The magazine of the ACCU

www.accu.org

{cvu}

Volume 19 Issue 1 February 2007 £3

Features

Pete Goodliffe
Progamming is...

Jez Higgins
Adventures in Autoconfiscation

{cvu}
Volume 19 Issue 1
February 2007
ISSN 1354-3164
www.accu.org

Editor
Tim Penhey
cvu@accu.org

Contributors
Lois Goldthwaite, Pete Goodliffe, Jez Higgins, Derek Jones, Roger Orr, Tim Penhey, Simon Salter, Emily Winch

ACCU Chair
Jez Higgins
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
David Hodge
accumembership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Thaddeus Froggley
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

A Mission to Increase Numbers

I was out for dinner with a friend some nights back, and we got to talking about ACCU. I was trying to explain what it was all about and why it is a good thing. He asked me “Are you looking to expand the membership?” I thought that was a funny question, and answered “Yes, of course.” He looked at me square in the eye and said “So what is your mission statement?” And there he had me somewhat stumped. I blurted out something about “professionalism in programming” and he nicely pointed out that what I said wasn’t anything close to a mission statement. The next day I was talking with Jez, the chair (ACCU chair, not ‘a’ chair), and spent a little time searching out how to make a mission statement, and looking at other companies’ and organisations’ mission statements. After a few, occasionally humourous, exchanges I came up with the following that I’d like to throw out there to the association:

> To improve the standards of practising programmers through association and mentoring.

Now does that sound like something that some of your co-workers or employers might be more interested in? I was talking to a co-worker of mine, mentioning ACCU and the “professionalism in programming” tag line, and he came back with “Do you need professionalism in programming?” I think this shows a distinct lack of understanding of professionalism, but we need to combat that up front in order to make ACCU more appealing to the programming masses.

For those of you who are not aware, there are a few new books available that are authored by members of your association. Pete Goodliffe’s book, *Code Craft*, is now available from No Starch Press, a new edition of Russel Winder’s book, *Developing Java Software*, is out and Matthew Strawbridge’s new book is called *Netiquette*. I think that a little self promotion in C Vu is a good thing as it lets you know that your association has very talented and dedicated members.

Lastly I’d like to wish all our members a happy new year. I feel it is time for a little personal disclosure. Here are a few of my new year’s resolutions:

1.Eat less and exercise more

2.Don’t wait until a few days before the submission deadline before writing articles

3.Get conference talk written well before the conference

Tim

TIM PENHEY,
EDITOR

The official magazine of the ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers’ conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by ACCU members – by programmers, for programmers – and have been contributed free of charge.

To find out more about the ACCU’s activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

FEATURES

REGULARS

COPY DATES

C Vu 19.2: 1 Mar 2007
C Vu 19.3: 1 May 2007

IN OVERLOAD

'Managing Technical Debt' by Tom Brazier, 'Exceptional Design' from Hubert Matthews and much more!

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

Programming is...

Pete Goodliffe has a surprise revelation about programming and monogamy.

I don't spend all of my time pontificating about programming, writing books and articles. Sometimes I actually write some code. Hard to believe, I know. But it's true. You have to find a way to pass the time.

These days I'm working almost exclusively in C++, although I keep my hand in a few other languages because it's a bad idea to ever let your brain stick in one gear. However, using C++ in anger for many years (sometimes, quite literally in anger) has been a great deal of fun, and a very enlightening experience. So much so, I realise that I now have a love/hate relationship with the language. It's gorgeous, in a sick-and-twisted kind of way. Old C++ hacks know what I'm talking about.

Oh! For an easy life...

I don't hate C++. It just annoys me sometimes. It's intelligent, deep, expressive, powerful, and great fun. But it's not perfect. C++ is a very sharp tool and, like all sharp tools, it has sharp edges. Often you can wield the language to produce incredible, expressive code in an way that no other language can match. And (hopefully not quite as) often, you accidentally cut yourself on one of its sharp edges and end up mumbling something involving expletives and "stupid flipping C++".

These mumbled expletives don't prove that C++ is a bad language (although some would claim that they do). It's just a language that you have to learn to live with. A language that you have to understand well to work with properly. You must grok how it works under the covers, and what makes it tick. You can work in many other languages without making such a serious commitment.

And then it occurred to me...

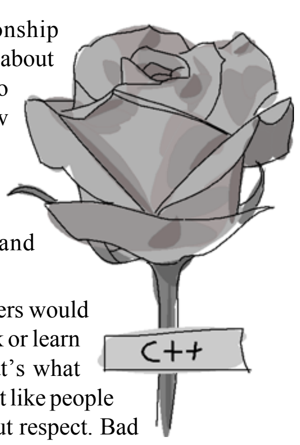
Programming in C++ is just like marriage. It's great. It's rewarding. But it requires work. You can't expect to breeze into C++ programming and for everything to be perfect. For your code to be great with no effort. For your life to be as easy as it was in a less committed programming relationship. It just doesn't pan out like that. You have to work at it. Put some effort in.

But it's worth it.

Sure, there are a few people who just seem to click with C++, who never have any problems with it. But they are few and far between. And they're probably not in a real programming relationship with C++; it's just a superficial arrangement without any really deep interaction going on. Programming for convenience.

You have to be careful how you treat C++. It's a fickle beast; it gives you many wonderful and exciting techniques to play with. And in doing so it simultaneously gives you plenty of rope to hang yourself with. With templates come unfathomable type-related error messages. With multiple inheritance comes greater potential for bad class design and the 'deadly diamond'. With `new` and `delete` come memory leaks and dangling pointers.

Does this mean that C++ is bad? That we should ditch C++ now and all use Java or C# instead? Cripes, no! Some programmers argue for this, but I think it's a very misguided and myopic viewpoint. Sometimes there is a place for a 'programming for convenience' relationship. And often there isn't. You don't get anything in life for free; you must expect to invest something costly to achieve a fulfilling relationship. It's hard to live with someone (or something) day-by-day, to share your most intimate (programming) thoughts, and to not get upset with them occasionally. You can expect a certain amount of friction as you get close and become accustomed to one another. Any relationship leads you along a path of constant learning about each other, of working out how to accommodate each other's foibles, and how to bring out the best in each other.



Admittedly, C++ isn't going to put much effort into learning about you, but many clever people (the guys who designed and standardised the beast) already did.

The fact that many professional programmers would rather earn a fortune without having to think or learn about the language is a sad thing, but that's what leads to so much bad C++ code. C++ doesn't like people to take advantage of it, or to treat it without respect. Bad programmers dash out C++ code without engaging their brains, and the result is a mess (whether they realise it or not). Those guys simply shouldn't be writing in C++; there are plenty of other languages they can prostitute themselves with. And there are many programmers who have divorced C++ as their first language and taken up with a younger, more glamorous alternative. (It's always a big ego boost to be seen with a young trophy language on your arm. Is this the programmers' equivalent of a mid-life crisis?) [1]

Of course, it's not just C++ that suffers from programmers who are not prepared to invest the appropriate level of commitment to live in happy union with the language. It's just one of the languages that tends to show the signs of neglect quickly, and that tends to make it's feelings known early on. You know when some C++ code has been neglected. (You come home late one time too many via an indirected pointer, and find your object inside the dog. And try as you might, you can't see what you've done to offend the language.)

So, C++ demands love and attention. You have to commit to it to be able to produce fulfilling code. But, you know – that's why I love it! When you give it the time and attention it deserves, you can do some quite amazing things in C++. To temper this you have to constantly think about what you're doing, and take responsibility for your work. It's tempting to use cool language features for feature's sake, or employ hi falutin idioms because they're cute. But the good programmer knows the correct, pragmatic, approach. When to let loose and when to hold back. As the mighty Spiderman was told: with great power comes great responsibility. C++ gives you the power. You supply the responsibility.

Cultivating your language relationship

There are still column inches to use up, so can we flog this metaphor any further? Can this programming/marriage comparison teach us anything useful? Well, perhaps it can.

There are a few generally accepted hallmarks of a healthy marriage. They can shed some light on a healthy relationship with C++. In fact, they're descriptive of a healthy programming approach in general, since very little of this is specific to C++. A good marriage requires:

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@cthree.org

- **Love**

  OK, let's start with the mushy stuff. For a marriage to succeed, the partners must like each other, value each other, and want to spend time with each other. There must be a level of attraction. They must love each other.

  Most code monkeys program because it's their passion. They love to write code. And they usually pick a language because they genuinely love it. Perverse, perhaps. But true. Now, many people are forced to use C++ at work because the existing codebase is written in it – in this sense they enter an arranged marriage. At home they'd rather tinker with a cool bit of Ruby or Python. Remember, some arranged marriages work perfectly well. Some do not.

  How much does your appreciation of a language shape the quality of the code you write, or the way you'll improve your skills working with it? How much of this is born from acceptance and a growing familiarity?

- **Commitment**

  These days commitment is not a fashionable word. With the erosion of the value of marriage in modern society, people have come to expect to be able to jump out of a 'committed' relationship as easily as they can enter into it. But that's not healthy by any means.

  To become an expert programmer in any given language, or with any technology, you must have a commitment to learn about it, to spend time with it, and to work with it. You can't be selfish and expect it to pander to your every need, especially when it is specifically designed to suit many diverse situations and requirements.

  Commitment may also mean that you have to sacrifice. You must give up some of your preferred ways of living and working in order to accommodate the other party. C++ has particular idioms and ways of working that suit it best. You might not like them, or would prefer to work in other ways. But if they are the definition of 'good' C++ code then they are the idioms you should adopt.

  Does your commitment to writing good code in the current language supercede your desire to do things your own way? Good code or an easy life? It's all about commitment.

- **Communication**

  In a good marriage you constantly communicate. You share facts, feelings, hurts and joys. You don't just talk at a superficial level as you would to acquaintances you meet in the street, but at a real heart-to-heart level. It's deep. You share things with your partner that you would share with no one else. This kind of communication requires an immense level of trust, acceptance and understanding.

  This isn't necessarily easy; men and women communicate in very different ways. Communication can be very easily misconstrued or misinterpreted. It takes a huge effort to communicate successfully in a marriage. It's something that you have to pay attention to and make a constant effort with. Communication is very much a skill that you learn, not just something you can or can't do.

  The act of programming is entirely about communication. The code we write is as much a communication of the intent of our program (both to ourselves and to other programmers who might pick it up) as it is a list of instructions for a computer to execute.

  In this sense, we communicate both to the language – to tell it what to do, in a clear, concise, unambiguous, correct way – and we also communicate to others using the programming language as the medium. Good communication is a vital (and often lacking) skill in high quality software developers. It takes a huge amount of effort, and constant attention, to do this well.

you must have a commitment to learn about it, to spend time with it, and to work with it

- **Patience**

  Good marriages don't appear overnight. They are cultivated over time, bit by bit. They grow. Gradually. In our fast food culture we have learnt to expect everything now, instant food, instant cash, instant downloads, instant gratification. But relationships never work like this.

  It's the same with our programming relationship. You can learn of the existence of a language in an instant. You might even have an instant attraction. Programming lust. But it can take a lifetime to master a language fully, to be able to honestly claim that you really know how to write 'good' code in it. It can take a long time, and an awful lot of patience, until you fully appreciate the beauty of a language.

- Shared values

  A strong glue thing that holds many relationships together is a common sense of morals, values and beliefs. For example, research shows that couples with shared strong religious beliefs are far more likely to stay together than those without them; it acts as a solid foundation to build the relationship upon.

  If you don't agree with C++'s basic values – that many facilities and idioms are available, but cost you nothing if you don't want to use them, that multiple inheritance can be a healthy and useful design tool, that templates can tame vast amounts of complexity (as well as introducing their own kind of complexity) – then you'll always have a skewed relationship with the language.

Illuminating. But no metaphor is perfect. Is fidelity as important to healthy C++ as it is to a healthy marriage? No. It is actually very useful to 'play the field' and mess around with other languages on the side. Make C# your mistress, and Python your muse. It'll teach you more about different programming skills and techniques, and help you to avoid getting stuck in a programming rut.

Or is that actually very like marriage? I'll leave that for you to decide...

Conclusion

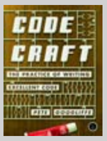
Sometimes it's really useful to anthropomorphise the things you work with. To try to get yourself into 'their' mindset. Learn to think like C++ does. To empathise with the confounded language. Yes, it's freakish. (And quite clearly 'wrong' – just don't tell your friends down the pub. Or your psychoanalyst.) But it is a genuinely enlightening exercise.

This colourful marriage metaphor shows us that knowledge of a programming language isn't all there is to programming. Consider how you work with your tools, the kind of relationship you have with them. Good programmers think about more then mere lines of code, or an isolated code design. They care about how they use and interact with their tools, and how to get the best out of them, as much as they care about simple fact-based knowledge of them. Good programmers don't expect quick-fix answers to problems, but learn to live with and appreciate the strong- and weak points of their tools. They commit to a life with them, and invest time and effort getting to know them. They appreciate and value them.

So... what languages and tools do you work with? And what tenuous metaphors can you concoct for your life with them? ■

Endnotes

[1] How many people ditch their first relationship in the hope of trading it from something that requires less maintenance, only to discover that the replacement model is just as fickle, equally as hard to live with, and not as fulfilling?



The Book List

Tim Penhey sets a python on the ACCU book list.

As most of you know, ACCU has books that are available for review. These books are made available by wholesalers and publishers in the hope that good reviews will increase the sales of the books. I'm not going anywhere near the topic of terrible books, or ones that should never have made it to print. The point of interest as far as I am concerned right now is that ACCU gets books to review. These books need to be recorded, handed out for review, and if all goes to plan, reviews sent back for publication in C Vu and the ACCU website.

For argument's sake, lets consider that we are storing the list of books as a comma separated value (CSV) file. It isn't really, but it was fairly easy to export it as CSV for the purpose of the exercise.

What do we want to do?

As far as dealing with the books for ACCU, there are a number of things that we are interested in:

1. Which books need to be displayed on the web as available for review
2. Which books have been hanging around for a long time and not been claimed
3. Which books were sent out some time ago and does not have a review yet

Answers to each of these questions trigger other events:

1. Update the web site with new books shown and claimed books removed
2. Send emails to accu-general suggesting people review the "older" books (hopeful, I know)
3. Chase up people for timely reviews

As well as being able to answer these questions, we also want to be able to add new books easily, and remove books once review have been published and added to the ACCU review website. (Backslashes have been added to Figure 1 by me to show line continuation.)

Figure 1

```
tim@slacko:~/Desktop$ head -2 stocklist.csv
"Web Accessibility - Web Standards and Regulatory Compliance", "Various", \
"Friends of Ed", "1-59059-638-2",2006,02/10/2006,,
"The Official Ubuntu Book + CD","Various", "Prentice Hall","0-13-243594-2", \
2006,02/10/2006,,
```

What we have here are the following values stored as CSV:

1. Title
2. Author
3. Publisher
4. ISBN
5. Year of publication
6. Date book was added to the list
7. Who has claimed the book
8. When they claimed the book

As you can see for the above two values, we don't have values yet for who claimed the book or when.

The preliminary work

Firstly, let's create a class to encapsulate our books.

Listing 1

```
#!/usr/bin/env python ①

class Book(object): ②
  def __init__(self, title, author, publisher, ③
               isbn, pub_year, date_added,
               reviewer=None, date_claimed=None):④
    self.title = title
    self.author = author
    self.publisher = publisher
    self.isbn = isbn
    self.pub_year = pub_year
    self.date_added = date_added
    self.reviewer = reviewer
    self.date_claimed = date_claimed
```

Notes on Listing 1

① For Linux systems it is a common idiom to execute the python as defined in the environment rather than hard coding a particular path.

② Classes are defined using the **`class`** statement. The name following **`class`** (**`Book`** in this case) is the name of the class, and the names in the parentheses are the base classes. Python can handle multiple inheritance, and base classes are defined by a comma delimited list. Also defining **`object`** as a base class makes the class a "new" style class. "Old" style classes are generally considered deprecated, but since the behaviour is subtly different, they could not have been removed without breaking code that already existed.

③ **`__init__`** is the constructor for the class. The variable scoping rules for python do not include object variable lookup, so all references to instance variables are explicit. Functions defined at the class level that are intended to be used as member functions take **`self`** as the first parameter (compare that with C++ implicit **`this`**).

④ Both **`reviewer`** and **`date_claimed`** are optional parameters. If the parameters are not passed into the constructor, then the default value is used.

Creating an instance of a class is just like calling a function, with the return value being the initialised object.

```
>>> import booklist
>>> book = booklist.Book('A title', 'The author',
'Publisher', '123-45678', '2006', '2006-12-31')
>>> book.title
'A title'
>>> book.reviewer is None
True
```

TIM PENHEY

Tim believes in choosing the right tool for the job. After years of hard core C++ hacking he's found that some things are just easier in Python. He can be reached at tim@penhey.net

Now that we want to be able to create a list of books based on the contents of the file. So now we'll create another class.

```
class BookList(object):
    "Holds the list of books"
```

We looked at some basic file functions in the last article: **open**, **read**, **readlines** and **close**. One way to address the problem is to read the lines in from the file, and split the string on commas. Then you hit the problem of embedded commas in strings and it no longer seems like such a wonderful idea. Luckily there is a module in the python library for dealing with CSV files. Unsurprisingly it is called **csv**.

Listing 2

```
>>> import csv
>>> reader = csv.reader(open('stocklist.csv'))
>>> for line in reader: ①
...    print line
...
['Web Accessibility - Web Standards and
Regulatory Compliance', 'Various', 'Friends of
Ed', '1-59059-638-2', '2006', '02/10/2006', '',
'']
<<-- snipped results -->>
```

Note on Listing 2

① The reader created is an iterable object, and returns a list of parsed fields.

Conveniently we ordered the parameters of the constructor for the Book to be the same order that they are stored in the CSV file. These could then be passed as:

```
  book = Book(line[0], line[1], line[2], line[3],
     line[4], line[5])
```

Python has an inbuilt operator for lists that does exactly this:

```
  book = Book(*line)
```

So the loading function could be simply:

```
class BookList(object):
  "Holds the list of books"

  def load_csv(self, filename):
    "Read the CSV file and create books"
    books = []
    reader = csv.reader(open(filename))
    for line in reader:
      books.append(Book(*line))
    self.books = books
```

Task 1: Filtering out claimed books

Filtering a list is a very simple task in Python. Especially when we are accepting or rejecting values with a simple boolean test.

Listing 3

```
def available_books(self):
  return [book for book in self.books if not
     book.reviewer] ①
```

Note on Listing 3

① The **[...]** notation is called list comprehension. It provides a simple syntax for constructing a list from another list. It does not add any functionality into the language that was not there prior, but does make it syntactically shorter and more understandable. The list comprehension statement here could also be written long hand as:

```
result = []
for book in self.books:
    if not book.reviewer:
        result.append(book)
return result
```

Task 2: Books that have been hanging around unclaimed

We could tackle this problem a number of ways, but the easiest is to work with some form of date object. Currently the **date_added** attribute of our **Book** class is a string. Again the Python library comes to the rescue with the **datetime** module. The **datetime.date** class is great for interacting with, but mildly a pain when converting from a string to a date. The **datetime.date** class is constructed with a year, month and day. A disadvantage of loosely typed languages is that you can't overload on parameter type. One way that python resolves this is to provide class level factory functions like **datetime.date.fromtimestamp**. In order to get from a string to something that we can use to construct a date object we can either handle the parsing of the string ourselves or use a library function.

```
>>> book.date_added
'02/10/2006'
>>> import time
>>> time.strptime(book.date_added, '%d/%m/%Y')
(2006, 10, 2, 0, 0, 0, 0, 275, -1)
```

Readers who are familiar with the C function **strptime** should notice a little familiarity here. The resulting tuple from **time.strptime** is year, month, day, hour, minute, second, week day, year day, and a daylight savings value. For the week day, Monday counts as 0, and after checking my calendar I can confirm that the 2nd of October 2006 was indeed a Monday.

The constructor for the **Book** class can be modified slightly as shown in Listing 4.

Listing 4

```
date_added = time.strptime(
   date_added, '%d/%m/%Y')[:3] ①
self.date_added = datetime.date(*date_added) ②
```

Notes on Listing 4

① We are only interested in the year, month and day part of the return value.

② Again the list expansion for parameters is handy.

To identify the older books is now just a matter of looking for available books that were added earlier than a specified date.

Listing 5

```
def older_books(self, day_count):
  cut_off_date = (datetime.date.today() -  ①
                 datetime.timedelta(day_count)) ②
  return [book for book in
           self.available_books()③
          if book.date_added < cut_off_date] ④
```

Notes on Listing 5

① **today** is a class function for returning a **datetime.date** instance for the current local time.

② You cannot subtract an integer from a date object, just other date objects (which yield a **timedelta** object) or **timedelta** objects (which yield other dates).

③ The source list with list comprehension can just as well be a return value from another function.

④ date1 < date2 is true if date1 precedes date2 in time.

The Two Sides of Recruitment

Simon Salter and Emily Winch look at recruitment at CherSoft.

