http://www.python.org/`
[2] Qt: `http:\\www.trolltech.com/products/qt/`
[3] WxWindows: `http://www.wxwindows.org`
[4] WxWindows Download page:
`http://www.wxwindows.org/downld2.htm`
[5] WxWindows wizard:
`http://www.wxwindows.org/dl_optn.htm`
[6] Full `CountriesList.py` script and a Microsoft Visual C++ 6.0 test application: `http://www.paulgrenyer.co.uk/download/python_lists.zip`

Thanks

Vadim Zeitlin, Patrick K. O'Brien

Parts one and two make a good introduction to the uses for and the architecture of a Web Services implementation. Touching on all elements, 'the web services technology stack', explaining roles of each. Key components of the stack include XML, WSDL, SOAP and UDDI.

Part three looks in more details at each element of the Java WSDP; these include JSPs and Servlets, JAXP, JAXB, JAXM, JAX-RPC, JAXR and JSTL.

At least a page of each chapter is wasted on a largely common introduction. I also found that plenty of the content of each chapter was previously covered in parts one or two and subsequently covered again in the appendices.

Saying that, each of these chapters does serve as a thorough introduction to the technology area concerned – most chapters offering tutorials to work through a given example. However, although references are made back to part two, it's still unclear how each of these Java technologies fit together.

In my experience of Web Services, one rarely needs to understand all of the processes involved; indeed, the authors often refer mysteriously to 'tools' that perform many of the processes for you. A thorough introduction to Web Services programming would make more of these tools – What are they? How do they bring all of these technologies together?

This book starts by providing great inspiration for would-be Web Services programmers, but by the time they've finished reading it, they'll be thoroughly confused.

XML and Java Developing Web Applications 2ed by Yuichi Nakamura et al. (0 201 77004 0), Addison Wesley*, 661pp + CD @ £37-99 (1.32)
reviewed by Emma Willis

I honestly feel this is the finest computing book that I have come across during my career (OK, so I've only been doing this for 2 years, but I feel really strongly about this book!). So many books struggle to cover numerous topics in limited space, so you'd think that relating the entire realm of XML in one book would be an impossible task – think again! The authors of this book do justice to all the common areas of XML usage such as parsing and validation; and some of those less widespread uses, e.g. web services provision, data binding and employment of XML for an application server.

The first half of this book illustrates the basic tools for dealing with XML in Java. Described as the 'absolute minimum' needed to understand and develop web applications, they draw upon solid and stable standards. The latter half of the book covers more exciting opportunities for XML and draws upon emerging technologies such as SOAP.

No areas were skimmed over – in fact, I felt each new concept introduced was accompanied by real-life, understandable and well-documented examples that really work. The accompanying CD contains all of the source code and the latest versions of all of the software that you will need to run the code and to start developing your own enterprise-standard web applications.

One of my worries when I started this book was that with 9 named authors, this book would suffer, as many do, from repetition and lack of coherence. My worries were unsubstantiated; the book benefits greatly from their apparent wide ranges of experience. I found it refreshing to read about XML technologies I may never previously have considered because Java books often concentrate on Sun or Apache technologies and on W3C options when it comes to XML.

I'd recommend this book to anyone who wants to build upon his or her basic knowledge of XML and to explore and apply new concepts to real-world situations.

C# Programmer's Reference by Grant Palmer (1 861006 30 6), WROX*, 555pp @ £25-99 (1.35)
reviewed by Paul F. Johnson

This is actually a difficult book to review. On one hand, it is very well written with all of the information clearly set out, the examples show how to compile the source (something lacking in most books of this type) and above all, the layout of the chapters is logical. On the other, I'm not sure the book knows who it is aimed at.

I'll explain what I mean by that.

In the early days post standard in C++, a lot of books were 'updated' to take into account the changes brought about by the standard. They still remained the same books with a nod to the standard. Some even were C books with `stdio` removed and `iostream` added and they were obvious. The early chapters of this book fall into this type. All of the C++ material is there with the C# bits plugged in and standard examples used. This extends for the first couple of chapters.

After that, the book starts to be a 'true' C# book – with some examples for beginners and some for intermediate users. The majority of the examples compile fine under both Microsoft's Visual Studio and Ximian's Mono packages (those calling the Windows API obviously won't work under Mono – well, not yet anyway).

The book itself starts off being decidedly beginnerish in feel and approach then stops. Then the book becomes a programmer's reference manual, which is what the title describes it as. The classes and methods are given with descriptions of what they do and the parameters they need.

If the book gave the C# API with examples (akin to Josuttis's C++ Standard Library) then it would be an excellent investment. As it stands, for the descriptions and clarity of the methods and classes, it is a good book to have to hand. The chapter on XML is one of the clearest I've read in ages. Recommended.

C# A Programmer's Introduction by Deitel & Deitel et al. (0 13 046132 6), Prentice Hall, 862pp @ $49.99 (no UK price)
reviewed by Paul F. Johnson

This is a fantastic book. It is well planned, well organised with plenty of examples and high quality images so not only can you see the code, but also see the results. Basically, it is what you would expect from a Deitel title.

You can see by the style of writing that the Deitel range has developed from the nigh on impossible to read earlier books to a book which can be used by anyone. Why anyone though? The answer is that the book covers its material in sufficient detail. Where other books (aimed at the same level) assume you already know key concepts (such as OOP), this one doesn't. The classes are well explained.

The code examples are well documented and all aspects of the .NET 1.0 framework are well covered. Just in time really for .NET 1.1 – but given the lead-time of books, that is to be expected.

Unlike other C# books I have reviewed, you definitely need Microsoft Visual C#. Some parts can be compiled under GNU .NET or mono, but not the specific GUI material. There is nothing wrong with this, after all, the book is designed for VS .NET.

There are a couple of very minor drawbacks to the book.

The book is not simple enough to pick up and put down – you start a chapter, you really do need to finish the chapter. There are two completely superfluous chapters at the end on the HTML 4 specification and there is a lack of any form of questions or end of section tests. I personally would have liked to have seen this as books aimed at an introduction level really rely on reinforcement of acquired knowledge and questions or mini-tasks are probably the best way of achieving this.

A CD of the source would have been an advantage, but not essential – the source is available from the Deitel website as are errata and other bonus material. Highly recommended.