Part 1 – by Simon Salter

Fresh out of college I applied for a job with a large company, ICL, to be a designer. I liked designing things, always have, and I thought designing large gate arrays in a big company might be for me. They invited me for an interview. I was to travel there the day before and stay at a hotel. They laid on a dinner event for all the candidates. We got to meet some of the ICL people, saw a short presentation and languished in the free bar. As an impoverished student this was great. Even worth cutting my hair for. Nice room. Good food. Chatted with the other candidates until quite late.

Next day I was given a pile of tests to do. Not technical stuff but strange psychometric things designed to reveal my true persona. After lunch, which was also pretty good, they asked if I wanted to be a Production Engineer. So I said no and went home.

Now I quite enjoyed the visit to ICL but came away with the general notion that although they had put a lot of time, money and effort into recruitment that they had somehow missed the point. I felt as if I'd been processed by a system and found lacking.

Much later I applied to the British Antarctic Survey to work at one of their research bases. Two years, one small base, eighteen people. This time I expected psychometric tests and other sorts of physiological evaluations. But there was none of it. Instead they spent a lot of time carefully explaining just what I was letting myself in for. The basic premise was that if I understood them, how they worked, what would be expected of me, what life is like down South and then still wanted to go I'd probably be fine. You can't just up and leave from an Antarctic base if you decide you don't like it. You are taken there and then the ship comes back a year later for a visit and then a year after that to take you home. That is pretty much it so far as commuting possibilities go. So picking the right people is really important. What I realised was that the recruitment process has to be a two way process. If someone is right for the job then quite often the job is right for them as well.

SIMON SALTER AND EMILY WINCH

Simon is a software developer who created CherSoft in 1994 to provide navigation software to the marine market. He can be contacted at simon@chersoft.co.uk.

Emily is mostly a software developer, but sometimes she writes and gives training courses, and sometimes she juggles flaming torches. Emily can be contacted at emilyw@chersoft.co.uk

The Book List (continued)

Task 3: Finding overdue reviews

The fundamental reasoning behind giving out books is to get reviews in return. Giving out books and not getting reviews is as good as standing at the corner of an intersection handing out money to passers by. What we want is some way to easily find out who has had which book for a long time (for some fairly arbitrary value of "long").

Firstly we need to change our book class again to have the date_claimed attribute also a date instance. Since there is the possibility that there isn't a value, we need to check that too.

Listing 6

```
def __init__(self, title, author, publisher,
             isbn, pub_year, date_added,
             reviewer=None, date_claimed=None):
  # snipped rest of the constructor
  if date_claimed:
    # string -> tuple  ①
    date_claimed = time.strptime(date_claimed,
                                 '%d/%m/%Y')[:3]
    # tuple -> date
    date_claimed = datetime.date(*date_claimed)
 self.date_claimed = date_claimed  ②
```

Notes on Listing 6

① One advantage of a dynamically typed language is the ability to change the type of a variable. Here we go from a string that contains a possible date value, to a parsed tuple, and from that to the date instance.

② The date_claimed value here is either None (or other false value passed in), or a valid date instance.

All we have to do now is to do more date arithmetic to find out how long the people have had their books. Listing 7 contains a couple of additional methods added to BookList.

Listing 7

```
def claimed_books(self):
  return [book for book in self.books
          if book.reviewer]

def books_with_overdue_reviews(
   self, review_time_in_days=60):
  today = datetime.date.today()①
  review_time = (
     datetime.timedelta(review_time_in_days))
  return [book for book in self.claimed_books()
    if (today - book.date_claimed) > review_time]
```

Note on Listing 7

1. today is a static method on for the class datetime.date. Similar to a C++ static function of a class.

Next time

We now have a handy script that gives us a few useful functions that we can play with, but it is far from really useful. The next steps to look at will be one or more of the following:

1. Reading and writing Open Doc format documents rather than the CSV files (dealing with zip files and XML reading and writing).
2. Adding and removing books from the list in an easy way. You don't always want to have to type in all the details of the books, especially when there is a web based API that you can use to get all the book details just from the ISBN number (using simple HTTP web services, XML parsing for the results).
3. Automating emails for those naughty reviewers that have been a little lax in getting the reviews in on time (python's email libraries).

Perhaps after all this we might even have something that will be really used. ■

The two years in Antarctica were great and now, a few more years down the road, I am part of a small software company. I'm involved with recruiting again but this time from the other side of the desk. Obviously I am very keen to learn from my earlier experience. Well, I think its obvious anyhow. The company philosophy is very much concerned with professionalism and quality. To this end we view employing a programmer as a long term proposition. Immediately this seemed to put us at odds with much of the industry so, like the good programmers we are, we decided to devise our own recruitment strategy.

Here is a simple recruitment algorithm: find a candidate, test for technical competency, test for ability to fit into company. What I find interesting about this algorithm is that the three steps are not independent. In fact, for us, it is only the last point that is the real test and everything else is just working up to it.

On a daily basis we receive many CVs via agencies. These do not even get read. This is not just because the agencies are parasitic warts on society but mainly because we want people that can think for themselves and are self-motivated. Writing a CV and searching the Internet is not difficult. It's not difficult if you have the independence of thought and creativity to be a good programmer. Certainly it is not so difficult as to justify costing a third of your first year's salary. There is really just the one pot of money from which we are going to pay recruitment costs and salary. So if the recruitment costs are high – guess what?

CVs are great fun. You would not believe some of the daft things people write. Personally I am not too interested in subjective stuff like 'I am a great guy and everyone likes me'. It is too easy to say and rarely has any supporting evidence. It may be true but really we're going to have to find that out for ourselves. Instead I look for hard facts. What and when. Basic qualifications and experience. Why are you looking for a job just now? So you have 12 years experience C++, 3 of Fortran and 6 years on some assembler that I never heard of. This does have some value but if I add up all the years experience and deduce that you must be well over a hundred years old then the value gets diluted a bit. Some personal information can be useful. If you are going to work here then we are going to have to like you as well. Strange hobbies are fine. No hobbies is a bit worrying. Occasionally we receive a CV which states the candidate's IQ. I don't think this is a very intelligent thing to do.

Technical competency. What makes a good programmer? Well the easy answer is that I don't know. However, one way to spot a good programmer is that they write good programs. This is the basis of the next stage in our recruitment process. Please will you write us a program? Only a short one. Just a few hours work. We've been doing this for years and find it incredibly useful. Software, good software anyhow, requires a strong creative input. So when you write code you always put something of yourself in it. I think you can often tell a lot about someone from looking at the code they write. Our coding test is pretty straightforward: write a PIN generator. However, as with most programming problems, there are many ways to tackle this. Over the years we've seen many from the brilliantly clever to the downright ludicrous. However, what we also see is something of the way that someone tackles a problem. How they think almost. We can tell a lot about how well someone will fit in with the company by looking at their code.

Occasionally we receive a CV which states the candidate's IQ. I don't think this is a very intelligent thing to do.

The best code was that which fitted the way we work. No big surprises there. However I do mean the way we work and not the elusive 'good code'. We didn't expect code which would immediately fit into our libraries but we felt we could spot code which showed that potential. Once you accept that crafting code is creative and individual then you can start to look beyond issues such as correctness and efficiency and see something deeper about the person that wrote it. I can usually spot who wrote what code in CherSoft just from the style.

So by this stage we have ascertained that the candidate has suitable training and experience. Also that they can write good code and we like the way they tackle problems. All that really remains is to decide whether we like them or not. This is not the one sided thing that it sounds like. For the most part it consists of doing our best to explain who we are and what we do. We'll introduce them to other people in the company and talk about working conditions, holidays and that sort of stuff. Programmers are a funny bunch and often have strong opinions about working conditions. The really good programmers are typically far more concerned with the what and how of working than with salary. Sometimes I think this part of the recruitment process might be best done down the pub over a few beers. Eventually we all go off and make a decision. Not immediately. It seems best to mull over things for a few days.

Strange hobbies are fine. No hobbies is a bit worrying.

This works for us. We do get and keep the people that we want. Including Emily.

Part 2 – by Emily Winch

At the beginning of 2006, CherSoft recruited me. I was particularly impressed by their novel approach to the task of recruitment, and how it worked well for me as an applicant as well as for them as an employer.

Not only were they able to find out a lot about me before committing to a job offer – they helped me to find out a lot about them. Unlike with most other job applications, I was happy that I had enough information to make my own decision – do I want this job or not?

I first heard of CherSoft when I posted my CV to the accu-contacts mailing list, and they replied. Since it's not possible to read accu-contacts without first joining the ACCU, I knew that at least one person at CherSoft was a member – which told me that the company values good engineering and continuing professional development. The fact that they were handling recruitment internally rather than using agencies told me that they saw it as a core part of the business rather than an ancillary function to be outsourced.

Of course, as soon as the email arrived through accu-contacts, I went to look at the CherSoft website. Unusually for a corporate site, it went into some detail about the company's philosophy and way of working – while still seeming as though it could have been written by a human being instead of a Marketing Department. By now, I was doubly convinced that it was a good idea to apply for the position, despite a long commute which would ordinarily have discouraged me altogether.

When I did apply, I was asked to write a short program for them. This impressed me for two reasons. Firstly, it was evidence that when CherSoft said that they valued good engineering, they really meant it. Secondly, it gave me confidence in the ability of the rest of the team. Any idiot can read a book about C++ and learn enough to answer a couple of questions about virtual functions. I don't want to work with idiots!

Finally, after I'd submitted my program and various CherSoft staff had inspected it, I was asked to an interview. By this time, I had enough information about the company to be fairly sure that I wanted the job. Even so, Simon was very careful to explain the company culture and values during the interview. I've heard plenty of managers give fantastic-sounding speeches before, only to find out that they were so much hot air – but in this case, the fact that Simon is a director of the company gave me some confidence that the values he was describing were those of the company itself, rather than of a manager with the right ideas but without the power to implement them.

By the time CherSoft offered me the position, I felt that I had all the necessary information to make the decision – Is this company right for me? Would I enjoy working there?

Reader, I took the job. ■

A Python Gotcha

Silas Brown revisits an old problem and identifies a solution.

In C Vu 15.2 (February 2003), *A Python Project*, I wrote code that included the following:

```
def overlaps(self,start,schedule,direction):
  if self.invisible: return 0
  if not schedule.bookedList: return 0
  count = 0
  # Skip over all events that finish before we start
  while schedule.bookedList[count][1] <= start:
    count = count + 1
    if count >= len(schedule.bookedList): return 0
  # Does this event start before we finish?
  if (schedule.bookedList[count][0] <
      start + self.length):
    # We have an overlap
    if direction < 0:
      # Make sure we finish before it starts
      backwards = (start + self.length -
                   schedule.bookedList[count][0])
      return self.overlaps(start - backwards,
          schedule, direction) + backwards
    else:
      # Make sure we start after it finishes
      forwards = schedule.bookedList[count][1]-start
      return self.overlaps(start + forwards,
          schedule, direction) + forwards
  return 0 # No overlap
```

This code proved to be the slowest thing in the application by far – all those recursive calls were duplicating a lot of work that had already been done (skipping over irrelevant events had to be done all over again for each recursive call). As the application ran too slowly, I re-wrote the above using a loop instead:

```
def overlaps(self,start,schedule,direction):
  # Returns how much it has to move, given 'start'
  # direction is +1 or -1 - which way to move
  # (return value is always unsigned)
  if self.invisible: return 0 # never has to move
  bookedList = schedule.bookedList
  if not bookedList: return 0
  count, toMove = 0, 0
  if direction == -1:
    count = len(bookedList) - 1
  while True:
    if direction == 1:
      while bookedList[count][1] <= start:
        # finishes before or as we start
        # - irrelevant
        count += 1
        if count >= len(bookedList):
          return toMove
      if (bookedList[count][0] <
          start + self.length):
        # starts before we finish
        forwards = bookedList[count][1] - start
        toMove += forwards
        start += forwards ; continue
    else:
      while (bookedList[count][0] >=
             start + self.length):
        # starts after or as we finish - irrelevant
        count -= 1
        if count < 0: return toMove
      if bookedList[count][1] > start:
        # finishes after we start
        backwards = (start + self.length -
                     bookedList[count][0])
        toMove += backwards
        start -= backwards ; continue
    return toMove
```

A bit longer, but much faster. But there is a subtle problem with the above when it is dealing with floating-point numbers that can cause an infinite loop, and it took me a long time to spot it. (I knew that the program sometimes crashed and an interrupt showed that it was in this method, but I wrongly assumed that the fault was in code that repeatedly called this method rather than in the method itself.) In the line

```
  backwards =
    start+self.length-bookedList[count][0]
```

there is a subtraction that calculates the difference between `start+self.length` and `bookedList[count][0]`. It could be that this difference is very small (say, 1e-15) and this is more likely than you think if lots of similar events are being added. A couple of lines later, that difference is subtracted from `start`, and it is not impossible that floating-point rounding will be such that such a small number makes no difference at all to the value of `start`, which will mean the next iteration of the loop will try to do the same thing and so on, creating an infinite loop. It seems that Python under Windows is somehow more susceptible to this than the Linux version, perhaps because of different default precisions.

(At least, my app crashed much more frequently under Windows than Linux, which was slightly annoying because I had to battle with a Windows box before I could reliably reproduce the problem.)

In this application it suffices to add the hack

```
  backwards = max ( backwards, 0.1 )
```

(suitably commented, of course), but in general you should avoid taking the difference between two possibly-similar floating point numbers and then trying to add this to some other number on the assumption that the addition will definitely change the number – if it's non-zero but small then the larger number might not be changed at all.

(Note: The line dealing with `forwards` is less likely to go wrong, because the small difference is then added to one of the arguments of the subtraction, rather than to some other number.

However, it's probably a good idea to fix that one too.)

(Note 2: This bug was present in the old version of the code also, but in that version you'd get a stack overflow rather than an infinite loop.) ■

SILAS BROWN

Silas is partially sighted and is currently undertaking freelance work assisting the tuition of Computer Science at Cambridge University, where he enjoys the diverse international community and its cultural activities. Silas can be contacted at ssb22@cam.ac.uk

Adventures in Autoconfiscation #2

Jez Higgins steps us through becoming autoconfiscated.

Last time [1] I sketched, in rather breathless terms, a brief history of my XML toolkit Arabica [2] and its evolution. I discussed why I decided to replace Arabica's wobbly build system with something more reliable. The "something more reliable" was, I declared despite a previous failed attempt some years ago, GNU Autotools. In this article, I describe how I made the change and examine whether it really did do what I'd hoped – let more people build Arabica on more platforms, more easily but with less fuss and less effort on my part.

This isn't the definitive guide to Autoconf, it's the how-I-did-it narrative which I hope will inform and entertain.

An Autoconf, Automake, Libtool Fly-past

So what is Autotools? Autotools is actually three things – Autoconf, Automake, and Libtool. As far as I can tell Autotools isn't an official name for this little trinity, and while you can use one without the others, in reality nobody does. Autoconf, Automake, and Libtool provide the magic behind the familiar "./configure; make; make install" incantation.

When you run `./configure`, you're running a shell script which probes your system for various bits and pieces. With the information gathered, it processes a number of templates to generate output files. Those files are usually (but not necessarily exclusively) Makefiles, and a header file commonly called `config.h`.

Autoconf is the tool that generates the configure script. The autoconf command looks for a file called `configure.ac`, processes the macros it contains and a fresh configure script pops out the end. `Configure.ac` macros are written in a language called m4 [3]. m4 is probably unfamiliar, but isn't difficult to get the hang of. Autoconf includes an extensive library of macros and writing your own is straightforward.

Automake is a Makefile generator. Well, almost... it's a Makefile template generator. Automake looks for a file called `Makefile.am`, from which it creates a `Makefile.in`, based on the macros it finds. Later, configure will use the `Makefile.in` to generate a Makefile.

Libtool looks after the oddities of building, linking, and loading static or shared libraries. It takes care of invoking the compiler and linker properly, as well as installing libraries and binaries which use the libraries according to your platform's conventions. It will, for example, relink after installation if required. In my experience, once you've added the appropriate macro to `configure.ac`, you don't actually have to do anything with Libtool directly. Configure will build a shell script called **`libtool`**, specific to your setup, which your Makefiles will invoke as and when.

In addition to the **`autoconf`** and **`automake`** commands I've already mentioned, the various Autotools packages include several other commands [4]. They manage some of the other support files used to generate configure, or when configure is run. Changes to `configure.ac` or a `Makefile.am` might require any or all of these commands to be run, and in the correct order. Fortunately, there's an uber-command to manage all that for you, **`autoreconf`** [5].

I may not have made it entirely clear, but Autotools are developer tools. Once the `configure` script and `Makefile.in` files have been generated, they become part of the source distribution. People building your source obviously need have a compiler, but they do not themselves need Autotools.

That's the overview, and is as much (in fact slightly more) than I knew when I started converting Arabica from its wobbly collection of Makefiles to Autotools.

Building Arabica

The Arabica package includes the Arabica library itself, several test executables and a number of sample applications. Clearly since the executables use the library, the library must be built first. It requires either the Expat, libxml2, or Xerces XML parser, and needs to include different source files according to the parser used. Executables using Arabica's XPath or XSLT engines need a recent version of Boost [6]. The test programs can be built in both narrow and wide string versions, although not all platforms [7] support wide strings.

As prerequisites and build options go, Arabica isn't particular extreme, but it's awkward enough to have played a part in prompting my move to Autotools.

The Arabica source lay out is relatively conventional:

```
/
/include
/src
/tests/SAX
      /DOM
      /XPath
      ....
/examples/SAX
         /DOM
         /XPath
         ...
```

Arabica is primarily implemented in header files, so the bulk of the source sits in `include`. The library source of around 20 files, mainly code conversion facets, sits in `/src`.

Starting small

After discarding my existing Makefiles, I had a big pile of source and no way to build it. My initial goal was to use Autotools to build the Arabica library. I wasn't going to worry about prerequisites, I'd simply assume they were there, nor would I worry about building anything else.

I spent a little time, only an hour or so, reading the `configure.ac` and `Makefile.am` files used in a few packages I'd recently built from source. Not surprisingly, I found smaller library packages like Expat more informative than the enormous tool suites like GCC. Finally I plugged `configure.ac` into Google, which pointed me to the Autoconf manual [8]. Reading the manual hadn't helped a great deal last time around, but both the manual and my comprehension skills have increased since then.

The manual states that the order in which `configure.ac` calls macros is not generally important, but must contain a called to **`AC_INIT`** to get started and a called to **`AC_OUTPUT`** at the end. We'll come to those in a moment. The suggested `configure.ac` layout is

```
configure.ac:
  AC_INIT(package, version, bug-report-address)
  information on the package
  checks for programs
```

JEZ HIGGINS

Jez works in his attic, living the on-and-off life of a journeyman programmer. Before work, he swims; after work, he walks the dog. In April, he became ACCU Chair. His website is http://www.jezuk.co.uk/

```
  checks for libraries
  checks for headers
  checks for types
  checks for structures
  checks for compiler characteristics
  checks for library functions
  checks for system services
  AC_CONFIG_FILES([file ...])
  AC_OUTPUT
```

That's a lot of checks to think about, which wasn't what I really wanted to do. So what's the smallest `configure.ac` you can write?

```
configure.ac:
  AC_INIT([Arabica], [Jan07], [jez@jezuk.co.uk])
```

Let's try that.

```
  $ autoreconf
  $ ./configure
```

It worked! It worked in as much as I got a configure script, and the script ran. Note that the M4 quote characters are `[` and `]`.

Since I'm building a library written in C++, I need to initialise Automake, find a C++ compiler, and set up libtool.

```
configure.ac:
  AC_INIT([Arabica], [Jan07], [jez@jezuk.co.uk])

  AM_INIT_AUTOMAKE

  AC_PROG_CXX
  AC_PROG_LIBTOOL
```

Macros starting **AC_** are Autoconf macros while macros beginning **AM_** are Automake macros, described in the Autoconf or Automake manuals [9]. The exception is **AC_PROG_LIBTOOL**, which is the macro to setup Libtool [10].

Running that gave the output in Listing 1 ...

Ah-ha. Not so good. The missing files are shell scripts used by configure during a build. They are included in the Automake package, and running **automake --add-missing** pulls them into your source tree by adding symbolic links. I take the view that symbolic links that point out of a source tree really don't mix with sensible source control procedures, and so removed the symlinks and copied the files across.

I created an empty `Makefile.am`, and re-ran **autoreconf**. This produced Listing 2.

Well, I didn't forget, I just hadn't yet. See Listing 3.

AC_CONFIG_FILES(file) primes the **AC_OUTPUT** macro to create 'filename'.

The file is created by copying an input file (by default, `filename.in`), substituting variables as it goes. For Makefiles, automake creates the `Makefile.in` from `Makefile.am`. I don't know exactly how or when this happens, but it does. I have faith. See Listing 4.

I'm curious as to why these missing files weren't picked up last time around, but I ran **automake --add-missing** anyway. Since Autotools were developed for the GNU project, they expect certain things – **NEWS**, **README**, etc – to conform to GNU standards. It is possible to relax this requirement [11], but I didn't bother. Be aware that the **COPYING** file automake creates for you is the GNU General Public License, as you may want to substitute another license.

It took me a while to find where `ltmain.sh` should come from. I discovered that **autoreconf** also has an option to create or copy the missing support files.

```
$ autoreconf --install
$ autoreconf
$ ./configure
checking for a BSD-compatible install...
 /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /usr/bin/
 mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for g++... g++
... 85 other lines snipped ...
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
$ make
make: Nothing to be done for 'all'.
```

Well, that looks almost convincing. Now to get it to build something.

Listing 1

```
$ autoreconf
configure.ac:6: required file `./config.sub' not found
configure.ac:6: `automake --add-missing' can install `config.sub'
configure.ac:3: required file `./missing' not found
configure.ac:3:   `automake --add-missing' can install `missing'
configure.ac:3: required file `./install-sh' not found
configure.ac:3:   `automake --add-missing' can install `install-sh'
configure.ac:6: required file `./config.guess' not found
configure.ac:6:   `automake --add-missing' can install
`config.guess'
automake-1.10: no `Makefile.am' found for any configure output
autoreconf-2.60: automake failed with exit status: 1
```

Listing 2

```
$ autoreconf
automake-1.10: no `Makefile.am' found for any configure output
automake-1.10: Did you forget AC_CONFIG_FILES([Makefile]) in
configure.ac?
autoreconf-2.60: automake failed with exit status : 1
```

Listing 3

```
configure.ac:
  AC_INIT([Arabica], [Jan07], [jez@jezuk.co.uk])
  AM_INIT_AUTOMAKE
  AC_PROG_CXX
  AC_PROG_LIBTOOL
        AC_CONFIG_FILES([Makefile])
        AC_OUTPUT
```

Listing 4

```
$ autoreconf
Makefile.am: required file `./INSTALL' not found
Makefile.am:   `automake --add-missing' can install `INSTALL'
Makefile.am: required file `./NEWS' not found
Makefile.am: required file `./README' not found
Makefile.am: required file `./AUTHORS' not found
Makefile.am: required file `./ChangeLog' not found
Makefile.am: required file `./COPYING' not found
Makefile.am:   `automake --add-missing' can install `COPYING'
configure.ac:6: required file `./ltmain.sh' not found
autoreconf-2.60: automake failed with exit status : 1
```

Getting the library built

So far I've been working the root directory of my source tree. The Arabica library source sits in the `src` subdirectory. I've no particular aversion to recursive Makefiles, so I want my top level Makefile to call `src/Makefile`. Since automake is creating my Makefiles, I'll need `Makefile.am` and `src/Makefile.am`.

The top level `Makefile.am` is straightforward. Subdirectories are specified using the **`SUBDIRS`** variable. You can specify a number of directories, and they will be visited in the order given. The subdirectories don't have to contain `Makefile.am`, only `Makefile`, which allows third-party party packages to be included in the build. At the moment, I'm only interested in the one subdirectory.

```
Makefile.am:
  SUBDIRS = src

Running autoreconf, configure and make -
  $ make
  Making all in src
  make[1]: Entering directory '/home/jez/arabica/
   src'
  make[1]: *** No rule to make target 'all'. Stop.
  make[1]: Leaving directory '/home/jez/arabica/
   src'
  make: *** [all-recursive] Error 1
```

I'm on the verge of actually building something. What goes in `src/Makefile.am`? Automake uses a combination of variables and naming conventions to describe what a Makefile should build. A variable which tells automake what is being built is called the primary. The **`_LTLIBRARIES`** primary declares libraries that are built with libtool.

```
src/Makefile.am:
  lib_LTLIBRARIES = libarabica.la
  libarabica_la_SOURCES = arabica.cpp \
              SAX/helpers/InputSourceResolver.cpp \
              ... other source files ...
```

Automake defines a number of Makefile targets automatically, including install. The lib prefix on the **`_LTLIBRARIES`** primary says that the library should be installed in Automake's libdir. Libdir usually points to the system's library path (e.g. `/usr/local/lib`), unless overriden by passing an option to `./configure`. Other prefixes include **`pkglib`** and **`noinst`**, to install to a package specific directory or to mark the library as not installed, respectively.

Note how the name of the library to build becomes the prefix of the **`_SOURCE`** variable giving its source files. Here, I've listed the C++ source files located in the `src` directory, but the library also uses a number of header files from the `include` directory. A large number, actually, which I was too lazy to list in its entirety. Instead I decided simply add the `include` directory to the compiler include path. Additional flags can be passed to the compiler (or more correctly the preprocessor) using the **`AM_CPPFLAGS`** variable.

The backslash character \ is, as in normal Makefiles, the line continuation character.

```
src/Makefile.am:
  AM_CPPFLAGS = -I$(top_srcdir)/include  [12]

  lib_LTLIBRARIES = libarabica.la
  libarabica_la_SOURCES = arabica.cpp \
              SAX/helpers/InputSourceResolver.cpp \
              ... other source files ...
```

Now the `src/Makefile.am` looks complete, the final job is list it `configure.ac` so that `src/Makefile` will be created.

```
configure.ac:
  AC_INIT([Arabica], [Jan07], [jez@jezuk.co.uk])

  AM_INIT_AUTOMAKE

  AC_PROG_CXX
  AC_PROG_LIBTOOL

  AC_CONFIG_FILES([Makefile])
  AC_CONFIG_FILES([src/Makefile])
  AC_OUTPUT
```

After another round of **`autoreconf`**/**`configure`**/**`make`** now gives Listing 5.

Goal!

It's worked. The library has built. I actually became momentarily light-headed when this happened. Even though I've been using **`make`** and writing Makefiles for years now, I generally start a new Makefile by copying an existing one because I don't usually get them right from a standing start. And here I was, after only an afternoon's work, with a

Listing 5

```
  $ make
  Making all in src
  make[1]: Entering directory `/home/jez/work/arabica'
/bin/sh ../libtool --tag=CXX   --mode=compile g++ -DPACKAGE_NAME=\"Arabica\" -DPACKAGE_TARNAME=\"ara
bica\" -DPACKAGE_VERSION=\"Jan07\" -DPACKAGE_STRING=\"Arabica\ Jan07\" -DPACKAGE_BUGREPORT=\"jez@jez
uk.co.uk\" -DPACKAGE=\"arabica\" -DVERSION=\"Jan07\" -DSTDC_HEADERS=1 -DHAVE_SYS_TYPES_H=1 -DHAVE_SYS
_STAT_H=1 -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1-DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 -DHAVE_INTTYPES_H=1
-DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 -DHAVE_DLFCN_H=1 -I.  -I../include -g -O2 -MT arabica.lo -MD -MP -MF
.deps/arabica.Tpo -c -o arabica.lo arabica.cpp mkdir .libs
  ... other source files ...
  ar cru .libs/libarabica.a  arabica.o InputSourceResolver.o base64codecvt.o iso88
591_utf8.o ucs2_utf16.o ucs2_utf8.o iso88591utf8codecvt.o rot13codecvt.o ucs2utf
8codecvt.o utf16beucs2codecvt.o utf16leucs2codecvt.o utf16utf8codecvt.o utf8iso8
8591codecvt.o utf8ucs2codecvt.o XMLCharacterClasses.o
  ranlib .libs/libarabica.a
  creating libarabica.la
  (cd .libs && rm -f libarabica.la && ln -s ../libarabica.la libarabica.la)
  make[1]: Leaving directory `/home/jez/work/arabica'
  make[1]: Entering directory `/home/jez/work/arabica'
  make[1]: Nothing to be done for `all-am'.
  make[1]: Leaving directory `/home/jez/work/arabica'
```

configure script that seemed to actually be configuring and working. I'd been developing using Cygwin [13], so I verified my new configure script on Ubuntu Linux, FreeBSD, DragonflyBSD and GNU/Darwin boxes [14]. It worked and the library built on all of them. I went for a lie down.

Arabica was now a package that would build out-of-the-box on all the platforms I had to hand

Building everything else

The library was built, but did it actually work? Time to build the test suite. The build needs to recurse down into the `tests` subdirectory and again into its subdirectories. I added tests to the **`SUBDIRS`** variable in the top level `Makefile.am`, and created a `tests/Makefile.am` which specifies the next **`SUBDIRS`** level. I added extra **`AC_CONFIG_FILES`** calls to `configure.ac`.

Building a program using Autotools is very similar to building a library, but you use the **`_PROGRAMS`** primary rather than the **`_LTLIBRARIES`** primary.

```
test/Utils/Makefile.am:
  noinst_PROGRAMS = utils_test

AM_CPPFLAGS = -I($top_srcdir)/include
  LIBARABICA = $(top_builddir)/src/libarabica.la
utils_test_SOURCES = utils_test.cpp \
... more source ...
utils_test_LDADD = $(LIBARABICA)
utils_test_DEPENDENCES = $(LIBARABICA)
```

Since this is a test program and does not need to be installed, I've given **`_PROGRAMS`** the **`noinst`** prefix [15]. As with libraries, the **`_SOURCES`** variable lists the program's source files. Extra libraries that a program needs to link are given in the **`_LDADD`** variable. It is sometimes useful to have a program depend on some other target which is not actually part of that program. This is done using the **`_DEPENDENCIES`** variable. I've included libarabica as a dependency to ensure the program is relinked if the library is changed. Note how it's also possible to declare your own variables in a `Makefile.am`, in this case **`LIBARABICA`**.

So does it work?

```
  $ autoreconf
  $ ./configure
  ... stuff ...
        $ make
  ... more stuff ...
  $ test/Utils/utils_test.exe
  StringTest
  .......
  OK (7 tests)
```

It does.

It sounds silly to say it, but I felt smugly pleased with myself once I had the libraries and the test suite building. For so long, I'd found, as a user, Autotools to be a fantastic thing, because the "./configure && make && make install" incantation just worked. As a developer, I'd regarded it as a strange and scary beast. As my test cases passed, the beast was slain.

Compared to my previous build system things had already improved, because Arabica was now a package that would build out-of-the-box on all the platforms I had to hand. My page long set of build instructions [16] could be thrown away, replaced with a three item long bulletted list.

I wasn't yet at one with Autotools yet, but I was comfortable enough and now had the confidence to start extending the build to look check which XML parser was available, whether the Boost libraries were available and so on. I'll walk through some of that next time, as well as looking at some of the expected and unexpected benefits of converting to Autotools. ■

References

[1] CVU Vol 18 Number 6

[2] That's Arabica, your No 1 choice for slinging XML with C++, http://www.jezuk.co.uk/arabica

[3] See http://www.gnu.org/software/m4. m4 seems to go back to early in Unix history, but I'm not aware of it being widely used outside of Autoconf. The GNU version is still active though, with the latest release as recently as last November.

[4] autoheader, for example, which generates a skeleton `config.h`, and **`aclocal`** which builds a local copy of the various m4 macros used in `configure.ac`

[5] Actually in the course of writing this article, I discovered that the dependencies are included in the generated Makefile, so running **`make`** at the top level will also prompt a rebuild if required.

[6] http://www.boost.org/, but you knew that. Specifically it uses the fantastic parser framework Boost.Spirit, and the rather handy **`lexical_cast`**.

[7] Platform here means operating-system+compiler+library.

[8] Autoconf manual, http://www.gnu.org/software/autoconf/manual/autoconf.html

[9] Automake manual, http://www.gnu.org/software/automake/manual/automake.html

[10] Libtool manual, http://www.gnu.org/software/libtool/manual/libtool.html

[11] Strictness, http://www.gnu.org/software/automake/manual/html_node/Strictness.html#Strictness

[12] **`$top_srcdir`** is a predefined autoconf variable, which is the relative path to the top-level source directory. Autoconf and automake define quite a number of variables like this, pointing to the source directory, the build directory, and so on. Unless a variable's purpose isn't clear from its name, I won't highlight them further.

[13] Cygwin is a Linux-like environment for Windows. http://cygwin.com/

[14] Virtual boxes mostly. VMWare is a wonderful thing.

[15] Programs to be installed usually use the bin prefix.

[16] Given last time.

Developer Beliefs About Binary Operator Precedence

Derek Jones concludes the report of his experiment.

Introduction

This is the second part of a two part article describing an experiment carried out during the 2006 ACCU conference. The first part was published in a previous issue of C Vu [1]. This second part discusses the remember/recall assignment statement component of the experiment. See part 1 for a discussion of the experimental setup.

The format of the task performed in this part of the experiment is very similar to the memory for assignment statements portion of the experiment performed at the 2004 ACCU conference [2]. See the write-up of that experiment for some of the common details omitted here. That experiment attempted to measure the impact of identifier length, measured in syllables, on subjects' ability to remember assignment statement information over a short period of time. Experience with running the 2004 experiment showed that subjects sometimes used a strategy of remembering identifiers by storing information on their first letter rather than the complete identifier spelling. The identifiers used in the 2006 experiment were chosen to investigate the performance differences caused by identifiers sharing a common first letter and having a similar sounding spoken form.

The identifiers used in the 2006 experiment all contained letter sequences, having the form consonant-vowel-consonant, that could be spoken as a single syllable. For some problems all of the identifiers started with the same letter, while for other problems the initial letter of each identifier was different but the last two letters were the same.

Within commercial source code a variety of different kinds of character sequences are used for identifiers. Some are recognizable words or phrases, some abbreviated forms of words or phrases, while others have no obvious association with any known language (e.g., they may be acronyms that are unknown to the reader). Reading involves converting these character sequences to sounds and it is to be expected that subjects memories of an identifier will be sound based, rather than vision based.

Characteristics of human memory

The human *short term memory* subsystems are a gateway through which all conscious input data input must pass. They have a very limited capacity and because new information is constantly streaming through them, a particular piece of information rarely remains within them for very long. Information in short term memory is either quickly lost or stored in another, longer term, memory subsystem.

An experiment performed at the 2004 ACCU conference [2] investigated the impact on subject performance of identifiers that required more of less short term memory resources. Experience with this experiment suggested that subjects used a variety of strategies to help improve their performance. One strategy was to remember the first letter of an identifier, rather than a representation of the complete letter sequence. The identifiers used in this experiment were chosen to investigate the impact of shared letters on subject performance; they either share the same first letter on the last two letters (in this latter case the spoken forms rhyme).

The following are some of the factors that studies have been found to effect subject performance of memory for lists of information. These factors are also likely to have some impact on subject performance in this experiment.

- People pay particular attention to the initial part of a word [3] [4] (this enables them to start looking up a word in the mental lexicon while its remaining sounds are being heard).
- A decrease in word list recall performance for similar sounding words [5] [6]. It is believed that the similarity causes confusion between the various word sound sequences and a subsequent failure to correctly retrieve the original information.
- The extent to which the information to be remembered is already stored in longer term memory subsystems (i.e., known letter sequences such as words).
- The time delay between seeing the information and having to recall it (because the remembered information degrades over time),
- A capacity limit on the total amount of information that can be remembered and shortly afterwards recalled or recognized,
- For known words, their frequency of occurrence, with better performance in many tasks for high frequency words (i.e., those that have been encountered very frequently by a subject) compared to low frequency words [7].
- Neighbourhood effects [8]. Words that differ by a single letter are known as *orthographic neighbours*.
- Both the *density* of orthographic neighbours (how many there are – *mine* has 29 (*pine*, *line*, *mane*, etc.)) and their relative frequency (if a neighbour occurs more or less frequently in written texts) can affect performance.

Spotting the identifier that did not appear in the earlier list of assignment statements is a recognition problem, while remembering the value assigned in a recall problem. Studies have found that recognition and recall memory have different characteristics [9].

developers are often competitive ... some subjects ignore the work rate instruction and attempt to answer all of the problems

Ecological validity

For the results of this experiment to have some applicability to actual developer performance it is important that subjects work through problems at a rate similar to that which they would process source code in a work environment. Subjects were told that they are not in a race and that they should work at the rate at which they would normally process code. However, developers are often competitive and experience from previous experiments has shown that some subjects ignore the work rate instruction and attempt to answer all of the problems in the time available. To deter such behaviour during this experiment the problem pack contained significantly more problems than subjects were likely to be able to answer in the available time (two people did answer all problems).

DEREK JONES
Derek used to write compilers, then got involved with static analysis and now spends his time trying to figure out developer behaviour. Derek can be contacted at derek@knosof.co.uk

The structure of the problem used in this experiment follows a pattern that is often encountered when trying to comprehend source code: see information (and try to remember some of it), perform some other task and then perform a task that requires making use of the previously seen information.

Taken as a whole the constant repetition of exactly the same kind of problem rarely occurs in program development activities. The constant repetition provides an opportunity for learning to occur, i.e., subjects have the opportunity to tune their performance for a particular kind of problem. The issue of learning and problem solving strategies used by subjects is discussed below.

Generating the assignment problems

The problems and associated page layout were automatically generated using a C program and various **awk** scripts to generate **troff**, which in turn generated postscript. The identifier and constant used in each assignment statement was randomly chosen from the appropriate set and the order of the assignment statements (for each problem) was also randomized. The source code of the C program and scripts is available from the experiments web page [10].

Due to a fault in the generation script the first 10 problems for each subject all used sets of identifiers where the last two letters of each set of identifiers were the same. The intent was that the same randomisation algorithm be applied to the choice of identifiers to use for all problems.

Selecting identifiers and integer constants

A sufficient number of letter sequences were created so that subjects would rarely encounter the same sequence. In all 40 different words and 40 different nonwords were used (dues to an oversight *cub* appeared both in a set of words and a set of non-words), see the following word list. This meant that the same identifiers would start to repeat after every set of 20 problems.

The impact of different kind of letter sequences is the primary concern and we want to maximise the impact of differences due to this factor. This means minimising the impact of other kinds of information (mostly integer constants) on subject performance. A good approximation to short term memory requirements is the number of syllables contained in the spoken form of the information. Choosing single digit integer constants containing a single syllable minimises their impact on short term memory load.

software developers are continually on the look out for ways to reduce the effort needed to solve the problems they are faced with

For simplicity it was decided that identifiers would consist of a sequence of three letters following the pattern CVC. The letter sequences were grouped in sets of four such that, they were all either English words or pronounceable English non-words, and either:

- had different first letters and rhymed (i.e., in the last two letters were the same),
- shared the same first letter and did not rhyme (i.e., the last two letters did not share any common letters).

No checks were made for nonwords, in English, that might be words in other languages that might be known to subjects.

For the letter sequences used in this experiment there was no STM capacity advantage to remembering just the letter of a word. The spoken form of single letters are represented by a single syllable (except *w*, which contains two) and each of the letter sequences used was pronounceable as a single syllable. However, there is a potential advantage to only remembering the first letter when the set of identifiers *sound alike*. Using this strategy would remove the possible confusion caused by similar sounding identifiers and eliminate a potentially significant source of performance loss.

The following lists the sets of four, three letter sequences used for identifiers appearing in the assignment part of each problem. The identifiers appearing in a single row were used to create one complete assignment problem. The letter sequence *cub* was mistakenly used in both a row of words and a row of non-words.

cat	mat	hat	pat
hen	pen	men	ben
din	pin	sin	kin
hop	pop	top	mop
cub	rub	tub	hub
dat	lat	wat	gat
gen	ren	sen	cen
nin	rin	zin	cin
dop	gop	vop	rop
fub	lub	wub	cub
dad	den	dip	dog
lap	led	lip	lot
pat	peg	pin	pod
sat	sir	sow	sum
wad	web	wit	won
fep	fis	fot	fum
km	kig	kod	kus
ras	rit	roz	ruc
tep	tid	tor	tul
vek	vib	vom	vup

The nonwords have a variety of characteristics, including: *sen*/*cen* different spelling same spoken sound, *cin* sounds like *sin* a word, *fot* could be remembered as *foot* + CVC pattern, *roz* rozzer slang for policeman (at least in British English) or abbreviation for rosalyn. While these issues might be important at some level, they don't seem to have had a measurable impact on the results.

Assignment problems were created in groups of 20. Each group of 20 used one of the rows of identifiers. The identifiers used in each assignment problem were selected by randomly choosing an unused row. Three of the identifiers in the row were randomly selected to be used in the list of the three assignment statements to be remembered. The fourth identifier was used as the *not seen* identifier.

Selecting integer constants

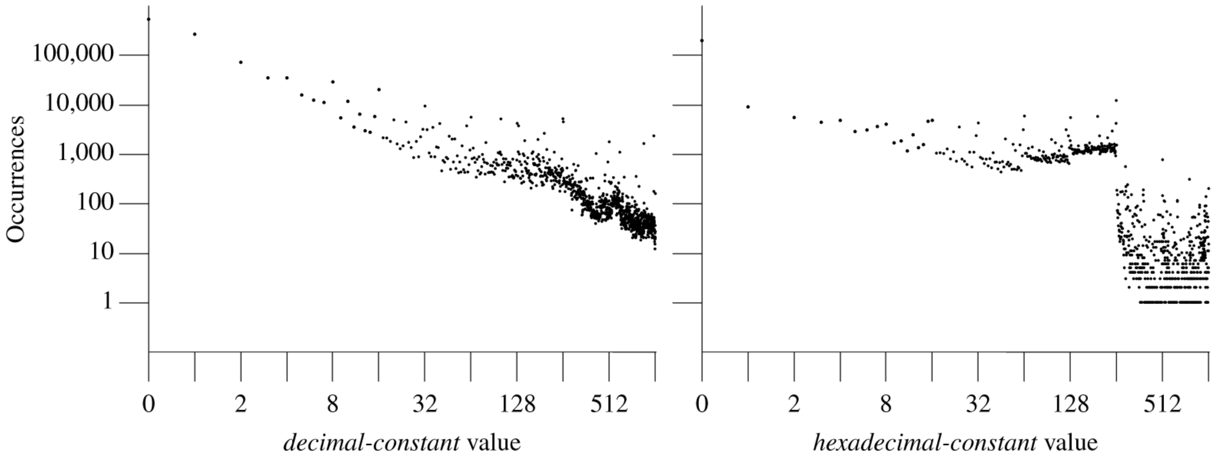
The integer constants chosen were 4, 5, 6, 8, and 9 (the digit 7 was not used because its spoken form has two syllables). They all have approximately the same frequency of occurrence in source code (it is within the same order of magnitude, see figure 1) and other contexts, and have a spoken form containing a single syllable.