Programming Spiders, Bots, and Aggregators in Java by Jeff Heaton (0 7821 4040 8), Sybex, 516pp + CD @ £45-99 (1.30)
reviewed by Mathew Davies

Prior to reading this book, I knew very little about how e-mail works or the mechanics of browsing the World Wide Web. After reading the book, I feel fairly confident that I could build an application to cruise around the Web and gather information on my behalf.

I am no expert on networks; nor am I one of the Java cognoscenti. I was therefore relieved to find that the opening chapters of this book provide a basic introduction to how PCs communicate with remote servers. What's more, the author provides a series of small Java applications to illustrate the process. For example, the book's initial description of SMTP includes a Java application that lets you watch the conversation between your own machine and the server at the other end. Call me childish if you will, but I was hooked.

As the book progresses, so the author focuses in on the HTTP (and HTTPS) protocols, showing how to access, parse and then use the contents of various (e.g. HTML) files distributed around the World Wide Web. This leads naturally to the development of those bots, aggregators and spiders that are mentioned in the book's title. What are they? A bot will collect data from a web site on your behalf; an aggregator will collect, combine and process data for you from a number of different sites; while a spider will carry out an autonomous trawl around the Web on your behalf.

As the book's title suggests, the author uses Java as his language of choice to demonstrate working applications. There are a couple of important points to be made here: first, you don't have to use Java to build bots, aggregators and

spiders; and second, the CD that accompanies the book includes not only the full Java source for the applications in the main text but also the author's complete package for building applications of this type. On this latter point, each chapter of the main text includes a detailed explanation of the corresponding classes from the Java package, while there is a separate appendix dedicated to a description of the package as a whole.

All in all, I consider this to be an informative and well written book. If you're interested in building applications to ferret around the Internet on your behalf, I can wholeheartedly recommend this one; even if, like me, you are not a Java guru!

Games Programming

Focus on 2D in Direct3D by Ernest Pazera (1 931841 10 1), Premier Press*, 270pp + CD @ £21-99 (1.36)
reviewed by Daire Stockdale

The only positive thing I can say about this book is that it doesn't appear to be inaccurate. I found it very difficult to imagine what audience would benefit from reading it. The book is far too limited in its scope, lacking depth, breadth and technical ambition to be of any use to a games programmer, 2D or otherwise; they would be far better served by spending an hour with the relevant DirectX documentation.

The book is written in a verbose first-person vernacular, which I found very irritating. It also contains many useless, poor quality illustrations that cannot but lead me to suspect space filling on the part of the author or editor. Examples include a half page picture of the author's work area (because it is a bitmap loaded by some example code) and a half page illustration of a black screen containing three dots, with the remaining half page taken up with a screenshot of the same three points, but now joined by lines. The illustrations themselves are of such poor quality as to be reminiscent of photocopies and they do not aid the understanding of the accompanying text in any way.

'*2D in Direct3D*' walks the user through enough Direct3D to allow him/her to begin rendering primitives on screen and covers texturing and alpha blending in the shallowest manner possible. However I think that two or three pages of concise, well documented code would be a better learning aid than this book and indeed the DirectX 8.1 SDK (which is needed to compile the author's code) provides reams of such code. I believe that when code is presented to a young and malleable audience (and surely this is the only audience who will suffer this book), it should be of the highest standard, within reason and must promote good habits.

In a book supposedly aimed at '*a game programmer and somewhere above the rank of total newbie*', it is continuously apologetic for having to use any maths and its treatment of the maths topics it does attempt to cover is terrible. Pages 146 to 148 of the book are just an unreadable expansion of the product of two 4x4 matrices and ditto with matrix addition on pages 144-145. The author should have consulted classic entry level texts covering the same topic, such as the OpenGL 'red book' to see how else it can be done. Or simply omitted the topics altogether and assumed some degree of numeracy on the part of his readership.

Incredibly, the author manages to cover lighting without even mentioning DirectX's lighting formula, which relates the scene's lights to a surface's material to the final rasterised pixel.

I was also very disappointed, although not surprised, by the low quality examples that accompany the book on the CD. Chapter 16 describes its accompanying executable as a 'simulation of rain', however it looks simply like flickering blue lines.

In conclusion, this book has nothing to recommend it to anyone with the mildest experience of 2D programming and I fear it may lead the innocent novice into bad habits and wasted time.

AI Techniques for Game Programming by Mat Buckland (1 931841-08-X), Premier Press*, 448pp + CD @ £43-99 (1.36)
reviewed by Daire Stockdale

The title of the book is slightly misleading and I think it would be better titled something like 'An introduction to Neural Networks and Genetic Algorithms'. The reader will not find mentioned many techniques used by most game engines, such as node based path-finding or A*. I would have hoped a thorough title on programming game AI would also mention 'bot' programming, state machines, virtual machines, aides to debugging AI, etc. none of which makes it into this book.

However, as an introductory book to the difficult topic of neural networks and genetic algorithms, I think this book is very useful. The author knows his subject well and explains it helpfully, accompanied by many examples on the book's CD. He codes in clean, well-written and nicely commented C++. This book would definitely be useful to anyone without experience of neural networks and genetic algorithms and who was thinking about using them in an application. These topics are covered in sufficient depth that even those who feel competent in this area might learn a thing or two. I would like to see the author write something a little more serious and definitive, perhaps aimed at the harder-core coding audience.

The book suffers from feeling it must cater even for readers with absolutely no experience of Win32 or 3D math programming by covering topics such as vectors, GDI and the Windows message loop. The first 20% of this book has nothing to do with AI whatsoever. This seems to be a common flaw in all the books I have seen in Premier Press' game development series. It would be far better if they devoted one title to this subject rather then force their customers to repeatedly buy the same material in every book.

Special Effects Game Programming with DirectX by Mason McCuskey (1 931841 06 3), Premier Press*, 913pp + CD @ £43-99 (1.36)
reviewed by Daire Stockdale

I was very disappointed by this book. It follows the same pattern of several books in the André LaMothe 'Game Development' series, by devoting too much of the book to irrelevant material. By the author's own admission almost 55% of this book has nothing whatsoever to do with special effects programming; on page 426 of the book's 913 pages the author says '*One part down, two to go! Now its time to sink your teeth into some actual special effects code.*' The first 426 pages covered introduction to Windows, matrices, vectors, DirectX, etc. It seems that games programming has not yet matured enough to warrant anything like the 'Advanced C++' series. Instead we must be content with books that are written for beginners and therefore assume no prior knowledge about any aspect of the subject.