Threats to validity

Experience shows that software developers are continually on the look out for ways to reduce the effort needed to solve the problems they are faced with. Because each of the problems seen by subjects has the same structure it is possible that some subjects will have detected what they believe to be a pattern in the problems which they attempt to use to improve their performance.

While the general format of problem used commonly occurs during program comprehension, the mode of working (i.e., paper and pencil) does not. Source code is invariably read within an editor and viewing is controlled via a keyboard or mouse. Referring back to previously seen information (e.g., assignment statements) requires pressing keys (or using

Figure 1



Occurrences, in the visible form of various applications written in C, of integer constants having specific values [11].

a mouse). Having located the sought information additional hand movements (i.e., key pressing or mouse movements) are needed to return to the original source location. In this study subjects were only required to tick a box to indicate that they *would refer back* to locate the information. The cognitive effort needed to tick a box is likely to be less than would be needed to actually refer back. Studies have found [12] that subjects make cost/benefit decisions when deciding whether to use the existing contents of memory (which may be unreliable) or to invest effort in relocating information in the physical world. It is possible that in some cases subjects ticked the *would refer back* option when in a real life situation they would have used the contents of their memory rather than expending the effort to actually refer back.

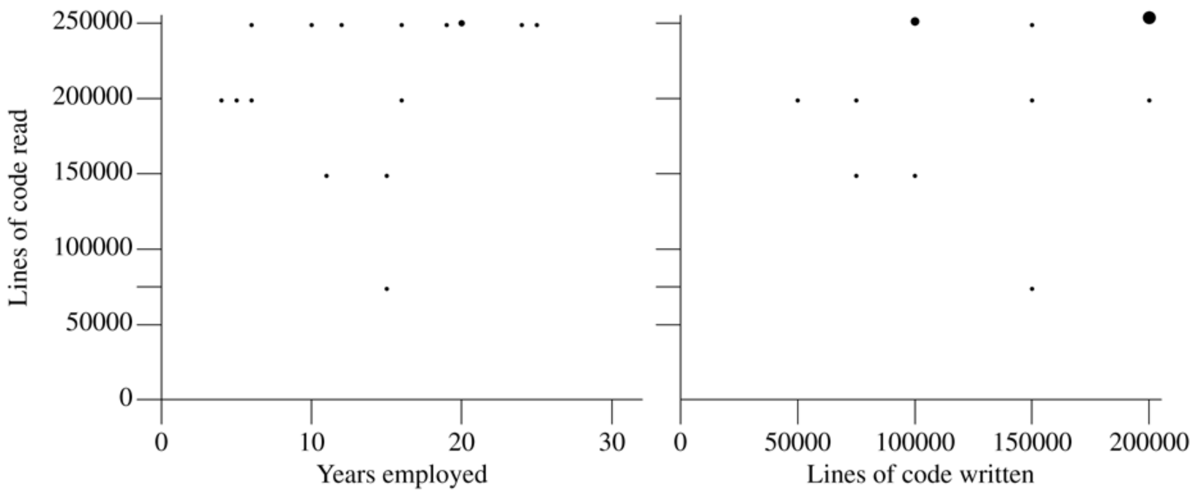
While subjects were told that they are not in a race and that they should work at the rate at which they would normally process code, it is possible that some subjects followed this request and some did not. A consequence of this is that the distribution in the numbers of problems answered, and perhaps the accuracy of the results, may be different than would have occurred if all subjects had reacted in the same was to the instructions.

Results

It was hoped that at least 30 people (on the day 18) would volunteer to take part in the experiment and it was estimated that each subject would be able to answer 20 problem sets (on the day 23.8) in 20-30 minutes (on the day 20 minutes). Based on these estimates the experiment would produce 600 (on the day 429) individual answers.

A total of 429 sets of assignment statements were remembered/recalled giving a total of 1,716 answers to individual assignments. The average number of individual answers per subject was 95.3 (standard deviation 38.8), the average percentage of answers where the subject would refer back was 26.3% (standard deviation 26.7), and the average percentage of incorrect answers 8.9% (sd 9.5).

Figure 2



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

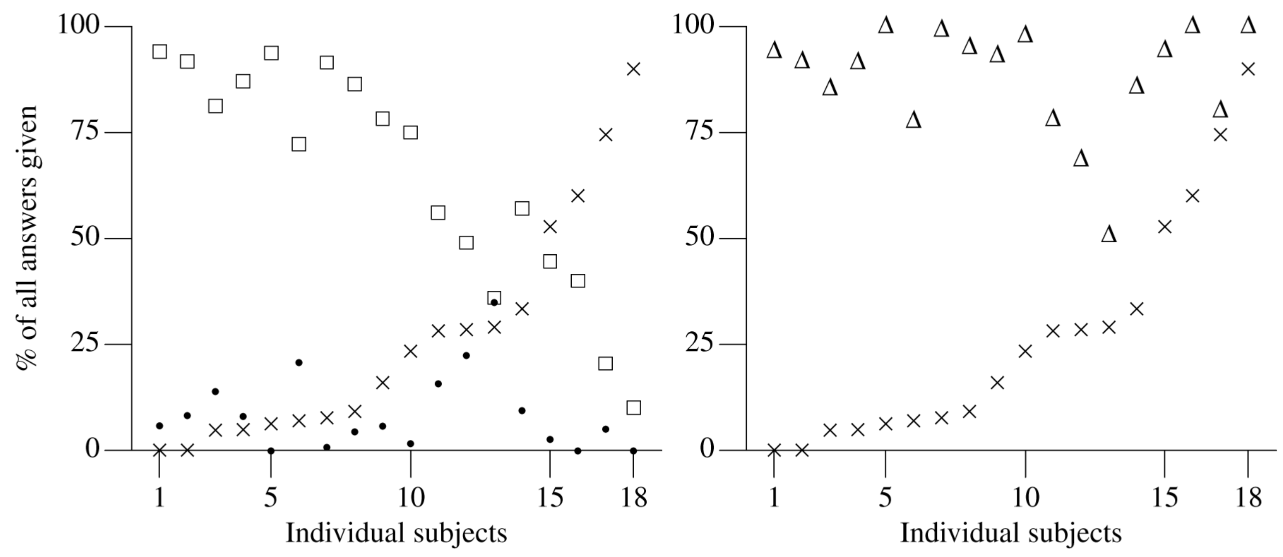


Figure 3

Number of problems answered by the 18 individual subject against the percentage of different kinds of answers. Left plot: cross for *would refer back*, bullet for incorrect answers and box for correct answers; right plot: cross for *would refer back*, triangle for percentage of correct answers for all cases where a numeric answer was given (i.e., *would refer back* answers were excluded from the percentage calculation). In both plots subjects, on the x-axis, are ordered by their *would refer back* percentage.

The average amount of time taken to answer a complete problem was 50.4 seconds. No information is available on the amount of time invested in trying to remember information, answering the parenthesis sub-problem, and then thinking about the answer to the assignment sub-problem (i.e., the effort break down for individual components of the problem).

While STM recall performance drops very quickly after the information is no longer visible (studies have found below 10% correct within around 8 seconds in many situations [5]). Even the fastest subject took over 25 seconds per complete problem and so recency effects will be minimal.

The raw results for each subject are available on the experiments web page [10] (they are in the file `results.ans`; information on subject experience has been removed to help maintain subject anonymity).

The following subsections break down the discussion of results by individual subject and by kind of identifier used in the assignment statements.

developer interaction with source code overwhelmingly occurs in its written, rather than spoken, form

Subject experience

Traditionally, developer experience is measured in number of years of employment performing some software related activity. However, the quantity of source code (measured in lines) read and written by a developer (developer interaction with source code overwhelmingly occurs in its written, rather than spoken, form) is likely to be a more accurate measure of source code experience than time spent in employment. Interaction with source code is rarely a social activity (a social situation occurs during code reviews) and the time spent on these activities is considered to be small enough to ignore. The problem with this measure is that it is very difficult to obtain reliable estimates of the amount of source read and written by developers. This issue was also addressed by studies performed at previous ACCU conferences [13] [2].

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

Part 1 of this article contained a plot of precedence problems answered against years of experience. Given that a complete problem required subjects to answer both assignment and precedence problems this plot is actually a combined count of problems solved. No break-down is available on the time spent on the two different kinds of problem subjects were asked to answer.

Subject strategies

Discussions with subjects who took part in the 2004 experiment uncovered that they had used a variety of strategies to remember information in the assignment problem. The analysis of the threats to validity in that experiment discussed the question of whether subjects traded off effort on the filler task in order to perform better on the assignment problem, or carried out some other conscious combination of effort allocation between the subproblems. To learn about strategies used during this experiment, after 'time' was called on problem answering, subjects were asked to list any strategies they had used (a sheet inside the back page of the handout had been formatted for this purpose).

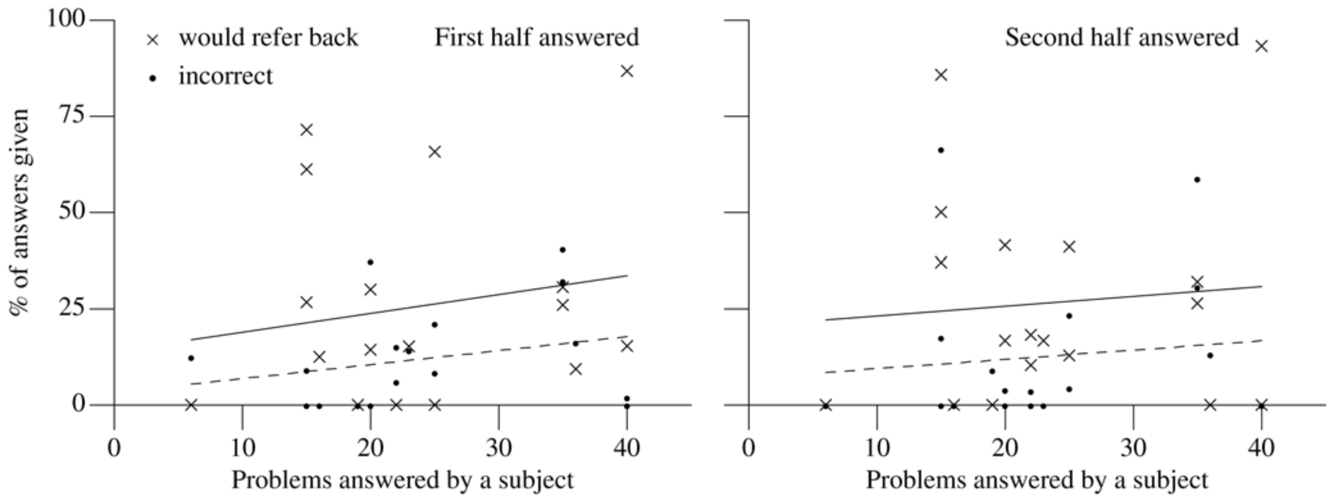
The responses given to the strategies question generally contained a few sentences. The majority of responses dealt with the assignment part of the problem, with three subjects also giving information about the precedence problem (e.g., "I always use parentheses", "... I tried to be consistent, but not very hard", "Didn't worry too much about task.").

The strategies listed consisted of a variety of the techniques people often use for remembering lists of names or numbers. For instance, number word associations, merging words into a larger word (e.g., penlenmen), reordering the sequence presented into a regular pattern (e.g., alphabetical), inventing short stories involving the words and numbers. The difficulty of integrating nonwords into these strategies was a common comment.

From the replies given it was not possible to work out if subjects give equal weight to answering both parts of the problem, or had a preference to answering one part of the problem.

No subject listed a strategy that was based on the visual appearance of the identifiers or numbers.

Figure 4



The percentage of *would refer back* answers (crosses, least squares straight line) and incorrect answers (bullet, least squares dashed line) plotted against the total number of problems answered by each subject. The left graph is based on the first half of the answers given by each subject, the right graph on the second half of the answers given. Each cross and bullet represents a single subject.

Individual subject performance

For each subject, figure 3 plots the percentage of each kind of answer they gave (in both graphs subjects are ordered by the percentage of *would refer back* answers they gave). The left plot is based on the percentages for all answers, while the triangles in the right plot are the percentage for those cases when a numeric answer was given (i.e., correct and incorrect answers only are used to calculate the percentage).

If subjects randomly guess answers to questions they cannot recall the answers to, then (given that only five possible numeric values were used and no value occurred more than once in the same problem):

- If subjects knew no answers and randomly guessed the three answers, then it would be expected that 0.7 of the three guessed questions (23%) of individual assignment questions would be answered correctly.
- If subjects knew one answer and randomly guessed the other two answers, then it would be expected that 0.5 of the two guessed answers (25%) for that problem would be answered correctly.
- If subjects knew two answers and randomly guess the last answer, then it would be expected to be correct 33% of the time.

Those subjects who gave few *would refer back* answers had a performance that was significantly better than that of random guessing. The analysis for those subjects who gave many *would refer back* answers and had a high percentage of correct answers is more complex. If these subjects randomly guessed the numeric answers they did give, the percentage correct would be very similar to that achieved when no *would refer back* answers were given. In this case the number of correct answers given by these subjects is significantly better than that of random guessing.

Possible reasons for this difference in performance, between subjects, include differences in subject's general approach to answering problems (e.g., in the case of *would refer back* the amount of risk they are willing to accept that the answer they are thinking of giving is incorrect) and differences in ability.

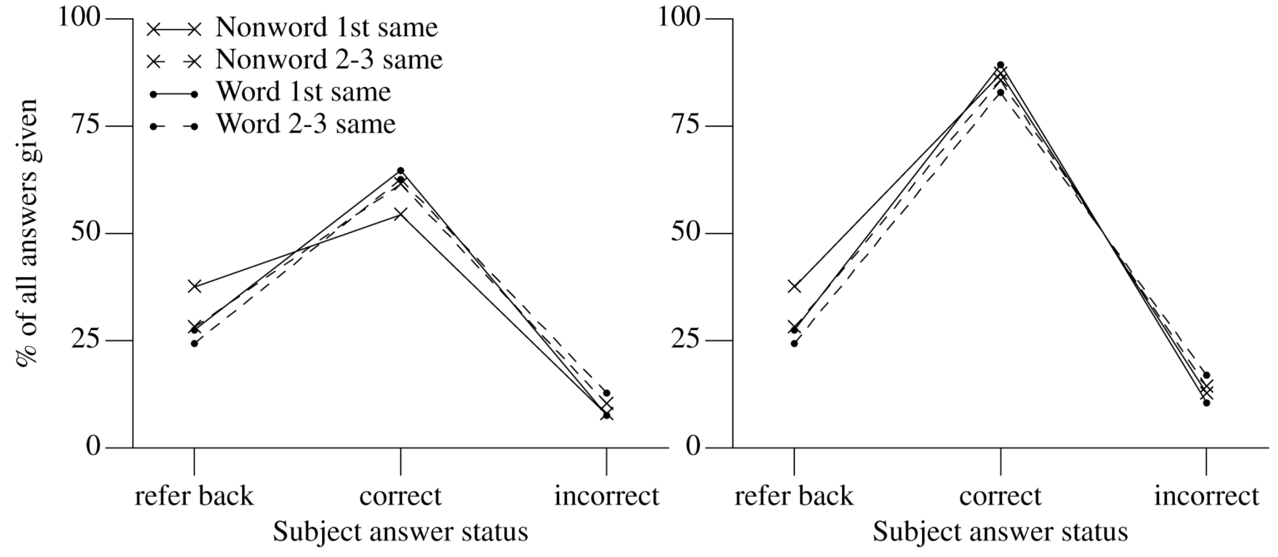
Looking at the right graph in figure 3, we see that:

- subjects 1-8 were generally certain of the answer in the sense that they gave few *would refer back* answers (less than 23%) and that this certainty was mirrored in the significantly higher than random percentage of correct answers (average 91.8%),
- subjects 9-10 gave a higher percentage of *would refer back* answers than subjects 1-8, but also had a high percentage of correct answers (95.4%),
- subject 11 might be grouped with the first 10 subjects or subjects 12-13. This subject's percentage of correct answers is very close to that of subject 6 (78% vs. 77.6%), however the number of *would refer back* answers is more than four time greater (i.e., closer to that of subjects 12-13),
- subjects 12-13 gave *would refer back* answers to just over 25% of questions and the two lowest percentage of correct answers (68.5% and 50.7% respectively),
- subject 14, like subject 11, could belong to one of two subject groupings,
- subjects 15-18 gave *would refer back* answers to over 50% of questions. While these subjects also had a high percentage of correct answers (average 93.6%), this may be because they only gave answers in those cases where they were very certain (had they taken more risk they may have given more incorrect answers, or perhaps additional correct answers).

Self-knowledge, metacognition, is something that enables a person to evaluate the accuracy of the memories they have. Subjects who gave many incorrect answers (i.e., subjects 12 and 13) did not accurately evaluate the state of their own memories of previously seen information (i.e., they overestimated the accuracy of their memories). It is also possible that subjects who gave many *would refer back* answers also showed poor metacognitive performance (i.e., they underestimated the accuracy of their memories and would have mostly given correct answers had they risked a numeric answer). However, it is not possible to make this claim from the available data.

Were different subject's performance comparable through out the experiment? Perhaps a subject who answers a greater number of questions is more likely to give incorrect or *would refer back* answers. A least squares fit of the data (see figure 4) suggests that subject's who answered more questions gave more *would refer back* responses and more incorrect answers. However, these results fail a statistical significance test at the 5% confidence level (i.e., it is not possible to claim that subjects who answered more questions gave more *would refer back* and incorrect answers).

In the 2004 experiment there was no significant difference in performance between subjects who answered a few questions and those who answered many. However, the small number of unique identifiers used in the 2004 experiment and the ordering of the assignment statements both provided an opportunity for learning to occur as subjects answered more questions. In the 2006 assignment problems there does not appear to be any



The percentage of *would refer back*, correct and incorrect answers for each kind of identifier, averaged over the individual respective percentage for all subjects. In the right plot, the percentage for correct and incorrect answers is based only on answers where a value was given (i.e., *would refer back* answers were excluded from the calculation). The vertical bars denote the standard deviation for each average (they overlap significantly because the standard deviations are all relatively large).

opportunity for subjects to improve their performance by learning as they answer more questions.

Performance changes over time

If subject performance was consistent for all problems answered, it would be expected that the percentage of correct answers for the first few problems answered would be the same as for the last few problems.

The data for figure 4 was created by dividing the answers given by each subject into two equal sized parts, i.e., the first half of the answers given by each subject and the second half of the answers given. A difference in the slope of the least squares fit would indicate that subject performance changed over time. Unfortunately these results fail a statistical significance test at the 5% confidence level and it is not possible to draw any conclusions from differences in the slope of the least squares fit.

As previously stated the first 10 problems all used sets of identifiers that followed a single pattern. It is possible that subject performance for identifiers following this pattern was sufficiently large that it biased the results for the set of *first half* answered. It is also possible that there are significant effects involving both kinds of identifiers and early/late answers and that they cancelled each other out because a random ordering was not used.

Impact of different kinds of character sequences

This experiment was designed to look for differences in subject performance for different kinds of identifiers.

Each identifier appeared once per set of 80 assignment statements. Based on expected subject performance, it was anticipated that most identifiers would be seen once, with only a few identifiers being seem twice by the faster subjects. Thus there was no opportunity for learning of individual identifiers to take place.

Figure 5 gives a break down of performance for the different kinds of identifier. While it is tempting to try and read small differences in performance from these results, the variations are swamped by differences in individual subject performance (vertical bars denote the standard deviation for each average and they overlap significantly because the standard deviations are all relatively large).

Comparison of 2006 results with 2004

How do the results of the 2004 and 2006 experiment compare? Both ran for 20 minutes and subjects completed an average of 22.7 problems in 2004 and 23.8 in 2006.

The *would refer back* percentages in 2004 were around 30%, correct answers around 60% and incorrect answers around 10%. These percentages are very close to the average percentages in 2006. Given that many of the 2004 identifiers contained three syllables (i.e., made greater calls on STM resources) the similarity between subject performances in the two experiments suggests that limited STM resources were not one of the primary factors affecting performance.