The remaining chapters of the book cover 2D and 3D effects. Many of these effects are implemented in software, which is fine if the entire purpose of your application is to showcase that effect. However as the book is purporting to be for games programmers, I would question whether any game could afford the luxury of this expensive processing. Of the effects that are covered, nothing is covered in great detail, or with professional applications in mind.

Pixel and vertex shaders, which allow graphics hardware to be programmed, are mentioned, but poorly covered, given the utmost importance of these to modern game engines. They are lumped together into a short final chapter. Many of the special effects the author illustrated could have been written as pixel or vertex programs. This might have made the book more appealing to a professional programmer keen to move into these areas.

The book's appendices then go on to 'cover' C++ and the STL. This is unnecessary and irrelevant in a text on special effects in DirectX and a waste of the buyer's money. References to several good texts are all that is needed.

If you are going to write a book on special effects and claim that it is for professional games programmers, then I think it would be best to address the subject in a slightly more abstract way; you could mention where a particular API allows certain optimisations, or has a useful feature, but there is no particular link between DirectX and special effects. Your code will have to be top notch and you will need to show how it circumvents common problems relevant to its area.

Alternatively, if you are going to write a book on DirectX programming, decide beforehand if you are writing for the professional coder who uses DirectX for a living and who will expect you to reveal tricks and areas not fully documented, or if you are writing for someone new to the subject, who may need to be walked through their first DirectX application.

The series editor clearly disagrees with me on this book however. In the preface he says:

'*This book is without doubt a secret weapon...you will have cutting-edge knowledge of 3D techniques that leverage DirectX and will amaze your friends and get you lots of dates with hot women...I mean no one has EVER done a book like this...killer stuff! ... It has been a pleasure developing this book and reading the chapters late at night while roller-blading on the empty streets...*'

Need I say more?

As a final criticism, the book is already dated; DirectX 9.0 is already old news, with 10 in testing. The author could have written a useful reference text on implementing special effects, which would have aged well. Now however it is destined to lie on my shelf alongside my DirectX 5 and DirectX 7 tomes, gathering dust.

Game Programming All in One by Bruno Miguel Teixeira de Sousa (1 931841 23 3), Prima, 952pp + CD @ £36-99 (1.35)
reviewed by Paul S Usowicz

This book is split into 4 distinct sections consisting of C++ Programming, Windows Programming, Hardcore Games Programming and Appendixes. I shall examine each of these in turn as each of them can really be considered a separate book!

The first section is C++ programming. This, oddly, is an actual tutorial of C++ from no knowledge to fully-fledged programmer in just 276 pages. This clarifies the title excerpt 'All In One'. The aim of the book is to take a complete novice and turn him into a games programmer in just one book with C++ being the first of 3 steps required. Now don't get me wrong, this section is very good with common sense and good code a plenty but feels rather rushed and cannot cover the full C++ language in any detail in such a small space. It also appears to be accurate code (if we let them off a few typos such as the scope resolution operator being shown as ;; in one of the tables).

The second section is Windows programming. This chapter covers basic Windows programming and includes an introduction to DirectX. The Windows programming tutorial only covers 40 pages with the rest of the chapter consumed by DirectX and some musing about the games library enclosed on the CD and used later on in the book. Call me old-fashioned but there is no way you can even begin to appreciate what is involved in Windows programming in just 40 pages. Fortunately the book comes into its own when DirectX is finally reached. This is what I wanted the book for. Basic 2D theory and practice is discussed including surfaces, vertices, bitmaps and primitives. If you are already a good Windows programmer who wants to learn some basic DirectX then page 357 (out of 862 relevant pages) is where you start.

The third section is hardcore games programming. Included in this section are chapters on data structures & algorithms, maths, artificial intelligence and physics modelling. All chapters are very in-depth and very readable. At the end of the section you will have a full DirectX working game based on Breakout. It is only 2D though with 3D being left to other books. The final chapter in this section is titled 'Publishing Your Game'. Although only 11 pages in length it does contain information that I have never seen before in a book. Topics covered include contracts and interviews with actual publishers.

The last section is the appendixes. This small section is where everything else has gone including CD-ROM contents (DirectX 8.1 SDK, various demos and the source code) and answers to the questions asked throughout the book.

So, whom is the book aimed at and should they buy it? If you are an inexperienced programmer I feel that this book will become very difficult about halfway through and you will need other books and experience before you can progress. If you are an experienced programmer you will start halfway through the book and thoroughly enjoy it. I feel that the author should have missed out the C++ tutorials and just written about the games programming, a subject he obviously knows very well. In conclusion, I will be keeping this book but only the latter half will ever be used again.

OS Based Development

Palm Development by Clayton Crooks (0 7897 2649 1), Que, 250pp @ £21-99 (1.36)
reviewed by James Amor

'Palm Development' – in my opinion this title implies that the book will provide an overview of the field of development for palm devices – not so; in actual fact the scope of the book is limited to development for the Palm OS only, neglecting other PDA/Palm operating systems, e.g. Windows CE. It does however meet the needs of its target audience and is very much a beginner's introduction to the field.

The main focus of this book is on proprietary software packages (e.g. AppForge and Satellite Forms) and this is where the main stumbling block lies; the book has no companion CD. Most chapters of the book are targeted at one RAD IDE or another, however after visiting the manufacturers websites (the links to which are provided in the text) it becomes clear that very few are available on a trial basis; obviously expensive IDEs are not an option when first exploring a platform and so many chapters of the book are made immediately useless. Should you be wishing to develop applications purely using the Palm OS C++ SDK then this book is not for you either, as despite acknowledging the fact that most applications are developed in C++ using the SDK, only 16 pages are dedicated to its description.

On the positive side, the book is well illustrated, with multiple screenshots supporting most examples; the examples are simple, enabling the reader to concentrate on understanding the development tool, rather than the application being developed; and the explanations of the implementations are clear and concise. All of these make this title an excellent beginner's book, but little use as a book for the intermediate/advanced developer and very little use as a reference book.

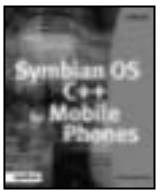
Symbian OS Communications Programming by Michael Jopping (0 470 84430 2), Wiley, 397pp @ £34-95 (1.29)
reviewed by Ralph McArdell

Symbian OS Communications Programming focuses on adding communications services such as Bluetooth, WAP or telephony to Symbian OS applications.

It is written in a clear and accessible manner and gives perspective to the Symbian OS support for communications that you do not get by just reading the SDK documentation. The layout is logical with most communications technologies being covered in enough detail to get you up and running. Copious pointers to further information are provided with the occasional question to make sure you are still awake. Where source code is shown only enough is reproduced to illustrate the point with the full source code for the examples available online.

On the other hand little space is given to Symbian OS programming in general so this would not be the only Symbian OS reference required to develop for this platform and other Symbian OS titles also cover communications, though not necessarily in as much detail. The version of the Symbian OS discussed - version 6.1 - although still very much in use has already been superseded. As a result some topics such as GPRS, which has limited support in version 6.1, do not get detailed coverage.

This book worked well for me as someone new to Symbian OS development. However, due to the review deadline requirements this was the first Symbian title I read. If you are developing for the Symbian platform and need more communications knowledge than your existing references provide then take a look at this book.



Symbian OS C++ for Mobile Phones by Richard Harrison et al. (0-470-85611-4), Wiley, 798pp + CD @ £34-95 (1.57)
review by Francis Glassborow

If you are in a position to develop for the Symbian OS, this book is close to a 'must read' even though the quality of the C++ source code is not always up to my standards. Otherwise this book is too target specific to be worth investing professional time in studying it.
[see website for full review]

Microsoft .NET Framework 1.1 Class Library Reference 1-4 by microsoft (0-7356-1555-1), Microsoft*, 8000pp @ £108-99 (1.38)
review by Francis Glassborow

I am not sure exactly how one is expected to review a reference volume of over eight thousand pages (actually the true total is nearer 12000 because despite what it says on the covers there is a fifth volume – in parts A and B that is not included in this set.) but here is some information that may help potential purchasers (not readers because even those who enjoy reading dictionaries will not want to read these)

First, the trivial information that the four volumes together will exceed the carry on baggage allowance for most airlines and keep you exercised. Joking apart, these are heavy books.

The books take each element of the .NET Framework Class Library and documents it with the following:

A brief statement of what it does followed by a summary of its 'provision' in Visual Basic, C#, C++ and Jscript. Each function has documentation on its parameters and return value as well as any possible exceptions. There follows a brief set of remarks followed by a set of requirements. These are the platforms on which the function will run.

Around ten per cent of the entries include brief sample code.

Very few entries are longer than a page, while less than half a page seems more normal.

I have been very critical of the sheer size of the Java libraries. Their size makes it unlikely that they are uniformly high quality products. It is also hard for a user to have a good overview of what is available. However this framework library is more than twice the size. It is not my job as a book reviewer to comment on the

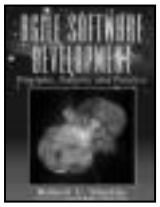
quality of the product being documented but, as a programmer, it leaves me despondent.

If you are going to do serious programming for .NET you need adequate documentation and you will need to choose between books such as these and the electronic equivalent. Alternatively you can rely on 'wizards' to do most of the work for you. However when such generate code fails to meet requirements you will need to refer to the documentation to discover exactly what the generated code does.

I do not think recommendations are relevant here. If you .NET documentation in hard copy you need it. If you do not you won't want these books cluttering your reference shelves.

Next time someone tells you how complicated the STL is, remember that you only need a few hundred pages to document it. A few months practice will allow you to master it. This won't be true for the .NET Framework libraries.

Methodology



Agile Software Development Principles, Patterns & Practice by Robert Martin (0 13 597444 5), Prentice Hall, 552pp @ £32-99 (1.67)
review by Uwe Schnitker

This book, as explained in the preface, was first intended to be a second edition of Robert C. Martins *Designing Object-Oriented Applications using the Booch Method*, but it turned out as a total rewrite, and includes several chapters intended to go into a forthcoming third edition of Booch's own *Object-Oriented Analysis and Design with Applications* – which didn't quite come forth – as well as several articles published on Robert C. Martin's Object Mentor website. Content dealing with Extreme Programming and, more generally, Agile Software Development Methods was added. This required a long time and lead to great anticipation, but also some confusion – e.g. some websites listed the book as 'available soon', under the old title, for some years.

My own anticipation of this book was mixed with some reservations: I had harvested the Object Mentor website for years, digesting and even ruminating all those articles and previews which would make the bulk of its content. Would there be anything new to be found? I had also noticed that Robert C. Martin had changed his focus from C++ almost completely to Java in the last few years, and is even strongly advocating now the use of dynamic, scripting, OO languages like Python and Ruby. The book would contain a lot of Java code examples. Would there be enough interesting stuff in it for me?

After finally receiving the book – a euphemism for 'robbing it from my the desk of my boss before he could really notice' – I was extremely pleased and even surprised by what I actually got.

The book looks very beautiful, with a richly coloured cover, almost like those popular introductory physics textbooks, with wonderful illustrations throughout, and lots of well-rendered UML diagrams and splendidly typeset code examples in all shapes and sizes. It is also very well organized, which was a pleasant surprise after I had tried in vain to understand the table of contents that is available online. (Note that he back cover text contains some of this seemingly inevitable marketing speak nonsense. The most disturbing – plain wrong and almost unbelievably stupid – assertion is that the book *contains a wealth of reusable C++ and Java code*. Why do first-class authors like Robert C. Martin allow such rubbish to be thrown at their books?)

Much to my astonishment, a large part of the code examples were written in C++, all the others in Java. No scripting language anywhere. Indeed, given that the book promotes Agile Methods and Extreme Programming at large, and the author mentions the huge impact of the internet colliding with the planet, the focus is refreshingly conservative, about writing programs and components, not just web services and stuff. (I'm a C++ programmer working on simulation and data preparation software products for the semiconductor industry.)