The filler problems used in the two experiments varied in the calls they made on cognitive abilities. The 2004 problem required holding information in STM and using it to solve an **`if-statement`** problem while the 2006 problem required making use of existing knowledge to solve a problem that only required a small amount of information to be held in STM.

Conclusion

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

The number of years of software development experience is likely to have a high correlation with a subjects age. While cognitive performance has been found to decrease with age [14] [15], age does not appear to have been a factor affecting the number of questions answered in this experiment (however, most subjects are likely to be younger than the age at which studies have found a significant age decrease in performance; i.e., 50s and over).

There was no significant difference in performance for the different kinds of identifiers used in this experiment. Any minor variations that might exist were swamped by differences in individual subject performance.

The most significant factors affecting assignment problem performance all seem to have their root in the mental characteristics of individual subjects. These characteristics are likely to include short term memory capacity limits, metacognitive (self-knowledge) ability, and degree of risk aversion.

Obfuscated Code Competition

Last issue we set, as a one-off, a slightly different competition – the challenge of writing the most obfuscated program to print the words of 'The 12 Days of Christmas'.

Why do people take part in an obfuscated competition? It provides an opportunity to explore the esoteric corners of a language (and its compilers), to demonstrate how clever or devious you are, to point out issues of style in an ironic fashion and lastly to give you a safe place to deposit that terrible code you've been burning to write for weeks…

Sadly, although Tim and Roger both enjoy obfuscated code competition entries, only one of the readers of CVu sent in an entry. So many thanks go to Lars Hartmann <lars@hartmannix.dk> for his entry; which is written in SML.

```
(fn
G => let fun
O 0 = [] | O n = (12-n,27-n)::O(n-1) fun
D l z [] = z | D l z (x::xs) = D l (l z x) xs fun
J 0 x = hd x | J n x = J (n-1) (tl x) fun
U [] xs = xs | U (y::ys) xs = y :: U ys xs in map
print(map(fn
L => implode(map(fn
A => chr(ord(A)+10))(explode L)))(map(fn
C => J
C G )(
U [12,0,13,14](D U [](map(fn(n,s)=>
   let fun th 13 = [] | th s=s::th(s-1)in
   12::n::13::th(s)end)(O 11))))))end)
[ "\\_hij\^V", "i[YedZ\^V", "j^_hZ\^V",
  "\\ekhj^\^V", "\\_\\j^\^V", "i_nj^\^V",
  "i[l[dj^\^V", "[_]^j^\^V", "d_dj^\^V",
  "j[dj^\^V", "[b[l[dj^\^V", "jm[b\\j^\^V",
  "Ed\^Vj^[\^V",

"ZWo\^Ve\\\^V9^h_ijcWi\^Vco\^Vjhk[\^Vbel[\^V]Wl[\^V
je\^Vc[\^@",
  "W\^VfWhjh_Z][\^V_d\^VW\^Vf[Wh\^Vjh[[$\^@\^@",
  "WdZ\^V", "jme\^Vjkhjb[\^VZel[i\^@",
"j^h[[\^V\\h[dY^\^V^[di\"\^V",
  "\\ekh\^VYWbb_d]\^VX_hZi\"\^V",
"\\_l[\^V]ebZ\^Vh_d]i1\^@",
  "i_n\^V][[i[\^VW#bWo_d]\"\^V",
  "i[l[d\^VimWdi\^VW#im_cc_d]\"\^@",
  "[_]^j\^VcW_Zi\^VW#c_ba_d]\"\^V",
  "d_d[\^VbWZ_[i\^VZWdY_d]\"\^V",
  "j[d\^VbehZi\^VW#b[Wf_d]\"\^@",
  "[b[l[d\^Vf_f[hi\^Vf_f_d]\"\^V",
  "jm[bl[\^VZhkcc[hi\^VZhkcc_d]\"\^V"
]
```

Developer Beliefs About Binary Operator Precedence (continued)

Future experiments need to investigate whether subjects giving many *would refer back* answers have less ability of remember information or are not able to reliably evaluate the accuracy of the memories they have.

Further reading

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

Acknowledgments

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

References

1. D.M.Jones, 'Developer beliefs about binary operator precedence' in *C Vu*, 18(4):14-21, Aug 2006
2. D.M.Jones, Experimental data and scripts for short sequence of assignment statements study, http://www.knosof.co.uk/cbook/accu04.html, 2004
3. H. F.Chitiri and D. M. Willows, 'Word recognition in two languages and orthographies: English and Greek'. *Memory and Cognition*, 22(3):313-325, 1994.
4. M. Taft and K. I. Foster. 'Lexical storage and retrieval of phymophemic and polysyllabic words'. *Journal of Verbal Learning and Verbal Behavior*, 15:607-620, 1976.
5. A. D. Baddeley, 'How does accoustic similiarity influence short-term memory?' *Quarterly Journal of Experimental Psychology*, 20:249-264, 1968.
6. V. Coltheart. 'Effects of phological similarity and concurrent irrelevant articulation on short-term-memory recall of repeated and novel word lists'. *Memory & Cognition*, 21(4):539-545, 1993.
7. M. Steyvers and K. J. Malmberg. 'The effect of normative contextual variability on recognition memory.' *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 29(5):760-788, 2003.
8. S. Andrews. 'The effect of orthographic similarity on lexical retrieval: Resolving neighborhood conflicts.' *Psychonomic Bulletin and Review*, 4(4):439-471, 1997.
9. J. R. Anderson, *Learning and Memory*, John Wiley & Sons Inc, second edition, 2000.
10. D. M. Jones. Experimental data and scripts for developer beliefs about binary operator precedence. http://www.knosof.co.uk/cbook/accu06.html
11. D. M. Jones. *The new C Standard: An economic and cultural commentary*. Knowledge Software Ltd, 2005.
12. ". T. Fu and W. D. Gray. 'Memory versus perceptual-motor tradeoffs in a blocks world task.' In *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, pages 154-159, Hillsdale, NJ, 2000. Erlbaum.
13. D. M. Jones. 'I_mean_something_to_somebody.' *C Vu* 15(6):17-19, Dec. 2003.
14. A. S. Gilinsky and B. B. Judd. 'Working memory and bias in reasoning across the life span.' *Psychology and Ageing*. 9(3):356-371, 1994.

Standards Report

Lois Goldthwaite brings news from the C standard committee.

As promised in the last standards column, most of this one will talk about news from the international C standard committee. Since C99 was issued, the committee has deliberately confined its work to clarifying defects in its phrasing and writing technical reports, which can suggest recommended practice or optional extensions, but which are not normative (part of the actual standard).

However, discussions at the semi-annual meeting in Portland pointed out that the standard C language has now fallen behind the current state of technology in compiler development. Virtually all major C compiler developers have developed extensions to the language which go well beyond the Standard. Meanwhile, the C++ committee is concentrating on features to support the near and medium future of programming, especially in a world where multi-processor computers and extensively multi-threading programs will be the norm. Therefore it is time to think seriously about revising the C standard, a process which takes about five years under ISO procedures. The group has not yet committed to starting the revision process, but does see the need to start planning for a revision.

The committee is compiling a list of suggested extensions, A full day at the London committee meetings (April 23-27 this year) will be devoted to sifting through these suggestions and writing a charter to guide the work. The criteria for sifting these proposals will be heavily weighted toward existing practice, features that are already in wide use. The committee is disappointed that adoption of C99 has been rather slow, and wants to avoid standardising features that will not be used.

Multi-threading and security features are probably a certainly for inclusion in some form, once a revision gets underway. The GNU C extensions, discussed at http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/index.html #toc_C-Extensions, have already been put on the agenda for discussion in April. The C committee has been closely following some of the developments in the C++ language and will not miss a chance to enlarge the common subset between the languages when new features fit 'the spirit of C'.

Additional proposals from outside the committee are welcome (if you have a proposal and do not know how to submit it, please write to standards@accu.org for further information).

Although active revision of the C standard has been off the table until now, the committee has been laying the groundwork for expanded existing practice, through official Technical Reports. The three latest of these can be found at http://www.open-std.org/jtc1/sc22/wg14/www/docs/post-portland-2006.htm. TR 24732, *Extension for the programming language C to support decimal floating-point arithmetic*, contains formal language for supporting base-10 floating point hardware directly (as opposed to the widespread binary floating point capabilities).

Also on the C committee website is a draft of TR 24731-2, proposing new C library functions which dynamically allocate memory for the values they read in or return. These may be considered more robust than the functions listed in TR24731-1, which have an extra parameter indicating the size of a buffer which has been previously allocated.

One other decision made in Portland will help developers write more robust programs – the `gets()` function will be officially deprecated in future. This function reads characters from stdin into a buffer until it sees a newline character or end of file. But it is a potential security hole, because it has no way to prevent unexpected input from overrunning the buffer. The Posix standard also will be making `gets()` 'obsolete' – an implementation must provide it, for backward compatibility, but it is emphatically not to be used in new code.

Virtually all major C compiler developers have developed extensions to the language which go well beyond the Standard.

Turning to C++ committee news, members will be extremely busy this year in the drive to put out a Committee Draft revised standard by the end of the October meeting. The convenor has called an additional full WG21 meeting July 15-20 in Toronto, in addition to the previously scheduled April and October meetings. The library subgroup will hold an ad-hoc meeting January 22-24 near Chicago, and another ad-hoc on the new concepts feature will take place February 22-23 in Mountain View, California. The April 15-20 WG21 meeting will take place in Oxford the week after the ACCU conference. If you are interested in sitting in on a session to see the committee in action, or if you want to get involved with the UK standards panel, please write standards@accu.org for more information. ■

LOIS GOLDTHWAITE

Lois has been a professional programmer for over 20 years. She is convenor of the C++ and Posix standards panels at BSI. One of her hobbies is representing the UK at international standards meetings!
Lois can be contacted at standards@accu.org.uk

Code Critique Competition 44

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Free book!

I thought I'd highlight the fact that the prize for the winning entrant of each competition is a free book, thanks to the sponsors above and Francis Glassborow who liaises with them. Entrants also get their name in print, which can impress your friends and family!

Last issue's code

This code is designed to provide a simple encryption of plain text, and `main` is a test harness than encrypts and then decrypts the text. There seems to be a problem with displaying the encrypted text for strings with spaces:

```
crypt "a test"
```

Please comment on the specific bug and the code as a whole.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void strreverse( char *src, char *dest) {
  int i=0, j;
  for( j= strlen(src)-1; j>=0;j-- )
  {
    dest[i++] = src[j];
  }
}

bool encrypt(const char *input, char *output, bool
encode) {
  char buffer[80] = {0};
  const char *src = input;
  char *dest = buffer;
  srand( strlen( src ) );
  int seed = rand();

  while (*src)
  {
    int minVal,maxVal;

    minVal = maxVal = 0;
    int charVal = (int)*src;

    if( charVal >= 33 && charVal < 48 )
    {
      minVal = 33; maxVal = 48;
    }
    else if( charVal >= 48 && charVal < 58 )
    {
      minVal = 48; maxVal = 58;
    }
    else if( charVal >= 58 && charVal < 65 )
    {
      minVal = 58;  maxVal = 65;
    }
    else if( charVal >= 65 && charVal < 91 )
    {
      minVal = 65; maxVal = 91;
    }
    else if( charVal >= 91 && charVal < 97 )
    {
      minVal = 91; maxVal = 97;
    }
    else if( charVal >= 97 && charVal < 123 )
    {
      minVal = 97; maxVal = 123;
    }
    else if( charVal >= 123 && charVal < 127 )
    {
      minVal = 123; maxVal = 127;
    }
    else if (charVal < 33 || charVal > 126 )
    {
      return false;
    }

    int range = maxVal - minVal;
    int key = maxVal % range;
    int delta = range - key;
    if ( encode )
    {
      if( charVal < maxVal - key )
        sprintf(dest, "%c", (charVal + key));
      else
        sprintf(dest, "%c",(charVal - delta));
    }
    else
    {
      if( charVal >= minVal + key )
        sprintf(dest, "%c", (charVal - key));
      else
        sprintf(dest, "%c",(charVal + delta));
    }
    *dest++;
    *src++;
  }
  *dest = '\0';

  char revBuffer[80]={0};
  strreverse(buffer,revBuffer);
  strcpy(output,revBuffer);
  return true;
}

int main( int argc, char **argv )
{
   for ( int i = 1; i < argc; i++ )
   {
      char *input = argv[i];
```

ROGER ORR

Roger has been programming for 20 years, most recently in C++ and Java for various investment banks in Canary Wharf. He joined ACCU in 1999 and the BSI C++ panel in 2002.
He may be contacted at rogero@howzatt.demon.co.uk

```
      char output[80];
      encrypt( input, output, true );
      printf( "%s\n", output );
      encrypt( output, input, false );
      printf( "%s\n", input );
   }
}
```

Critiques

From John Appleby-Alis <John.Appleby-Alis@celoxica.com>

The problem displaying encrypted text for strings containing spaces is due to the encrypt function only handling strings that are composed of characters with ASCII code values within the range 33 to 127. Google for 'ASCII lookup table' and you will find that the space character has an ASCII code value of 32.

The encrypt function returns `false` on failure, but your test harness fails to inspect this return value, and so prints the contents of an uninitialised array instead of reporting the encrypt function failure.

Here are the other more general issues I spotted in your code.

You are using fixed length arrays to store strings with lengths specified by the input to your program. You have no guarantee that the input strings will fit inside the fixed length arrays. This makes your program vulnerable to buffer overrun bugs. If I pass a string longer than 79 characters to your application it might crash (in fact it does on my system). There are (as usual) many different ways to address this problem. One is to allocate buffers of variable length yourself on the heap using `new` and `delete` (or `malloc`/`free` if you must), better would be to use the stl `vector` class which has an array compatible interface and will manage heap memory for you automatically.

```
std::vector<char> buffer (strlen (input)+1);
```

My preferred approach though would be to redesign your code a bit. Your encrypt function operates on characters one at a time and with no dependency between them. I see no reason why you shouldn't modify the input to encrypt in situ, removing the need for buffers to hold intermediate values altogether.

There is a lot of repetition in your code that you could roll up into a loop. The long chain of if else statements could be expressed like this:

```
const int values [] =
  {33, 48, 58, 65, 91, 97, 123, 127};
const size_t values_size =
  sizeof (values) / sizeof (values [0]);

for (size_t i = 0; i < (values_size-1); ++i)
{
  if (charVal >= values [i] &&
      charVal < values [i+1])
  {
    minVal = values [i];
    maxVal = values [i+1];
  }
}
```

The final `else if` which tests for `charVal` outside the acceptable range of character codes doesn't fit inside this loop though. You can still reuse the contents of the values array to test for this, rather than repeating constant integer literals in your code.

```
if (charVal < values [0] ||
    charVal >= values [values_size-1])
  return false;
```

You use `sprintf` to append characters to a buffer. As I mentioned earlier, you don't need to use an intermediate buffer at all, but the use of `sprintf` here is serious overkill anyway because you can just assign the value directly. Instead of

```
  sprintf (dest, "%c", (charVal + key));
```

say

```
  *dest = charVal + key;
```

You forgot to make the `src` parameter const in the `strreverse` function, but the fact that you used `const` on the input parameter to `encrypt` suggests you already know this is a good idea.

The `seed` variable in `encrypt` is never used. When I turned on all warnings from my compiler (gcc 4.0.3) it spotted this straight away.

Your use of `printf` in main is overkill. Both `printf` and `scanf` have to parse and interpret the format string you pass to them. Since you're not actually producing formatted output, use the less expensive `puts` function instead.

From Calum Grant <calum.grant@sophos.com>

The algorithm implements a "simple monoalphabetic substitution cipher" to encrypt a string of text. The `encrypt()` function maps each character to a new character, then calls `strreverse()` to reverse the string. Be aware that this cipher is very easy to crack.

The first problem is that the `encrypt()` function is quite long and should be split up. It is actually a `std::transform` followed by a `std::reverse`. A look-up table could be used to map characters, which is simpler, more flexible and very efficient:

```
const char encrypt_table[TABLE_SIZE] = {...};
const char decrypt_table[TABLE_SIZE] = {...};
```

We also need to handle characters that exceed our `TABLE_SIZE`, since unfortunately we can't assume chars are 8-bit. In your code you reject certain characters and fail silently. The character 32 (ASCII space) is not handled, which results in your bug. You don't need a failure case, though we probably want to leave out of range characters unchanged.

Now we come to a serious issue: you don't have a key. Your 'key' variable is completely misnamed. `encrypt()` should use a key, which could just be a look-up table. Instead of passing a `bool` into `encrypt()`, you can pass in a pointer to the appropriate look-up table.

In your code, the `seed` variable is not actually used and the `while` loop could be replaced by a `for` loop, or, even better, a `std::transform` in C++. You should avoid hard-coded values in your code, and to be truly portable you can't even assume ASCII. Don't use `sprintf()` to write to a pointer, instead you can just write `*dest = ...`. You allocate fixed-length arrays for intermediate results, which can quite easily lead to a buffer overrun. Never make up a size and hope that an array will be large enough.

You can just encrypt the memory 'in-place', with no memory allocation, no unnecessary copying and a smaller 'working set'. We need to keep an eye on performance for encryption. You can write a version of `strreverse()` that operates in place on a single array (this is an exercise), or just use `std::reverse()` in C++.

You could pass the length of the string to `encrypt()`, which allows it to work on binary data.

```
void encrypt(char * data, std::size_t size,
  const char * table)
```

To ensure that key is the expected size, we could introduce an auxilliary class for the key (which could then serve as a functor), or just do the following:

```
 void encrypt(char * data, std::size_t size,
   const char(&table)[TABLE_SIZE])
```

From David Carter-Hitchin <David.CARTER-HITCHIN@rbos.com>

Initial Exploration of the code

1. First thing to note is that the code looks like C and has C-style headers. However, upon compilation with GCC's gcc command, it fails on line 13, which contains the `bool` datatype. I suppose some C compilers may implement this datatype for you, but strictly speaking `bool` is only part of C++, so rolling the g++ compiler over the code results in an executable. So far so good! This quirk (inclusion of `bool` in C-code) should have been commented by the student as a non-standard feature – or alternatively note that this was a C++ project written mostly in C-style (for historical reasons perhaps).
2. When running the binary with several test cases reveals this code implements a primitive Caesar cipher, where plaintext letters are shifted by a fixed number positions along the alphabet, so here 'a' is shifted 19 characters to 't', 'g' to 'z' and 'h' wraps around the end of the alphabet to become 'a'. Similarly 'A' becomes 'N', 'M' becomes 'Z' and 'N' becomes 'A'. Numbers seem to work in a similar manner too. Entering some non-alphanumeric characters works sometimes, e.g. '&' becomes ')' (ASCII 38 and 41) and '%' becomes '(' (ASCII 37 and 40). So we expect ' ' (ASCII 32) to become '#' (ASCII 35), but no such luck. On my terminal in comes out as 'y' with an umlaut followed by a '3/4' character, followed by a German double s.

Code Analysis

Time to look at the code. After a minute or so the error is pretty obvious – one of the ranges of characters starts at ASCII 33 (!) and not at 32 (space).

Unfortunately the fix to this isn't as simple as altering the range from 32 instead of 33, as if we do that then the statement `int key = maxVal % range;` sets the key to `0` as `32` is an exact multiple of the range 32 to 48 (16), which is bad news as this means the following characters would not be 'shifted' (`SP` = space):

```
32  SP      33  !       34  "       35  #

36  $       37  %       38  &       39  '
```

This is perhaps why the student amended the code to skip over the problematic value of 32. To solve this we need an additional step to check if the key is zero and, if it is, set it to something sensible (such as the middle of the range).

So lines 31 onwards become:

```
if( charVal >= 32 && charVal < 48 )
{
  minVal = 32; maxVal = 48;
```

line 58:

```
else if (charVal < 32 || charVal > 126 )
```

new lines after 64:

```
if (key == 0) {
  key = range/2;
}
```

With these changes, the code at least does what it's supposed to:

```
[io]-> ./crypt "a falling apple"
xeiit(zgbeety(t
a falling apple
```

General Code Critique

As it stands the code is pretty 'raw'. It doesn't carry a single comment, which is very bad news for anyone trying to understand it, including the person who wrote it when they come to look at it after a few years! That said, the choice of variable names seems to be ok and reasonably understandable. The code is reasonably formatted and includes are in sensible places and so on. Apart from the error with spaces, the code essentially works, but there are a number of defects.

Design-wise the code is pretty strange. For example, it is odd that the ASCII set is split up into separate ranges and the 'rotations' are done within those ranges. This means that a lot more code is required to do the same job, than if we treated the entire printable ASCII set as one range, i.e. from ' ' (32) to ~ (126). Caesar ciphers are very weak anyway, so why make it even weaker by always using numbers for numbers, letters for letters and symbols for symbols? Perhaps this was part of the specification; perhaps it was just some evil teacher making the poor life of a student more difficult. Rewriting this so that the entire set is treated as one block, we get a much more compact encrypt function:

```
while (*src)
{
  int charVal = (int)*src;

  if (charVal < 32 || charVal > 126 ) {
    return false;
  }

  int minVal = 32;
  int maxVal = 127;
  int range  = 95;
  int key    = 32;
  int delta  = 63;

  if ( encode )
  ... [ as before ]
```

Also the output is far less amenable to 'casual decryption':

```
[lon0020xuc]-> ./crypt2 "a falling apple"
%,00!?'.),,!&?!
a falling apple
```

It is very common in cryptography to use a substitution letter for either something unprintable or spaces. So instead of bailing out of the encryption, it would be better to substitute in a letter such as X, which would make it obvious to the reader of the decoded message that this was outside the scope of the encryptor. So we have:

```
if (charVal < 32 || charVal > 126 )
  charVal = 88;
```

Resulting in ASCII 8, Backspace, (for example) becoming X in plain and w in crypt:

```
[lon0020xuc]-> ./crypt2 "a falling apple^H"
w%,00!?'.),,!&?!
a falling appleX
```

There are numerous other problems with the code:

- the buffers used are fixed at 80 chars. If the input exceeds this, we get unpredictable behaviour. In one run I found that a portion of the test was displayed in plaintext. Another run core-dumped. To fix this the student could `malloc` the required amount of memory, or simply use a `std::string`, which does all that size management for you.
- It is inefficient to cast the current character to `int`, just to inspect its ASCII value; it would be easier to just use the value of `src[0]` directly, since chars are really bytes in disguise.
- It is inefficent to build up a result char array and reverse it - much quicker to write the results into the final result array in reverse order.
- The code needs to decide if it's C or C++. If it's C++, the headers and output commands need updating and obviously it should make

use of STL. If it's pure C then it needs to get rid of **bool**, by using an **int** or a **char** instead.

- **main** doesn't actually check the return status of **encrypt** or **decrypt**.
- Could be argued that two functions for encrypting/decrypting istead of one make for a clearer design.
- The random seed is never used, and should be removed if it's not needed.

From Balog Pál <pasa@lib.hu>

I'll not go down to philosophical issues of what 'simple encryption' means; let's just assume the letter-substitution combined with rev is okay for the task. First I scan only the function signatures, to have a picture, with a 'reader' hat on. Here's how I would decrypt the signatures if they appeared in my program:

```
  void strreverse( char *src, char *dest);
```

src will be a 0-terminated C-style string (called just string in the rest), that will be modified in some cases. **dest** will receive the output producing src in reversed order, zero-terminated. Probably if **dest** is null, output will go to **src** doing in-place reverse, otherwise **src** and **dest** shall not overlap. If **src** is **NULL**, behaviour is unguessable.