The C++ code shown – I can't speak for the Java examples, of course, but they look very nice, too – is of high quality, adhering to a sensible naming convention and really illustrating one of the most important points in the book, the Agile and XP mantra that code must communicate its intent as clearly as possible. Code examples are used to demonstrate all design principles and patterns mentioned, and a dedicated programming episode shows the practices – pair programming, test-driven development, simple design, refactoring and targeting design patterns – in action.

Of course, the case studies shown – as far as they are complete – are very small, and are intended to illustrate techniques that are only really beneficial for medium to large projects. This is the typical problem with teaching advanced topics and it can only be solved if you are able to abstract away from the examples to a certain extent. But if you can't, why would you develop software?

The other main point of the book is the idea that dependency management is the single most crucial issue with growing and evolving software. In the authors opinion OO techniques are so very useful simply because they allows dependencies to be managed very effectively. This has of course attracted strong criticism from those who promote the *OO models the real world better* and/or the *OO fits the way we think better* view, as well as critique for empirical-pragmatism. You make the call whether these accusations speak for or against this book.

As a passionate C++ programmer strongly interested in generic and multi-paradigm programming and modern C++ styles and techniques, I find the complete lack of discussion of these topics a bit unfortunate. Robert Martin discusses templates only as alternative to interfaces in speed-hungry programs and as a useful code-duplication remedy. (Please don't misunderstand me, he uses STL containers, algorithms and iterators were appropriate.) He also has the good taste and judgement to defer any discussion of C++ resource management and exception safety to the real experts on that topic, urging his readers to run, not walk, and get Herb Sutter's books, which is definitely good advice, if they want to do real C++ programming.

I'm convinced that quite a lot is left to be said about the interaction between modern, multi-paradigm C++ and Agile Development Methods. But no book can cover everything, and I'm not really unhappy that this one leaves room for a more specialized treatment of those issues.

To balance these omissions, the book also avoids any discussion of dynamic languages. Using those should have a tremendous effect on the dependency management issues shown. You don't need templates, you don't need inheritance, and many of the patterns discussed in the book are probably not needed any more or change significantly in focus – e.g. the gist of the visitor pattern is understood almost completely different by C++ vs. Ruby programmers.

Now for some more of the praise I should probably give to the book since I declare it to be highly recommended:

This book shows you how to effectively and efficiently design and code software that fulfils its duty now and can withstand the changes to come in the future. It tells you how to keep software soft and avoid the problem of software rot.

Robert C. Martin emphasizes that you cannot achieve these goals just by adhering to a process, even if it would be XP. Neither would you succeed with slavishly doing OO and using patterns. You need to have knowledge, experience, talent, creativity, and discipline, and you need to balance all those virtues with each other. And 'you' in the preceding sentence should better refer to a team working closely together and communicating well.

This book is also extreme fun to read. This is good and even important, because the book also shows you that software development can be fun, and should be fun, and that having fun isn't contrary to, but indeed necessary to, a high productivity process and a high quality product.

If you want to, have to, or just happen to program in C++ and/or Java – or at least can understand examples in those languages – and want to improve your way of writing software – whether or not you want become agile or to go extreme – you will probably profit from reading this book.

Extreme Programming Applied: Playing to Win by Ken Auer & Roy Miller (0 201 61640 8), Addison-Wesley, 326pp @ $36.99 (no UK price)
reviewed by James Gordon

This book sets out to help programmer and manager to implement an 'XP' style of programming. It doesn't show you what XP is, that is in the first book, this book puts that information into practice.

The book is heavily laced with examples of good practice, normally more than one way so that they fit in with people's needs and restrictions.

What I liked about the book was the stories of different companies that implemented XP and showing why it worked. It makes you feel like this book isn't just telling you what to do, rather showing how others have managed to get it working.

There is a lot of information in this book from across the whole scope of XP from 'reasons why your manager will never let it happen' to 'when to write acceptance tests'.

It's a good book, well-written and easy to read. It can certainly help somebody implementing XP, giving numerous examples of good practice and benefits of thinking differently.

A Requirements Pattern by Patricia Ferdinandi (0 201 73826 0), Addison-Wesley, 506pp @ $39.99 (no UK price)
reviewed by Michael J. Pont

I found this book extremely difficult to fathom. The title – and table of contents – led me to expect a book of patterns for requirements engineering. That is, I expected to find a book along the lines of 'Design Patterns' by Erich Gamma et al., or 'Analysis Patterns' by Martin Fowler. In fact, there is pattern collection in this book, but it is 'hidden' in Appendix D and occupies only 20 pages out of a total of around 500.

In place of the pattern collection I had expected, this book consists of nine, rather long, chapters describing techniques that are intended to help the reader 'succeed in the Internet economy'.

Overall, I was perplexed by both the structure and contents of 'A Requirements Pattern'. I'm sure that Ferdinandi has considerable experience in the field of requirements engineering. However, I did not feel that she succeeded in communicating her experience to me through this book.

Writing Better Requirements by Ian F. Alexander and Richard Stevens (0-321-13163-0), Addison-Wesley, 158pp @ £19-95 (1.75)
reviewed by R N Lever

Requirements are a fundamental part of any project, however, that does not mean that this is a subject that has been so well documented that there is nothing left to learn. In fact many might suggest that writing requirements is not understood at all based on some so-called requirements documentation. That is where this book aims to help – writing better requirements.

Although it is not a large book (~150 pages) it does cover the subject in sufficient detail that you will undoubtedly learn something even if you think you know a bit about it to start with. The book itself is structured around capturing, organising, writing and reviewing requirements. Each topic is addressed in an engaging, clear and concise style and each chapter includes exercises to help reinforce the material. These questions have some suggested answers at the back of the book and this is a useful way of realising how little critical thinking is applied to even some apparently simple statements.

There is a lot of good commentary in this book and unless you have spent a great deal of time learning the ropes and improving continuously you will learn something new. This learning will be reinforced by the many examples of both the right way and the wrong way to produce requirements. For those people who are interested in this area it is a very useful book to have to hand. Before I started reading this book I thought I knew something about writing requirements – I definitely know more now.

Database Issues

Data Analysis for Database Design 3ed by David Howe (0 7506 5086 9), Butterworth Heinemann, 356pp @ £21.99 (1.50)
reviewed by Asad Altimeemy

This is a very good guide for a beginner in data base analysis and design and indirectly useful to practitioners of object-oriented analysis. Concepts of ER data model are explained in a very easy and clear way. The book follows a very good and fundamental approach to data analysis and design. Although it is a small book, it covers all the important concepts of data analysis and design.

The book is much more suitable for self-study than most. Every section of each chapter ends with useful questions and exercises. Suggested answers and solutions are at the end of the chapters.

The sections Part 2: Relational Modelling and Part 3: Entity-Relationship Modelling are the heart of the book. Howe's treatment of these two topics is very good. However, Part 4: Implementation shows how to do physical database design for a network-style (CODASYL) database management system. This has little practical value, as CODASYL-compliant DBMS products are a bit out of date. But even this material will be useful to OODBMS designers. Just substitute 'collection' or 'container' (an OOD concept) for 'set' (a CODASYL concept) and you will be well on your way.

UML for Database Design by Eric Naiburg & Robert Maksimchuk (0 201 72163 5), Addison-Wesley, 300pp @ $39.99 (no UK price)
reviewed by Asad Altimeemy

This book introduces the Unified Modelling Language (UML) and how it can be applied to database modelling and design. The authors describe how the artefacts provided by business modelling are used to establish system requirements, address issues that arise during the mapping of object models to data models and illustrate application of the UML diagrams and techniques through a recurrent case study

However, the book contains many errors and peculiarities. There is the occasional inane comment, e.g. *'It is important to have the database running at full speed all the time; this can be accomplished by having a well-designed database and taking advantage of specific DBMS properties. Running the database with the correct amount of storage helps keep the database running at its best*' (p. 152)

Most annoying of all, however, is the proliferation of descriptions that explain nothing, e.g. *'After the intense mining of already captured information and using much of their own knowledge of database design and experiences building several other databases, the database designers make some decisions on how to build the database storage in table spaces. The team determines that there is a need to have five different table spaces ...*' (p. 164).

This book is very poor for a UML book. The book's website is empty and does not supplement the book's content – it only contains a link to order the book and an email to contact the authors.

SQL Performance Tuning by Peter Gulutzan and Tmdv Pelzer (0-201-79169-2), Addison-Wesley, 528pp @ $44.99 (no UK price)
reviewed by Christopher Hill

Take eight computers. Install a different database system on each (MySQL is included in the eight). Then work your way through the SQL language seeing how you can make significant portable improvements in performance.

This book is a labour of love. The authors try out all the wheezes and 'everybody knows that.' to produce a catalogue of ways to write portable SQL code (ANSI/ISO standard SQL:1999) that works swiftly over the major platforms.

The relative performance of the database is not compared (this is prohibited by the vendors) but for each item two snippets of SQL are compared and if there is a better than 5% improvement on a platform it gets an 'improvement' tick. Most items get 8 ticks. A very few have a negative impact on particular platforms.

The first third of the book covers the main parts of the SQL that cause bottlenecks: simple searches, Order By, Group By, Joins and Subqueries. They then move on to the physical aspects; Columns, Tables, Indexes, Constraints and Stored Procedures. The final third looks at ODBC, JDBC, Locks, Client/Server and Cost-based Optimisation.

Given the subject matter, inevitably the book has a 'cook book' feel, although the writing style does a great deal to reduce this. There is plenty of background information on why you should use this technique over another, so you can decide which is best for your circumstances. Some solid theory is also presented (including Btrees and normalisation) in a very approachable manner.

'Your DBMS is your pal. You should try to get to know it better and help it to help you'. This should not be your first book on SQL, but every DBA and SQL writer should have a copy to hand. Highly recommended.

Cryptography for Internet and Database Applications by Nicholas Galbreath (0-471-21029-3), Wiley, 400pp @ £29-95 (1.34)
reviewed by Christoph Ludwig

The book promises to tell you how to cryptographically secure your applications with Java. However, even though it gives an overview of public and symmetric key cryptography, introduces Java's cryptographic frameworks JCA and JCE and discusses typical issues when implementing web and database applications, the book falls short. The book suffers most from two problems; the unclear target readership as well as the numerous and often severe errors.

The book's strong points (like its extensive, partially commented bibliography) cannot make up for its shortcomings. Not recommended.
[see website for full review]

Linux & Unix

Tuning and Customizing a Linux System by Daniel L. Morrill (1-893115-27-5), Apress, 433pp @ £28-00 (1.60)
reviewed by Joe McCool

There is sparse little in this book about 'Tuning' anything. Unix and Linux are close cousins and if what is meant by that word is the accepted usage under Unix, then it has little to offer. Indeed the word 'tuning' doesn't even have a mention in the book's index !

To me 'tuning' means altering parameters in the kernel – constants in source or header files. Sometimes a simple linking process follows this or perhaps a complete recompile and install. The idea is to optimise for processing speed, disk usage, disk access or whatever. This has parallels with tuning a motorcar engine for maximum performance against fuel consumption or longevity.

This book covers nothing of that.

On the customisation front it scores little better. Morrill mentions possibilities like the Apache web server, the CVS versioning system, security; PAMs, the secure shell and the SOCKS library, but it lacks meat. The ratio of prose paragraphs to source code is very high. It is not even informative about the rich sources of mailing lists, newsgroups, FAQs, HOWTOs and SxSs (step by steps) on the net. Without these one will very quickly grind to a halt using or administrating Linux.

I don't want to be totally dismissive of this book. There is some useful material on the history of Linux and relationships and differences between the various proprietors and the chapters on security offer interesting overviews. However, I feel that it adds little to the publications already out there and if ones objective is to set up a Linux system, the money might be better spent elsewhere. Essentially Linux is free. In keeping with this, the best documentation is often right under one's nose, on line.

LPI Linux Certification in a Nutshell by Jeffrey Dean (0 59592 748 6), O'Reilly, 551pp @ £28-50 (1.40)
reviewed by Joe McCool

This is a gem of a book. For someone wishing to get up to speed on Linux, or wanting to brush up on skills, it might be worth buying, over and above LPI certification. O'Reilly publishes other Nutshell works on Linux, but the exercises at the end of each chapter in this one are particularly useful. It is full of rich little crumbs of knowledge that I've seen nowhere else before. (For example, I had no idea the ! operator was so powerful on the command line.)