```
  bool encrypt(const char *input, char *output,
               bool encode);
```

This function will convert plaintext and cyphertext. The final **bool** parameter will order the direction **true** being encode, **false** decode. We'll have a string in input that will be converted to output. The source is never modified, the destination shall not overlap. Result of conversion is returned: **true** means success, **false** means failure, for that case state of output is unguessable – assumed indeterminate. The function would be better named **crypt**, as encrypt suggests one direction contradicting the direction param.

For both functions the caller is responsible to pass a destination buffer that has enough space to hold the output.

Then I think of the test cases for verification. We certainly have to encrypt and decrypt and compare the final result with the original plaintext. And do that with strings of various length (including length 0, 1 and 'very long') and composition (including generation of all possible 1-character long strings as a trivial test).

Now let's see the actual test harness in **main**. Even at the first glance it looks lousy. Fixed 80-char length buffer, no check of function result, no compare.

And we immediately see two easy ways to crash the program: we can supply input long enough to overrun the 80-char output buffer, or provoke encrypt to fail, and then the code will still use the indeterminate buffer content for further processing. Later we'll see that is exactly what happens with "a test" input: the space character is outside the legal range, encrypt fails, but indeterminate garbage is still passed to a following **sprintf** as a zero-terminated string.

Back to the implementation. **strreverse** turns out to work quite differently than we deduced from the signature. It actually works from **src** to **dest**, so the correct signature would have **const char * src**. Then I see no reason to have **i** and **j** outside the **for**. And the most important problem: the output string will not be zero-terminated. Keeping as much as possible of the original, I'd write the implementation this way:

```
// src and dest must not overlap!
void strreverse(const char *src, char *dest)
{
  for(int j=strlen(src)-1 ; j>=0 ; --j )
    *dest++ = src[j];
  *dest = 0;
}
```

Or just use an 'in-place' reverser with a single argument, unless we really need to keep an original intact. That would remove the need of an output buffer, and attached constraints of being large enough and not overlapping. (**swap** can be inlined if we want to keep it C99).

```
#include <algorithm>
void strreverse( char * str )
{
  for(int i = 0, j = strlen(str)-1; i < j ;
        ++i, --j)
    std::swap(str[i], str[j]);
}
```

The crypter function starts pretty ugly. Again a fixed-len buffer, then **srand** and **seed** that is not used for anything at all, within the loop introducing variables without initialization, then immediately use assignment, and a chain of similar-looking code just crying for some abstraction.

As a start I'd want to move most of the code in the loop to a separate function **mapChar()** as what happens is actually the characters are substituted. For the simple case it would be

```
  char mapChar(char in, bool encrypt);
```

But we notice from the code that not the whole character range is handled. So, unless we come up with sensible support of the whole range (I see no reason for restriction), we must pass back the error status. It can be a special value of **char** that can't be result of mapping – **0** looks good – or we return **bool**, and pass in a pointer/ref for the resulting **char**. Taking the first idea what remains of the **encrypt** function:

```
  bool crypt(const char *input, char *output,
             bool encode)
  {
    char * dest = output;
    for(;*input; ++input, ++dest)
    {
      *dest = mapChar(*input, encode);
      if(0 == *dest)
        return false;
    }
    *dest = 0;
    strreverse(output);
    return true;
  }
```

Those who like code with least lines could even write the loop as:

```
  for(;*input; ++input)
    if(0 ==(*dest++ = mapChar(*input,encode)))
      return false;
```

but others would find that less easily readable. [Ed: I removed some spaces to get the single liner to fit in the column format] Note what we gained up to here:

1. code is small and readable
2. no more local buffers introducing extra constraint on length
3. though it's an accidental side-effect, on error return the output string is 0-terminated. We might even specify it as a postcondition

The only thing left is implementing **mapChar**. I would probably just use a static table of 256 characters, and return the proper entry:

```
  const char encr_tab[256] = { 0, ...  };
  const char decr_tab[256] = { 0, .... };

  char mapChar(unsigned char in, bool encrypt)
  {
    return in[encypt ? encr_tab : decr_tab];
  }
```

(note : the actual initialiser for this mapping would be something like "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0$%&'()*+,-./!"#8901234567<=>?@:;NOPQRSTUVWXYZABCDEFGHIJKLM\]^_`[tuvwxyzabcdefghijklmnopqrs~{|}")

The tables can be generated by a simple program. The decryption table can be generated from the encryption table (`decr_tab[encr_tab[i]] = i`). The encryption table can be generated at startup as well, if we include a proper function and we can even pass it a key – so it finally deserves the title 'encryption' as simple as it is. Anyway, here's a mapper for the original behaviour:

```
char mapChar(unsigned char in, bool encode)
{
  static const int ranges[] = {0, 33, 48, 58,
     65,  91, 97, 123, 127, 256};
     // keep sorted, with 0 and 256 at sides!
  int min, max;
  for(int i = 0; ; ++i)
    if(in >= (min = ranges[i]) &&
       in < (max = ranges[i+1]))
      break;

  if(min == 0 || max == 256)
     // those ranges considered illegal...
    return 0;

  const int range = max - min;
  const int key = max % range;
    // arbitrary consistent pick...
  const int delta = encode ? key :
    range - key; // up or down?
  const unsigned char res = in + delta;
  return res < max ? res : res - range;
    // adjust turnaround
}
```

How is it better?

- all the redundancy with the range check is removed. Just try to add, remove or change a range, and see how many places need update in the original and here
- we streamlined the cases, so we have just one point where encryption and decryption differs to see it's a proper inverse. And turnaround code is no longer duplicated

Other problems with the original code that just disappeared:

- `sprintf` to `%c` could be simply `*dest =`
- `*dest++` has an excess `*`

If someone wonders why the code worked with `strreverse` not terminating the string: the `={0}` initialiser of buffer actually causes it zero-filled, so as long as we write less than 80 characters, there will be a 0 at the end. I was also shocked by the practice of overwriting strings pointed by `argv[i]`, but the standard says that is okay.

From Michal Rotkiewicz <michal_hr@yahoo.pl>

This encryption/decryption method resembles the Caesar method, where each letter is shifted by 3, i.e. 'a' becomes 'd', 'b' becomes 'e', 'c' becomes 'f' and so on. When we are approaching to the end of alphabet we 'wrap' encrypted letters: 'x' becomes 'a', 'y' becomes 'b' and 'z' becomes 'c' and so on.

In the presented code the encryption/decryption algorithm is a little bit more complicated:

1. there are seven ranges and each range has different shift value (variables key or delta depending if we have to do wrapping or not: wrapping is done when `charVal+key>=maxVal`)
2. encrypted text is reversed

Each letter is converted from `char` to ASCII value and then we check in which range it is. ASCII value of the 'space' is 32 and it is not within any range therefore encrypt function ends with return value `false`.

At first sight it may be obvious to extend first range by changing 33 to 32:

```
if (charVal >= 32 && charVal < 48)
{
    minVal = 32; maxVal = 48;
}
```

But it's wrong. Let's calculate `range`, `key` and `delta`:

```
  int range = maxVal - minVal // 48 - 32 = 16
  int key = maxVal % range    // 48 % 16 = 0
  int delta = range - key;    // 16 - 0 = 16
```

It means that every character in the range 'space' - '/' (ASCII values 32 - 47) won't be encrypted as all of them fulfill `if (charVal < maxVal - key)` and because `key=0` then `charVal+key` (in the `sprintf`) is in the fact `charVal+0`. There are several ways of solving this problem. I will present here two of them.

1. The simplest way is to check whether `key` equals 0 and if so change it to any value between 1 and `range`. It may be hardcoded or calculated like this:
   ```
   int key = maxVal % range;
   if ( 0 == key ) key = range - 1;
   ```
2. In this approach we may extend the upper limit of the first range to 49. Then first two ranges are determined as follow:
   ```
   if (charVal >=32 && charVal < 49)
   {
      minVal = 32; maxVal = 49;
   }
   else if (charVal >= 49 && charVal < 58)
   {
      minVal = 49; maxVal = 58;
   }
   ```

Then for the first range we have:

```
  int range = maxVal - minVal // 49 - 32 = 17
  int key = maxVal % range    // 49 % 17 = 15
  int delta = range - key;    // 17 - 15 = 2
```

Having above 'space' is encrypted to '/'.

But by extending the range we included character '0' which previously belonged to the second range. We don't know how the encryption requirements looks like (if any). Maybe there is some reason for having all characters 0-9 in the separate range. If there is such requirement we have to change we calculate the key: it may be for instance explicitly provided for every range.

That's all about algorithm. The rest of my comment is regarding code itself:

1. Program may crash if input string is longer than 80 characters. There is no any check whether this condition is violated. To cope with it we may either check input string length or allocate memory dynamically. I prefer the second approach, as it seems to be more flexible.

We have to remember that if we are deleting dynamically allocated memory pointer in the instruction `delete`, `*ptr` must point to the beginning of the memory. Therefore I introduced `const char * const ptr` that is set to the beginning of the array. As it's a `const` pointer we are sure that it not moved.

2. Argument `char  *src` of the `strreverse` function may be declared as `const char *src`.

   It's good practice to declare as `const` all arguments that are not changed within function.
3. Lines:
   ```
   srand(strlen( src ) );
   int seed = rand();
   ```
   are unnecessary as seed variable is not used.
4. Instead of reversing buffer to the `revBuffer` and `strcpy` it to output we may use output as a second argument of the `strreverse` function.
5. In `main` function return value of the `encrypt` function should be checked.

Taking into account points 1-5 and extending first range code looks like:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void strreverse(const char *src, char *dest) {
  int i=0,j;
  for(j=strlen(src)-1; j>=0;j--) {
   dest[i++]=src[j];
  }
}

bool encrypt(const char *input, char *output,
             bool encode) {
  const char *src=input;
  int inputLength = strlen(src);
  char *dest=new char[inputLength+1];
  const char * const ptr = dest;
  while(*src)
  {
    int minVal, maxVal;
    minVal=maxVal=0;
    int charVal=(int)*src;
    if (charVal>=32 && charVal<49)
    { minVal=32;  maxVal=49; }
    else if (charVal>=49 && charVal<58)
    { minVal=49;  maxVal=58; }
    else if (charVal>=45 && charVal<65)
    { minVal=58;  maxVal=65; }
    else if (charVal>=65 && charVal<91)
    { minVal=65;  maxVal=91; }
    else if (charVal>=91 && charVal<97)
    { minVal=91;  maxVal=97; }
    else if (charVal>=97 && charVal<123)
    { minVal=97;  maxVal=123;}
    else if (charVal>=123 && charVal<127)
    { minVal=123; maxVal=127;}
    else if (charVal<33 || charVal>126)
    {
       delete [] ptr;
       return false;
    }

    int range=maxVal-minVal;
    int key=maxVal%range;
    int delta=range-key;
    if (encode)
    {
       if (charVal<maxVal-key)
         sprintf(dest,"%c",(charVal+key));
       else
         sprintf(dest,"%c",(charVal-delta));
    }
    else
    {
       if (charVal>=minVal+key)
         sprintf(dest,"%c",(charVal-key));
       else
         sprintf(dest,"%c",(charVal+delta));
    }
    *dest++;
    *src++;
   }
    *dest='\0';
    strreverse(ptr,output);
    delete [] ptr;
    return true;
 }

int main(int argc, char **argv)
{
  for(int i=1;i<argc;i++)
  {
    char *input=argv[i];
    char *output = new char[strlen(input)+1];
    bool ret = encrypt(input,output,true);
    if (ret == false)
    {
      printf("Invalid character\n");
      delete [] output;
      return 1;
    }
    printf("%s\n",output);
    ret = encrypt(output,input,false);
    // Below check may be omitted as assuming
    // that algorithm is correct we are sure
    // that all characters are in valid
    // ranges.
    if (ret == false)
    {
      printf("Invalid character\n");
      delete [] output;
      return 1;
    }
    printf("%s\n",input);
    delete [] output;
  }
}
```

Commentary

This code slightly horrified me when I first saw it because it was wrong on so many levels!

Firstly, it was really just C code although it was being compiled as C++ code (all it needed was `#include <stdbool.h>`). One problem with C++ is that many people still write C in it – if C compatability is not required I prefer to write in idiomatic C++ because this allows me to use more of the power of the language and the library; in this case for example my preference would be to make the function take `std::string` arguments rather than character pointers to avoid the problems with fixed length buffers without burdening the code (or the caller) with explicit memory allocation.

Secondly the algorithm used had an unstated restriction on the range of input characters supported – hence the presenting problem of failing to deal properly with the embedded space. Of course, when used to encrypt passwords. It also uses a very trivial cipher that provides almost no real security; Caesar ciphers are easy to crack and one that breaks the input into several disjoint ranges is even easier.

Thirdly the code had lots of duplication that could be reduced either by using a lookup table or by changing the algorithm to use a single range. Duplication is usually bad because it affects readability, maintainability and performance.

Fourthly the code was not very efficient – the unnecessary use of **`sprintf()`**, the double copying of characters at the end of encrypt and using **`printf()`** in **`main()`** rather than **`puts()`**. Premature optimisation is bad, but these seem to me to be some good examples of so-called ‘premature pessimisation’.

Finally the test code in **`main()`** relied on manual inspection of the output by the programmer rather than checking the return codes and output values automatically.

The Winner of CC 43

I found it hard to pick a winner for this competition since all the entries gave a good critique of the code and each one covered most of the problems in the example. Eventually I picked the winner of CC 43 as Balog Pál, partly because I particularly liked the brief discussion about what should be in a minimal set of test cases.

Code Critique 44

(Submissions to scc@accu.org by 1st March 2007)

My thanks to the donator of this piece, who would like to remain anonymous – but you know who you are.

This is a scaled down version of some production code which was longer and slightly more tortuous.

Feed the program (assume here built as `run.exe` or **`run`**) with arguments like:

```
run foo bar baz
```

And it prints:

```
f,o,o
b,a,r
b,a,z
```

Please critique this code, suggesting how the writer could improve their coding technique.

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

using std::cin; using std::cout;
using std::ostream_iterator;
using std::stringstream;
using std::string; using std::vector;

int main(int argc, char *argv[])
{
  vector< string > names(
      argv + 1, argv + argc );

  size_t written = 0;
  while( true )
  {
    stringstream output;
    bool added = false;
    if( written == names.size() )
      break;

    string name = names[ written ];
    for( int index = 0; index != name.size();
         ++index )
    {
      output << name[ index ];
      if( index < name.size() - 1 )
        output << ",";

      if( written > names.size() )
        continue;

      added = true;
    }
    if( !added )
      break;