Certification is sometimes a contentious issue. In the IT industry opinions are divided as to its usefulness. It is assigned little credence among SCO Unix gurus for example. In the job market, I can see sparse demand for it, with the exception of the Microsoft variations. It has a place in that world, perhaps, as a means of sifting the varied, voluminous and often suspect levels of expertise.

Typical of fragmented Linux/Unix there are too many suppliers and too many certifications available. Which one will win out in the long run will be determined by the commercial success of its sponsoring company. Red Hat offer two. SCO/Caldera several (if the Unix qualifications are included), but this is likely to stabilise now with SCO becoming involved in United Linux and in turn with LPI.

The Linux Professional Institute offer a certification that is independent of any particular proprietary Linux. (In keeping with this Dean covers material on both the Red Hat and Debian package management: rpms and apt/dpkg.) The idea being, partially, to reduce confusion on the part of employees and candidates alike. Only time will tell if the venture has been a success. I have no figures on the uptake. Those interested might like to have a look at`www.lpi.com`.

One can pursue certification for reasons other than seeking employment. One can pursue it for fun and over the next six months or so, that is my objective. In doing so, Jeffrey Dean's book will play a large part in my efforts.

Production is in keeping with the high standards we've come to expect from this publishing stable, although I did find the odd bum steer amongst the illustrative figures and worked examples. If anything this adds to the thrill. It is easy to check a text like this with a keyboard and working Linux system on the same workspace. It really is fun.

Understanding the Linux Kernel 2ed by Daniel P. Bovet and Marco Cesati (0-596-00213-0), O'Reilly, 765pp @ £35.50 (1.41)
reviewed by Tony Houghton

This title has been around for a while, but the 2nd edition is new and covers kernel 2.4; the book is commendably up to date, but for how long? Kernel 2.5 (the version in development) is in feature freeze with 2.6 (the next major release that 2.5 will turn into when it's considered stable) due by the summer. Such problems are the bane of people writing about rapidly developing software.

Obsolescence isn't the only problem caused by the subject matter; an OS kernel is very complex. Despite what the introduction says, I don't think this is a good book for anyone who's just curious about the Linux kernel; it's a book that has to be studied rather than read and requires a good deal of motivation, making it more suited to academics and those wanting to get involved in kernel development. A reasonable knowledge of C is required and knowledge of x86 architecture and assembly language is helpful.

The subject matter is loosely divided into memory management, processes, interrupt handling, calls between processes and to the kernel, filing and networking. The divisions between these subjects are somewhat blurred and they encompass too much for each to be covered neatly in its own chapter, so the subjects appear somewhat interleaved.

I found this book very difficult to get to grips with, but I feel confident that it's of a high technical standard and that it would be impossible to cover the Linux kernel so thoroughly without being difficult to read. Therefore I hope that someone else will offer to give it the more in-depth review I think it deserves.

Unix Power Tools 3ed by Shelley Powers et al. (0-596-00330-7), O'Reilly, 1116pp @ £49-95 (1.40)
reviewed by Tony Houghton

The title of this book rather understates its scope. 'Unix power tools' conjures up images of programs that are very powerful if you read reams of man pages and learn arcane syntaxes and options, but are beyond the reach of mere mortals. While it does help with those, the book also has a wealth of tips on how to use the simpler and every day tools more effectively.

Although my Unix use is almost entirely Linux, the good thing about books which are more generic is that they explain the old fashioned tools and techniques which can still be used to very good effect in Linux and once learnt are often more powerful - in ways that can actually make them easier to use for those that are familiar with them - than the front ends added in recent years to widen its appeal to users brought up on GUIs. Linux specifics aren't neglected here though and even Mac OS X crops up a few times.

Broadly, 'Unix Power Tools' covers the general user environment (the common shells and X), working with files, text editing - both interactive and non-interactive, process management, scripting, installing software and backing up, networking and security. Rather than being a reference it's more of a well-arranged collection of articles containing hints and tips.

What I found to be one of the best features is the excellent cross-referencing. Articles often briefly mention a related topic, when they do a link is given to where it's covered in more detail.

Within minutes of picking up the book I had learnt a few useful facts about programs I use every day without realising before that they could do that. I think the book's most useful to someone who's either just installed Linux and wants to learn how to get the most of it, or has been plonked in front of a commercial Unix workstation and told to get on with it, but also has a lot to offer far more experienced and expert users. Highly recommended.

Other Programming

HTTP The Definitive Guide by David Gourley Brian Totty (1-56592-509-2), O'Reilly, 635pp @ £31-95 (1.41)
reviewed by R N Lever

This is a book about HTTP, which describes the various aspects to allow the reader to understand web internals. The book is composed of six parts:

1) HTTP: The Web's Foundation (HTTP, URL, resources, messages, connections...)
2) HTTP Architecture (web servers, proxies, caches, robots, gateways...)
3) Identification, Authorisation and security (basic, digest, secure...)
4) Entities, Encodings and Internationalisation (entities, encoding...)
5) Content Publishing and Distribution (web hosting, publishing...)
6) Appendices (URI, Status Codes, Header Reference, MIME types...)

The authors have produced a very clear and easily read text that explains the protocol and how it works along with some of the core Internet technologies. There are plenty of diagrams to help clarify the processes and to ensure that the reader can see how everything works. Interspersed within various parts of the text are snippets of code that illustrate a particular point. These code snippets are typically in Perl or C. However, for programmers there are not enough code examples to be used as a programmer's reference guide.

At just over six hundred pages there is plenty of room to cover the topic and the authors have made a good attempt to explain the various different areas in a clear and understandable way. For those who wish to gain an understanding of HTTP then it will undoubtedly be useful. However, the book will not provide enough code for programmers or discuss some of the protocols

that have been built on top of or around HTTP such as HTML, XML or SOAP. So for its target audience, those who wish to understand HTTP, it will provide a useful and clear guide.

Non-Programming

Windows XP Pro: The Missing Manual by David Pogue, Craig Zacker, and LJ. Zacker (0-596-00348-X), O'Reilly, 658pp @ £20-95 (1.43)

review by Francis Glassborow

If you use Windows XP and like having hardcopy manuals this book will meet your needs.