    cout << output.str() << "\n";
    ++written;
  }

  cin.get();
  return 0;
}
```

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue’s deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Prizes provided by Blackwells Bookshops and Addison-Wesley

We need you!

C Vu and Overload rely on article contributions from members. That’s you! Without articles there are no magazines. We need articles at all levels of software development experience; you don’t have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

If seeing your name in print wasn’t enough, every year we award prizes for the best published article in C Vu, in Overload, and by a newcomer.

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous “not recommended” rating, you are entitled to another book completely free.

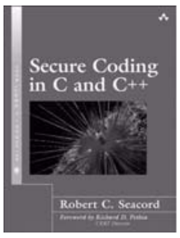
I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

C and C++

Secure coding in C and C++

by Robert S. Seacord, published by Addison-Wesley, ISBN 0-321-33572-4.

Reviewer: Ian Bruntlett



This book draws upon (nearly 18000) software vulnerability reports made to CERT/CC and the relatively small number of software defects that cause these problems.

Before I started reading this book I thought I was pretty much on top of the problems:

- When dealing with arrays, always pass the size of the array (as a `size_t`) as well as a pointer to the array.
- Avoid unsafe C library routines (use `get_s` not `gets`, `strcpy_s` and `strcat_s` instead of `strcpy` and `strcat`). If `strcpy_s` or `strcat_s` are unavailable then either (1) use your own drop in replacements – in the past I wrote my own, `strcpyterm` and `strappend` or (2) use someone else’s such as BSD’s `strlcpy` and `strlcat`.

I was surprised that the use of `typedef`s wasn’t dealt with more closely. In particular, if you are working on a large application that will get ported to different architectures, some application specific type choices are best dealt with by using a `typedef`. To give a more concrete example when 16 bit integers were no longer big enough for the Libris OPAC search engine, I had to (1) go through all the code used replacing certain `int` declarations with `TotBook_t` and (2) use a `#define PF_TOTBOOK` for use in `sprintf` calls. By having a `typedef` and `#define` it made the software more supple with regard to future developments.

The main topics are “strings”, “pointer subterfuge”, “dynamic memory management”, “integer security”, “formatted output” and “file I/O”. I was particularly pleased to see production quality code in a book – on pages 293 and 294 there were some string sanitization functions. This book was both interesting to read and enlightening.

At the back of the book is a “Recommended Practices” book this is useful but I think a “quick reference” section would have been helpful here.

VERDICT: Recommended.

Data structures and alogrithm analysis in C++. Third edition.

by Mark Allen Weiss, published by Addison Wesley, 586 pages. IBSN 0321397339

Reviewer: Frances Buontempo.

This book provides a thorough introduction and in depth look at how to define the complexity of an algorithm and then goes on to look a various structures including lists, stacks, queues and trees. The algorithms encompass sorting, disjoint set classes (i.e. equivalence relations) and graphs. Various techniques such as greedy versus divide and conquer, randomised and backtracking algorithms plus dynamic programming are detailed. A clear, in depth explanation of amortized analysis is presented and finally other specialised data structures such as splay trees, red-black trees, skip lists, other trees, traps, and pairing heaps are covered.

Prior to this, the book starts with a hit and run overview of C++ (including classes, `std::vector` and `std::string`, pointers, parameter passing, templates and functors) and mathematics (exponents, logarithms, series, modular arithmetic and proofs). Prior to starting a computer science course, a friend with advanced-level school mathematics background worked through the revision mathematics session with me. Suffice to say it served as a good basis of things to learn but was too sparse and terse to be of any use other than as a pointer to which topics to pursue in another book. The same is true of the C++ introduction, though the author lists all the right books for a C++ rookie to buy and read.

Introductory chapter aside, this book is very well written, and given the in-depth technical content, a pleasure to read. The algorithmic analysis gives detailed mathematical proofs, for example of the complexity of operations on a variety of data structures. Without a strong mathematical background or some determination, this could prove heavy going but worth it. The illustrations frequently manage to convey some complicated ideas with clarity. There is enough sample code to build examples to pursue the ideas further, but this does not distract from the main narrative as can happen with some books.

The majority of the material covered in this book is probably amply covered elsewhere (Knuth springs to mind). Nonetheless it pulls together a neat selection of ideas and presents them in a format that can be read in a few (determined) sittings. The introductory chapter will be little use to beginners and of limited value for those in the know, but apart from that this is a good book.

Recommended with the reservation: buy if you can afford it and don’t already have Knuth or similar, or have Knuth and can't manage to read it from cover to cover but wanted to.

Bookshops

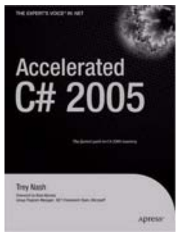
The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU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

- **Computer Manuals** (0121 706 6000) www.computer-manuals.co.uk
- **Holborn Books Ltd** (020 7831 0022) www.holbornbooks.co.uk
- **Blackwell’s Bookshop**, Oxford (01865 792792) blackwells.extra@blackwell.co.uk

C#

Accelerated C# 2005

by Trey Nash, published by APress,US (31 Aug 2006), 401 pages (paperback). ISBN: 1590597176



Reviewer: Paul Grenyer

I was expecting this book to be a lot like Andy Koenig and Barbara Moo's Accelerated C++, but for C#. It bears almost no resemblance at all which, although disappointing, doesn't make this a bad book. Well, not all of it anyway.

I found the first three chapters dull and boring. This was probably because I've read a number of introductory books on C# over the last few months and usually look forward to getting straight into some code (just like in Accelerated C++), rather than going over the details of .Net. These details are important, but do they really require 3 chapters over 33 pages?

Chapter 4, Classes, Structs and Objects, is where it all starts happening. From this point on the book is informative, detailed in a good way and interesting. The chapter on interfaces and contracts is particularly good, with some emphasis on how interfaces define contracts.

This book, like so many others, mentions the singleton pattern. I'll begrudgingly permit the suggestion that there's nothing intrinsically evil in 'mentioning' singleton. However, yet again there is absolutely no mention of the downside of using the singleton pattern. When you consider that in the view of many of the best developers these downsides are considered so bad that singletons are avoided at all costs, this is a fairly large flaw in the book.

The final chapter is entitled 'In Search of C# Canonical Forms'. This is 50+ pages of effective C# type items. The item that sticks out as being wrong is the suggestion that using the NVI (Non-Virtual Interface) interface is a good thing. Some people, including Herb Sutter, agree that NVI is a good thing. Many of the rest of us think it adds unnecessary code overhead and the need for strangely named protected methods. Again, only one side of the argument is given.

I think that overall this is a good book. The author knows his stuff and is good at presenting it for the most part. Personally I would like to see the first three chapters reworked and shortened, the singleton pattern either dropped or both sides of the argument given and the final chapter replaced with a reference to Wagner's *Effective C++*.

C# Precisely

by Peter Sestoft and Henrik I. Hansen, published by the MIT Press, ISBN 0-262-69317-8, 204 pages

Reviewer: Silvia de Beer

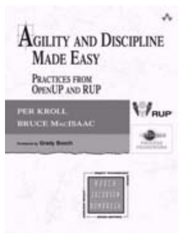
The back of the book claims 'This book is intended for readers who know Java and want to learn C# and as a quick reference for anyone who wants to know C# in more detail than that provided by a standard textbook.' The book presents the entire C# 2.0 programming language, including generics, iterators and anonymous methods. It excludes most of the extensive Microsoft.NET framework class libraries except threads, input/output, and generic collection classes.

The layout of this book is special in the way that all even pages are used to explain the language concepts in theory and all odd pages are used for examples. The idea behind this is nice, but does not always work out that well. Sometimes the even page is difficult to understand, and can only be completely understood after reading the examples. Sometimes the examples do not cover all the theory, and do you keep wondering about other cases. The examples consist of little bits of code, and the output is not always given in the explaining text, so for a thorough understanding you should really execute the code yourself to verify your understanding. I would rather use this book as a quick reference, because to use it as a tutorial can be a bit hard, not everything is immediately understandable if you are new to C#. I had difficulties understanding some theoretic pages, because some forward references to concepts specific to C# are used. I had difficulties in understanding some of the theoretic explanations like delegate and yield. To learn C# one would need much more exercise to use it properly than just reading this book. For example, 6 pages on the subject of threading do seem too little to me on such a complex and error prone subject.

Software Development

Agility and Discipline Made Easy

by Per Kroll & Bruce MacIsaac, published by Addison-Wesley, ISBN: 0-321-32130-8



Reviewer: Paul Thomas

Recommended.

This one differs from the huge array of introductions to the RUP in that it seems to be mostly apologising for not being XP. The high priests of the Unified Process were caught off guard by the sudden surge in popularity of the 'Agile' methodologies. It was a surge they helped to cause by pushing woefully bad case tools and venerating the process too highly (which is, after all, their main source of revenue) but there is always a danger of the pendulum swinging too far the other way. So they did what every threatened business does these days, they heavily re-branded and blamed customers for misinterpreting their message. There are passages in the book that hint at this: you are advised not to go looking through pre 2005 versions of the RUP for example.

This is the first I've seen of the newly re-branded RUP and, apart from the defensive tone, it seems pretty good. Although there are hundreds of process books available, they are almost always enormous tomes of lore or highly personal opinion pieces. This has the benefit of decades of process research behind it but is light enough to start using it after a weekend reading in the right places.

The core of the book is 20 practices (like the recipes in the popular 'cookbook' format) that managers can cherry pick to get them started. Not only that, but each practice can be applied in varying levels of severity. There is even a graphical key to show you how agile or disciplined you'll become by following the advice. As you'd expect, it's a little buzzword heavy.

I'm making light of the spin associated with the process industry but the truth is that I like this book. It has just about everything you need to get going in digest form and is thick with references if you want to progress. Sure there's a heavy bias toward the IBM/Rational world view, but there's plenty of coverage of other methodologies like XP, Scrum, Crystal. All in all, it looks like a good way in if you really want to improve things and it won't lock you in to a tool chain too early.

Dynamics of Software Development (2006 edition)

by Jim McCarthy and Michele McCarthy, published by Microsoft Press, 199 pages, ISBN: 978-0-7356-2319-4

Reviewer: Pete Goodliffe

Rating: Recommended

This is a reissue of a classic software development tome originally released in 1995. It was a worthwhile read then. Time has been kind to it, and a lot of the points discussed are still perfectly relevant 11 years on. In fact, the McCarthy's ideals can clearly be seen as precursors to many fashionable 'agile' disciplines.

The original book presented a series of ideas and rules of thumb to help shape better dynamics in software development teams and processes. These items ranged over the lifetime of the development process: from the 'opening moves', through shipping, and the launch, and beyond. It also provided an appendix on hiring and keeping good people.

This 2006 edition reproduces the original book and adds a little extra material (28 new pages, added to the original 150-odd). It introduces some very superficial updates to the original material: mostly the addition of cross-references to the new material.

The 'old' 1995 book content has been largely left alone and called Part 1. The book therefore has an slightly odd structure: the original book's appendix is reproduced at the end of part one, BEFORE part two's new 2006 material. There is a *kind* of logic here.

The authors originally intended to weave all of the items into a richer tapestry in subsequent editions. The rules don't interconnect as much as they could. The 2006 update doesn't touch the old material, and so doesn't attempt to complete this weaving. This edition adds three new rules of thumb, and then provides a fantastically brief summary of the McCarthy's 'The Core System 3.0' – a process/system for interacting teams as described in their 2002 book *Software For Your Head*.

To be honest I felt this 'Core Protocol' stuff was a total waste of time. The material is presented so briefly – a mere summary of another book – that without any proper discussion it serves to confuse and baffle the reader more than add any value. It's 20 pages of quick overview and bulleted lists. As it's about 3/4 of the entire 'new' material, it doesn't makes a compelling case for buying the new edition.

The book includes CD ROM with a video Jim McCarthy giving his famous '23 1/2 Rules of Thumb' presentation, upon which the book is based.

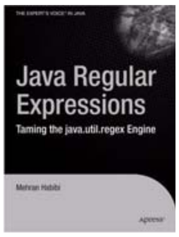
This *is* a good book, but I was left with the feeling that a more substantial update could have been very interesting. This edition seems like a release for marketing purposes alone. If you don't already own the original book, then it's an interesting read; for historical purposes as much as anything else. If you do have it, read that copy again, and spend your money elsewhere.

Java

Review of Java Regular Expressions

by Mehran Habibi, published by Apress 2006, ISBN 1590591070

Reviewer: Christer Lofving



This is maybe the first book from Apress that I DO not really like.

Even a brief look around the pages reveals that the author must have been either short of material, or there was a narrow deadline to keep.

Maybe both of them ?

Because something of a world record in spaces and abundant code must have been hit here. For example, the print out function being identical all the time, is being repeated at the end of every example.

Furthermore, because it's Java this is about, and because the target group is not novices, in most cases 5-10 lines of code would be sufficient. Instead 1-2 pages are used for every single code sample. Even worse is that some of the introducing examples import abundant code. For example you can find this

```
import java.util.regex.*;
```

Note that the imported-by-default `java.lang.string` is enough to compile and run the actual sample! Can seem to be trivial, but is still confusing when learning a new Java concept.

An other known trick to increase the number of pages is to copy-and-paste a language API, and pretend it to be a chapter of its own. That trick is tried here at least half ways. The `java.util.regex` API is inserted after a very long introductory chapter. However, some useful explanations for that package are included, not to be found found in the original Java doc.

So, there are glimpses of light in the book.

There are accurate and detailed step by step explanations following all regex syntax. And the pedagogic approach to advance from very simple to finally advanced examples is a good one.

Maybe much of the problems is not about the content itself ?

One rather gets the feeling of a movie uncut by the director, or why not a manuscript released while still being a draft.

The topic at hand, the excellent support for regular expressions from Java 1.4 is appealing indeed. An alternative (and cheap) way of learning regular expressions is to use the excellent, free tutorials on http://www.regular-expressions.info/.

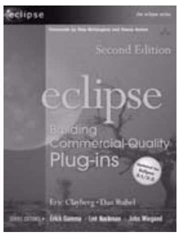
And if you want a book, *Mastering regular expressions* from O'Reillys is the Bible. Now in its third edition (2006) and expanded to include special chapters for Perl, NET, Java and more.

At the best this title can aid as a complement when learning regular expressions, but not more.

Eclipse: Building Commercial-Quality Plug-ins (2nd Edition)

by Eric Clayberg, Dan Rubel, published by Addison-Wesley Professional; 2 edition (March 22, 2006), Paperback: 864 pages, ISBN: 032142672X

Reviewer: Derek M Jones



Eclipse is an open source IDE written in Java. The architecture of the system is such that virtually all of its functionality is implemented using plug-ins. This book claims to help programmers build commercial-quality plug-ins. Presumably commercial-quality is meant to imply high quality, but other that a discussion on how to test plug-ins this book does not discuss commercial or quality issues.

This book will appeal to people who like lots of hand holding. I could have done without being told how to navigate down a list to find the option I needed to click (in some cases accompanied by a snapshot of the menu hierarchy). The over 800 pages could have been significantly reduced by removing a lot of superfluous code snippets and the oh so many two sentence synopses of API methods (this is what online documentation is for).

The book covers version 3.1 and 3.2 and I suspect that its detailed navigation tips and screen shots might not be portable to later versions.

The text is readable, if somewhat long winded, and there is an extensive index. Distilled down to 200 pages this book could be a useful accompaniment to the online documentation.

Internet and Email

The Google Story – Inside the hottest business, media and technology success of our time

by David A. Vise, ISBN 0-4050-5371-2.

Reviewer: Ian Bruntlett

This is a business story, not a technology HOWTO. It tells of Google's remarkable success, the uncanny business sense that keeps it one step ahead of Microsoft and offers snippets of information about the company. One day in the future, someone will write a book on how Google's army of PCs cooperate to provide search results so ably.

As well as telling the Google story, it describes how Brin & Page are setting out to improve the world we live in, armed with billions of dollars. They've so many novel ideas that I can't list them all – buy or borrow this book to find out more.

As a bonus to the book, Appendix I provides useful Google search tips, Appendix II the Google Labs Aptitude Test and Appendix III Google's Financial scorecard.

VERDICT: Recommended.

Cyber Spying

by Ted Fair et al, published by Syngress, ISBN 1 931836 41 8, 439 pages

Reviewer: Silvia de Beer

I was severely disappointed in this book. From the title of the book I guessed that this book was about how to protect your computer from being spied upon, e.g. by Internet spy ware etc. This is not the case, the book is basically about how to spy upon other people in your household, how to spy upon their internet use, what email they type, what messages they send with various instant messaging services. These other people can either use the home computer, or use their own laptop or computer on the home network. This book was like reading the Marie Claire, and not technical at all. It was about how you could suspect your spouse having an extra-marital affair, or how you could suspect your child of drug abuse etc. An indication of the Marie-Claire style of the book is the many examples like "Between their junior and senior year Greg confessed his love and proposed to Camille. She accepted...."

The book explains various ways how you could read someone's emails without him knowing, if you know his password, how to guess his password, how people use the same login for various accounts. To indicate the technical level of this book: chapter three of the book explains what a directory and a file are. The book talks about various software tools like keystroke loggers and sniffers, how to remove programs from the start menu or uninstall list by editing the registry, how to cover your spying. Playing around with the registry editor does not seem to be a good idea to me if you have only as little technical knowledge as the book assumes. Chapter 10 (Advanced Techniques) explains what ARP (Address Resolution Protocol) spoofing is, possibly un-understandable by a non-technical reader.

Web Accessibility: Web Standards and Regulatory

Compliance by various, published by Friends of ED, ISBN 1-59059-638-2

Reviewer: Silas S. Brown

The chapters of this book were contributed by many different authors, and the type, quality and relevance of the content varies enormously from chapter to chapter. Although there are passing references from some chapters to others, overall it feels more like reading a set of conference proceedings than a unified book. The authors sometimes seem to disagree with each other and this can be confusing. For example, should one's Web accessibility effort revolve around lengthy law-book stuff, or would it be better to understand the underlying issues? Is there any point in automated testing? Does 'accessibility' include catering for a variety of devices and operating systems, or just for expensive commercial Windows-only screen-reading software? Should 'skip to content' links be visible or not? It all depends which chapter you read, and the book's target audience may find that unhelpful.

There are some gems in this book if you can find them. The chapter on stylesheets is good and the chapter on accessible scripting has some good points, and the introduction is very reasonable if you can put up with the American jokes. But why do technical books have to be two inches thick to sell? Couldn't some of those extra chapters be more concise and to the point? Couldn't there be more co-ordination between them? (Chapter 16 tries to tell you what HTML is as if you didn't know anything about technology, but would you have got that far if you didn't?)

The cover prominently says 'Mac and PC compatible', so why is so much of the text Windows-only? There is much discussion of Windows accessibility software (and some of bundled features), but absolutely no mention of the Mac's accessibility, which has improved greatly in recent years and ought to be at least acknowledged if you're going to write on and on about Windows in a book with a 'Mac' label on it. And although IBM people were involved in the book, mentions of Linux are infrequent, brief, and arguably inaccurate.

The printing could be better too. Why not reduce those big margins to allow for a clearer font? Some figures are confusing, and it seems the authors weren't told they'd be printed without colour. There are misprints too. And a label on the front says that an accessible HTML version of the book is available, but they won't tell you the URL; they just tell you to go hunting for it on the publisher's site. This practice is a great example of NOT helping accessibility. When I did manage to find it, I was told that I had to pay for it (having obtained a paper copy of the book doesn't exempt this).

I must point out the following piece of extremely bad advice. Chapter 12 (which is about using Adobe's tools) recommends OCR-ing court documents as 'searchable images', which means the results of the OCR are hidden behind the original image. This is great for searching, but it is extremely risky practice if you're involved in making court documents accessible, because it amounts to running them through an OCR program without even glancing at that program's output before you give it to someone who depends on it. Would you want that done to your legal documents?

I don't want to give this book the 'not recommended' tag because it does have some good points. But sadly it has serious bad points too, so be careful.

SpamAssassin

by Alistair McDonald (2004) ISBN 1-904811-12-4 pp221

Reviewer: Mark Easterbrook

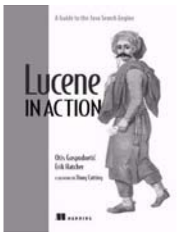
Conclusion: Recommended

The subtitle of this book is "A Practical Guide to Configuration, Customisation, and Integration" and it lives up to this admirably. The first few chapters cover what is Spam, where it comes from, how it is sent, and methods used to detect and classify it. All this provides grounding so that by chapter six – Installing SpamAssassin, the reader is well acquainted with the characteristics of spam and how to use SpamAssassin to fight it. There are two sides to a good anti-spam installation: integration with the mail system in use and the configuration of the spam rules. For the former the book provides a step-by-step guide for the most common mail system and should prove no problem as most of these are fairly mature products. In contrast, configuration of SpamAssassin is addressing a rapidly moving target requiring constant attention paid both to the profile of spam and ham the users are receiving, and the type and methods of unwanted email being generated. The book does a good job of covering the topic, but an efficient anti-spam filter requires knowledge of how spam has evolved since the book was published and how it will evolve in the future. Although SpamAssassin is a best-of-breed today, it is a victim of its own success and already the spammers are working out how evade it, so it is possible that the book could suddenly become very dated. Until that happens, this is an essential book for anyone trying to hold back the flood of unsolicited email.

Lucene In Action

by Erik Hatcher and, Otis Gospodnetic, ISBN: 1932394281 Paperback: 421 pages

Reviewer: Derek M Jones



Lucene is an open source search engine written in Java (it requires that the pages to index and search have already been collected, eg, by a web crawler such a Nutch). To be exact Lucene is a search engine library containing classes and methods that programmers can call to create their own customized search engine.

This book is essentially a 'how-to' guide for creating a search engine using Lucene. I found it to be very readable and the extensive code-snippets were on the whole useful (ie, not just padding).

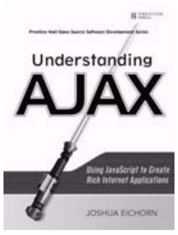
The discussion starts with how to index web pages, the various tradeoffs involved and how the various Lucene options can be used to tune an index to have the desired characteristics. This is followed by a very interesting discussion of how to parse the search queries, dealing with issues such as what constitutes a token and possible ways of dealing with various forms of the same root word (e.g., past/future tense, singular vs. plural). Subsequent chapters deal with more advanced topics including extending the search engine, performance testing, parsing common document formats and the book ends with a discussion of various applications (written by people involved with implementing these applications). The thickness of the book is kept down by not duplicating the online documentation by including a detailed listing of the API.

If you are building a search engine using Lucene this book is a must have. Even if you don't plan to build your own search engine this book provides a fascinating discussion of the nut-and-bolts issues involved in creating one.

Understanding AJAX: Using Javascript to create rich internet applications

by Joshua Eichorn, published by PrenticeHall

Reviewer: Tim Pushman



There are many technologies that get hyped by the media and Ajax even had a catchy name that didn't sound like an acronym (although it is). But even more impressively, it was a real technology that worked, in fact had been in use before the media discovered it. It was first described nearly two years ago and the defining application of it is the GMail webmail site. Since then a rapidly growing industry of frameworks and sites have sprung up based on Ajax.

Joshua Eichorn has been using Ajax since its inception and has an excellent grasp of the concepts involved. Although the book is about using Ajax, another title would have been Building Rich Internet Applications, as that is his primary focus in the book.

The book starts by giving a background to Ajax and how it fits into existing web development, not as something completely new, but to complement and improve what we already do. There is quick run through of making some simple home brew Ajax enabled pages using various techniques for passing data back and forth to the server before moving on to integrating it into existing applications.

One of the common threads throughout Eichorns book is using the technology to improve the users experience and there a couple of substantial chapters devoted to that with a strong focus on usability.

Part 2 of the book describes a selection of libraries and frameworks for Ajax. Eichorn is a developer of one of the libraries mentioned but he doesn't favour it in particular, the treatment of all libraries is fair. There are three small sample applications developed here, using a selection of the libraries facilities. It's this section of the book which could be dated quite quickly in that the libraries could disappear or be superceded, but the descriptions are written in a way that also details the decisions made on what to use and how to implement it. And as a developer himself, there is a good section on debugging the applications.

Finally there are three appendices giving a brief rundown of the existing libraries and frameworks, their strengths and weaknesses, with web site addresses. This is probably a good starting place for choosing a library. He also lists the licenses they are distributed under, an important point.

The book is well put together, with very few typos and edited to a high standard. The index is a little thin but usable (you get access to the searchable online Safari edition for 45 days when you purchase the book). Eichorn's writing style is very clear and readable. He does assume that you have a good understanding of HTML, DHTML, Javascript and HTTP protocols. All the server side code is in PHP but should be easily transposed into Perl or Ruby if needed.

I would have liked a little more on the internal details of how Ajax works but that is not essential for creating Ajax web sites. On the other hand, I appreciate the fact that he hasn't felt to pad it out with descriptions of how to setup your webserver and what HTML is.

My other reservation is that some of the chapters on specific libraries will be out of date quite quickly. This is nearly half the book, but on the other hand he has picked mature libraries that will probably be around for a while, so that may not be too much of a concern.

If you have a clear understanding of web development and want to start using Ajax on your web sites, this book is Recommended.

Miscellaneous

NO-NONSENSE guide to SCIENCE

by Jerome Ravetz, published by Verso in association with New Internationalist/Amnesty International, ISBN 1-84467-503-3

Reviewer: Ian Bruntlett.

This book discusses science from the anti-science lobby to the pro-science lobby, discussing this spectrum with enviable objectivity. It quotes Bill Joy's GRAIN acronym – standing for Genomics, Robots, Artificial Intelligence and Nanotechnology. The downfall of science is given an acronym, M&M – Malevolence & Muddle – and its counterpart, SHE – Safety, Health & Environment – is introduced. The old, applied science is shown to have relatively few uncertainties whereas a new extreme PNS – Post-Normal Science – is introduced.

Conventionally, science is taught as a subject that moves inexorably from one discovery to another. This book details some of the errors made by key scientists in history – this does not belittle them, it makes them more human and makes their discoveries shine even brighter.

The industrialisation of science from small science to big science to mega science is discussed. The biopiracy committed by some unscrupulous corporations is discussed – misuse of science is an issue that cannot be swept under the carpet these days.

Although computer science may or may not be a science, ICT may be moving from being an "applied science" to a "Post Normal Science" (PNS) where the stakes and uncertainties are high.

The role of political structure and high finance are discussed in a chapter on "Science and democracy" which has the following set of questions about new technology:

- Who needs it?
- Who will benefit from it?
- Who will pay its costs?
- What happens when it goes wrong?
- Who will regulate it, how and on whose behalf?

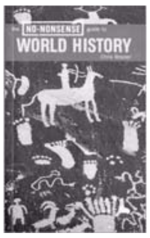
This book concludes with the old assumptions, the political questions and a table of personal questions so the reader can evaluate their own contribution to science.

VERDICT: An interesting read

The NO-NONSENSE guide to WORLD HISTORY

by Chris Brazier, published by published by Verso in association with New Internationalist/Amnesty International, ISBN 1-85984-355-7

Reviewer: Ian Bruntlett.



This is an attempt to tell the history of the world in 40,000 words (136 pages plus a 7 page chronology). If, like me, your history lessons covered Ancient Greece, Imperial Rome, Persia as Ancient History and the history of Europe as, well, History, this book is full of things your History teacher didn't tell you about. This book will also tell you about civilisations in Africa, China and India and how the far flung reaches of the globe were colonised by people crossing "land bridges", formed when the sea levels lowered or by conventional sailing, long before Europe "discovered" the rest of the world.

There are a lot of startling facts/theories to learn from this book, so much so, I'll end this review to say "buy it and have your misconceptions blown away".

VERDICT: An interesting read.

Electronic Brains / Stories from the dawn of the computer age

by Mike Hally, published by Granta Books, ISBN 1-86207-839-4,

Reviewer: Ian Bruntlett.

This book charts the development of the first electronic computers in parallel across the world after World War II. Contrary to popular belief this took place in Britain, America, Australia and the then Soviet Union. It also discusses the development of mechanical computers.

First of all the mechanical computers are chronicled – the Codex Madrid, Pascal's Arithmetic Machine, Babbage's Engines, Scheutz's machines, Hollerith''s tabulators and Zuse's machines.

The early computer applications were either military (shell trajectories etc) or weather forecasting. Also computing started off with "controversy, falling-out, setbacks, smears, financial mis-management" – something the industry still hasn't grown out of. Later on the census departments and payroll departments all started using computers.

Chapter 4, “When Britain led the computing world” describes the work of many but Alan Turing and Maurice Wilkes both stand out from the crowd. This was a temporary advantage, soon to be outstripped by America’s economic might.

Chapter 6, “Leo, the Lyons computer” describes how Lyons started as a wholly owned subsidiary of a tobacco company. Lyon’s the tea company had a company culture of self-sufficiency in order to maintain high levels of quality. So it is not surprising that the company built its own computers for data processing work – so the LEO (Lyons Electronic Office) was born,

This book also discusses the parallel invention of computers in the Soviet Union – in the Ukraine the MESM (Small Electronic Calculating Machine) was developed from the ground up with little in the way of resources. Also in the USSR another system, the STRELA was developed. The M-1 was developed using copper oxide diodes, making it probably the worlds first computer to use semi-conductor logic. Not only was the USSR not benefiting from cross-fertilisation but the USSR’s different teams were more interested in with competing with each other instead of co-operating. Innovative Soviet computing came to an end when the powers that be made a political choice to go for IBM 360 compatibility(!).

Chapter 7, “Wizards of Oz”, describes the sometimes parallel evolution of computer facilities in Australia.

Chapter 8, “Water on the brain” discusses the MONIAC, a hydraulic computer that graphically illustrates the flow of money in an economy, developed in New Zealand by Ben Phillips. He made ground-breaking work in economics in his career.

Chapter 9, “Its not about being first: The rise and rise of IBM” describes the conditions that Thomas J. Watson worked under at NCR (a rapacious company if there ever was one) before setting up IBM, after he got his hands on one of his employer’s companies, CTR (Computing-Tabulating-Recording), in the late 19th century. In 1924 CTR was renamed IBM and all of Watson’s experience shaped the development of IBM. Watson demanded absolute loyalty from his staff and gave them absolute loyalty in return. The global success of IBM’s machines resulted in their subsidiary, Dehomag, providing the administration equipment used to organise the Holocaust.

After the war, Thomas Watson Jr took over the reins from his father and moved IBM away from mechanical systems to electronic computers, starting with the IBM 701. Later on he bet the farm on a family of compatible computers, the IBM 360 and the company came to dominate the market – even the Soviet Union, in the cold war, the authorities scrapped their own designs and produced their own IBM clones.

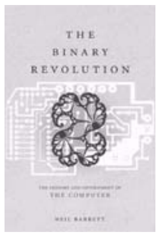
For readers who want to delve deeper into the history of computers a bibliography is provided.

VERDICT: Highly recommended.

The Binary Revolution – The history and development of the computer

by Neil Barrett,
ISBN 029784738-4.

Reviewer: Ian Bruntlett.



This book tries to answer a number of interesting questions about computers and our culture(s). They are “What is a computer?”, “How did computers develop?”, “How do computers work?”, “How are computers programmed?”, “What do operating systems do?”, “Where did the internet come from?”. It also tackles posers such as “Putting the internet to work”, “Problems with computers” and “The future of computing”. It is an ambitious book, dealing with underlying foundations and principles, the maths and physics involved when a computer is being used.

It gets one or two things wrong, though. It refers to PCs being 32 bit, overlooking the 64 bit revolution (the Intel Itanium or PowerPC or AMD64 should have given the author some kind of a hint). It also states that the Apple Macintosh was 6800 based – it wasn’t, it was actually based on the 68000 based family. One of its sources of information is www.wikipedia.org

I like this book and the examples did seem clear when I read them but that’s probably because they're already preaching to the converted. The true test of this book’s mettle is when you give it to an inquisitive friend or relative who wants to understand computing. And, if you know someone who is going to study computing as a prelude to a career in the industry, it would be kind of you to give them a copy of this book.

Verdict: Recommended but please do note that it does make the occasional mistake.

Serious Creativity – Using the power of Lateral Thinking to create new ideas

by Edward de Bono, published by Harper Collins, ISBN 0-00-255143-8

Reviewer: Ian Bruntlett.

This book is split into three parts - 1) The need for creative thinking, 2) methods of lateral thinking and 3) applying creative thinking.

de Bono argues that brainstorming, while initially useful, is no match for the application of better creative thinking techniques. His work is built on the premise that the brain is a self-organising pattern-making system. His goal is not to improve the skills of artistic creativity but to deal with the creative skills that change concepts and perceptions via the behaviour of self-organising information systems. He argues that creativity is not just making things better but is a way to make full use of information and experiences.

Conclusion/Verdict – buy and study his previous work, “Lateral Thinking” by this author first. Then, when you want to move on, buy this book – it is an excellent Post-Lateral Thinking book.

Enterprise Services for the .Net Framework

by Christian Nagel, publihsed by Addison Wesley, part of the Microsoft .Net Development Series, 539 pages

Reviewer: Simon Sebright

In short, for the target audience, this book is recommended.

It is about the features of .Net and the Microsoft operating systems which allow enterprise-scale applications to be built, with such considerations as distrubution, performance, fail-safety, transaction integrity, and so on. Chapters cover concurrency, .net remoting, COM interop, data access, transaction services, compensating resource management, state management, queued components, loosely-coupled events, security, and deployment.

There are 15 chapters, the first being an introduction, 14 being a forward look at what’s coming, and 15 being a case study. Throughout the book, code samples are given to illustrate the concepts being described. In general, I found these to be straightforward, and it was good that they all formed part of the larger case study (although there was potential for confusion in places as similar class names were in use for different parts of the system). I did not actually try any of them, or the whole case study, but my impression was good.

Starting off, I got a bad feeling when in chapter 1, all sorts of concepts were thrown at the reader. It didn’t seem to make much sense, and I began to think it would be a difficult 540 pages. But, after that, the remaining chapters were good, each explaining one particular feature, what, why, how and with simple code examples. Options available in each case were clearly explained, and I got a good understanding of what each chapter was saying.

Some experience of the .net environment and programming in it is necessary, as there is a lot of detail to cover. On the other hand, if I were to decide to implement a system such as a queue component, I think I would use this book as a springboard, and find other sources of information to give me a really detailed picture. From reading the book, I would feel fairly comfortable about using the MSDN as a primary source of detailed information, but reserve the right to order a more specific book.

One minor niggle was that in each double-page, the left page has at the top the name of the book, the right page the name of the chapter.

Personally, I generally know what book I am reading, so would have preferred the more detailed sub-chapter section to be available there.

Also, like most books I have read recently, the index was not great, more of an alphabetical list of terms, instead of a comphrensive matrix allowing me to find something even if I didn't know exactly what it was.

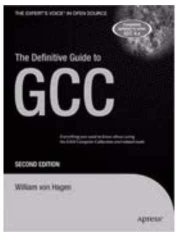
I couldn't even see a reference to Visual Studio in there!

Still, the book is recommended.

The Definitive Guide to GCC

by Willian von Hagen, published by Apress, 550 pages, ISBN: 1-59059-585-8

Reviewer: Pete Goodliffe



Rating: Recommended

This book is a rarity in the market right now: a tutorial on how to use the popular GCC family of compilers. Given the remarkably large user base of GCC (which is due to its quality, its open source licencing model, and – probably most significantly – its price), it's surprising how few books there are about it. Hagen provides a fairly comprehensive 'howto' for GCC; this is not a language tutorial.

There is already a large body of adequate free documentation on the net about GCC, and all the answers are usually available for patient people who are competent with Google. But it's always useful to have this information collected into one well-organised authoritative source, and that's what we have here.

This is a large, comprehensive book. It's easy to read: well written with a clear layout and sensible organisation. It has a useful index, which is very important in a reference book like this.

Unfortunately, it's fairly Linux-specific. It's a real shame not to include more information of the flavours of GCC installation available on Windows, and this limits the book's market somewhat.

This book is now in its 2nd edition. It has been updated to cover the GCC 4.x series of compilers (which are vastly improved over the 3.x series described in the first edition). This update strips out all coverage of the previous versions, which is actually a shame; some details on older GCC versions might have been a useful aside – they still see a lot of use in specific installations and will for some time yet.

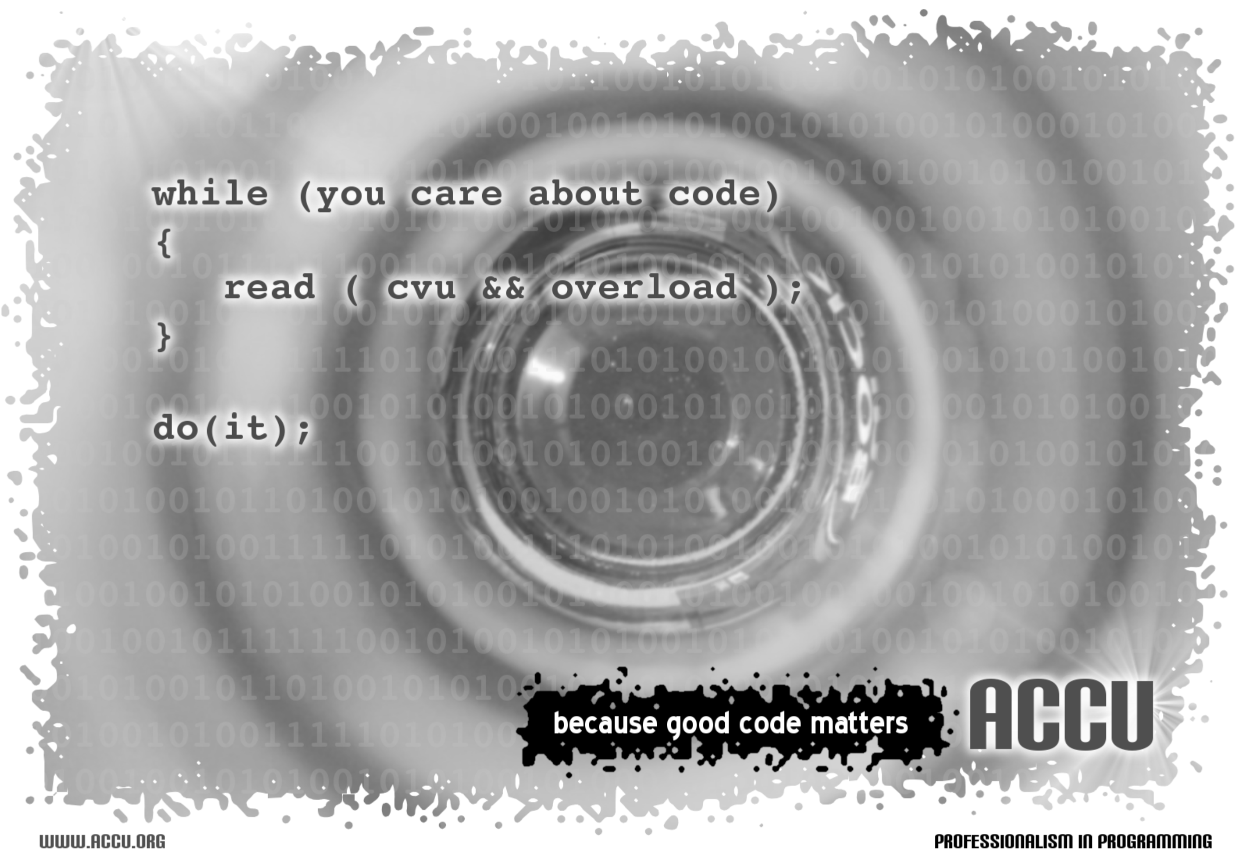
This edition contains new material, and presents the content in a far better order (for example: the chapters on building and installing GCC have moved from the very beginning towards the end of the book; most people have GCC available, or can install it very easily, without needing to build manually anyway).

Unlike first edition, there is a separate chapter for language supported by GCC (C, C++, Fortran, Java – the book pragmatically skips Ada coverage). Whilst this is undoubtedly simpler for someone who is only interested in a particular language, there's an amount of duplication inherent in this approach, and this is where a lot of the book's bulk (literally) is held.

The two chapters on autotools (`autoconf`, `automake`, and `libtool`) have moved over from the first edition. I think that these really shouldn't have been included at all. This is a book about GCC, not about the arcane twists involved in packaging software for random Unix-like operating systems. The information just isn't relevant and is mostly fluff. There are better books on autotools than the 46 pages of coverage provided here.

However, the second edition adds some advanced topics that are genuinely useful: building and using cross compilers and alternative C libraries. These chapters are good forays into the subject, as far as they go. But if you're doing anything so complicated that you need to read up on this (usually some form of embedded work) then these chapters alone will not be sufficient. They do, though, serve as a very good introduction and will set up for some more intelligent web searching and experimentation afterwards.

If you're using GCC 4.x (but not for Ada) and would like a printed reference then this book comes recommended. If you're happy with the documentation that comes with GCC then this book isn't a must-have. If you're considering GCC for embedded applications then this book might provide some useful insights to get you going.


while (you care about code)
{
read (cvu && overload);
}
do(it);
because good code matters
ACCU
WWW.ACCU.ORG
PROFESSIONALISM IN PROGRAMMING

ACCU Information

Membership news and committee reports

accu

View From The Chair

Jez Higgins
chair@accu.org

It seems to have become the tradition for columnist, pundits, sundry celebrities, politicians, and so forth, to issue New Year's messages. Looking back at the year gone by is a popular subject, as is predicting what might happen in the year to come. The laziest (or most unimaginative) simply throw out a list of resolutions.

This is the fifth View From The Chair I've written, and so far none have reached the editor before the copy deadline. In fact I don't think any of them have actually been written before the deadline. This year I resolve to get at least half in on time.

I've been saved from the embarrassment of finishing there by being able to announce that bookings are now open for accu2007, the 10th anniversary ACCU Conference. We (and by we, I mean Ewan and the conference committee, not that I've started referring to myself in the plural) have, as ever, a enormous pile of great stuff lined up this year, from keynotes featuring Mary Poppendieck, Mark Shuttleworth and our own Pete Goodliffe, through another diverse and highly engaging programme featuring tracks on development for mobile devices, dynamic languages, Java, C#/.NET, and two days discussion of the Future of C++ from the world experts gathering for the ACCU-hosted ISO standards meeting taking place the following week. The closes with a rare appearance from Dan Saks, one of the speakers at the first conference in 1997.

This year's conference will be at a new location, the Paramount Oxford Hotel. Moving away from the Randolph wasn't an easy decision, but the Paramount's modern conference centre with ample public space, not to mention air-conditioning should, at the very least, make the event more comfortable for everyone. Although my own racquet skills are minimal, I'm hoping to organise an emacs vs vi squash tournament. That should put that particular question to bed once and for all.

While I'll be travelling to Oxford by train, I know many of you will be pleased to hear the Paramount not only has a car park, the car park is large enough to actually park your car in.

Full programme and booking details are on the website at http://accu.org/conference.

It's going to be good. See you there..

Membership Report

David Hodge and
Mick Brooks
accumembership@accu.org

Please note the change of email address for membership, all email to the old address will be soon be dumped in the trash bin (99.99% spam).

Membership at the end of 2006 was 38 up on last year at 884.

Yechiel Kimchi recently asked on the Accu-General list where he could find an electronic list of ACCU members. The answer is that currently he can't – the information is only made available to members in the 2006 Members Handbook. His query was well-timed, since the committee have been discussing this situation recently, with a view to making a proposal at the next AGM (in April, during the conference). The issue is whether an electronic copy of the information should be made available to members, and, if so, whether the paper handbook should be printed and distributed at all. We'd like to hear what the membership feel about this, so we can make a representative proposal.

Currently the handbook is printed once a year, and each member receives a copy. It has the names of every member of the association, and contact details (postal and email addresses) for those who haven't chosen to withold this information. Use of this information for commercial purposes is forbidden. How would you feel if this was made available as a PDF on a members-only section of the website? What if the info were made searchable, so that you could easily find all the members in your town, or country? If you don't currently withhold your details, would you feel more inclined to do so if they were going on to the website? Do you make use of the printed handbook, and would you miss it if it were to go?

Any of your thoughts on this would be much appreciated: raise them on accu-general, or please email them to me (mick.brooks@gmail.com), David or any member of the committee.

Website report

Tim Pushman

A year's end look at the statistics for the accu.org site. Not for those with an allergy to numbers!

The site was reborn about 10 months ago, in February 2006, and at the end of the year I take a look back at how many folks dropped by and where they came from.

Altogether about 85,000 visitors have passed by and read a total of 920,000 pages, resulting in 4,250,000 hits and pulling down about 20Gb of bandwidth.

Monthly averages vary between 6,000 to 8,000 visitors a month. The average visitor reads about 6 pages.

Saturdays are the low days for the site, with only about 20% of the visitors of a weekday, although we have a few Sunday surfers, with Sunday getting about 40% of a normal day. Overall our visitors are definitely surfing from work. Most visitors are from Britain, Europe and the USA, with quite a few from India and Australia. Since putting the journals online they've been downloaded about 4,500 times, the most popular being Overload 75 (700 downloads) and Overload 74 (470 downloads).

The most popular area of the site is definitely the book reviews, with about 150,000 page views.

None of these stats include robots and spiders, which have, between them, indexed 720,000 pages and used 4.7Gb of bandwidth.

Alexa ranks the accu.org at about 400,000 (which given the millions of sites, is respectable). Speed of the site puts it in the top 30% with an average page load time of 1.4secs. You can see the ACCU details here.

In June I split off the access gateway into its separate subdomain, partly in order to keep the statistics separate. The gateway transports about 1Gb of data every month except during school holidays, when it drops to about half. And almost nothing on weekends. USA and the UK predominate here, but there are also a lot of users from Spain and Saudi Arabia.

Overall 80% of our visitors claim to be using Windows, with about 7% on Linux and 2.7% on Macs. Internet Explorer is used by 60% and Firefox by 25%.

I hope that in 2007 we can persuade even more visitors that we have something worth looking at and that it is a useful and worthwhile thing to join the association. There is a lot that can be done to make this site a valuable resource for programmers wherever they are and whatevery they program in, it just takes a few volunteers to step up and help out. So, if you have some spare time or even just some ideas, let us know!