[see website for full review]

Winn L. Rosch Hardware Bible, 6ed (0-7897-2859-1), Que, 1128pp @ £28.99 (1.38)

review by Francis Glassborow

If you want/need a comprehensive guide to hardware for PCs this one is well-written and well-informed.

[see website for full review]

Google Hacks by Tara Calishain and Rael Dornfest (0-596-00447-8), O'Reilly, 330pp @ $24.95 (no UK price)

review by Francis Glassborow

This is an interesting book that will extend your understanding of how you can use Google. It will tell you about many things that are not immediately obvious to the casual user of Google. For example it will tell you about the GooglePeople engine located at `www.avaquest.com`. Using that can be both informative at times and funny at other times. I tried 'Who is Francis Glassborow' and received the response 'I am not sure but one of these might help.' and that was followed with a list headed by Andrew Koenig. I wish I had checked that before the ACCU Spring Conference 2003 because it would have given me a great opening line to presenting Andy's keynote for him.

While I think that O'Reilly are guilty of over publishing at the moment (too many books on niche subjects) I think this one deserves to sell well. If you want to improve your use of the net for information this book might help you.

A Pattern Language for Web Usability by Ian Graham (0-201-78888-8), Addison-Wesley, 283pp @ $39.99 (no UK price)

review by Francis Glassborow

Let me start by quoting the first paragraph from the back cover:

> Despite the astronomical number of hours invested in developing websites for commercial and other uses, it is now clear that many websites are poorly designed and have floundered as a result.

This book focuses on the need for usability. However good the content, the navigation and the aesthetics the site will likely fail to meet expectations if it is not usable.

Patterns in software design terms refer to the encapsulation of standard solutions to common problems. Actually I should qualify the use of 'standard' with 'effective'. A standard solution may be ineffective. We see a great deal of that when surfing the web, where so many sites seem incapable of understanding that the effective visual aspects of printed fliers do not transfer to effective electronic 'fliers'. In both cases we have a minute window of opportunity to grab the attention of an individual. Beautiful and/or startling graphics may achieve that on paper but if they result in loading times in excess of a few seconds then they will be failures on the web.

In this book the author covers 79 patterns (and their inter-relationships) that cover issues of web usability. The book derives from a workshop at OT2001 (held in Oxford, England). Those that are familiar with this event will realise that this book will be a serious and well-designed example of a pattern language. Indeed I think the book actually is a good case study of the concept of a pattern language and is worth studying from that perspective. However if you are responsible for a website (if only your own home page) this book could be a major factor in helping you do a professional job rather than the job that so many so-called professionals are satisfied with.

Web Metrics by Jim Sterne (0 471 22072 8), Wiley, 430pp @ £22-50(1.33)

reviewed by Christopher Hill

This book is written by a management consultant for middle to senior management in large firms. There is a great deal of useful background information and theory, but just when it starts to get interesting, it peters out (as it gets too technical for the audience), or you spend $10,000 (only to realise 'we probably needed the deluxe version that cost 15 times as much!').

So what do you want to measure? A very good point is also made that measurement is not enough – comparison against a target is the mantra here. Are you doing better or worse?

There are no standards in web metrics and so talking figures is very difficult, clicks vs. hits vs. page views vs. sales. There is a summary of what data is in the server log file and a brief discussion about what you might do with them. How do you measure the speed of your website? What do you do if you are very successful?

To gauge if your web site is successful – first define what success means to you. Maybe a reduction in calls through the support desk; or improved brand image; or more sales?

Which are the best ways to attract people to your site, keep them there long enough and get them to come back again. Part of this 'stickiness' might be personalization of the web site for the visitor. What points might you configure per visitor, how do you help navigate your site and many other nebulous topics.

This is a book for the big boys and girls in web land. If you have lots of money and lots of staff then I would recommend that you get your own copy of this book. For the rest, borrow a copy from the library for the few useful ideas that mere mortals can use.

How to do just about anything on the Internet by Reader's Digest, (0-276-42560-X), 350pp @ £42 (1.49)

reviewed by Silas S Brown

This book is a shame. It is meant for complete beginners who want to buy a computer and jump on the Internet 'bandwagon' and it contains 350 pages of material that might appeal to them. However, in the name of simplification it contains many mistakes, especially near the beginning (which is the most important part of this kind of book). Besides factual errors (e.g. 'Internet' is NOT short for 'International Network'), confusing recommendations with requirements (e.g. they say 'you need 32Mb of RAM to go on the Internet') and the inevitable unspoken assumption that the entire world uses Windows PCs (it would only have taken a few sentences to acknowledge the existence of other platforms and to define the book's coverage more accurately), the book promotes questionable practices in the area of Web design and fails to adequately address legal issues when downloading music and so forth.

I could go on, but I couldn't bear any more of it. It is books like this that contribute to the large numbers of people who don't really know what they're doing on the Internet. I was given this book by a friend who had mistakenly purchased it from one of the Reader's Digest book-purchasing schemes. She asked me to sell it to someone, since I was 'in the trade'. I will do no such thing; even if someone would buy it (which is unlikely), I couldn't in good conscience pass on a book like this to anybody.

I hope they'll improve it in the next edition.

Hackers Beware by Eric Cole (0 7357 1009 0), New Riders, 780pp @ £34-99 (1.29)

reviewed by Asad Altimeemy

This book is written so that beginners can get an understanding of what hacking is about and how to protect against it. The examples are thorough and provide step-by-step screen shots as to what happens during a certain exploit.

The book details how attackers gain access into different type of operating systems and hardware platforms. More importantly, Cole describes countermeasures to use to defend against the various types of attacks and exploits he describes. Overall, this is an excellent reference for anyone needing to understand how hacking works and how to defend against it.

Hackers Beware is a good reference for individuals who are new to the information security field or managers who need to have some understanding of information security. It discusses in basic terms the general issues that affect network security. Technical issues are discussed in a manner that allows non-technical individuals to understand the severity of the vulnerabilities, attacks and exploits

The most useful section in the book concerns the Internet. This chapter on the SANS Top 10 Exploits is excellent, as it describes what exploits must be fixed for any organisation that is connected to the Internet.

This is an excellent book for any potential reader who is looking for a text that gives an overall viewpoint of hacking, hacking techniques and defending against hacks.