{cvu}

Volume 18 Issue 3
June 2006
ISSN 1354-3164
www.accu.org

Once More from the Breech

"Well, Mrs Miggins, at last we can return to sanity – the hustings are over, the bunting is down, the mad hysteria is at an end. After the chaos of the ACCU conference, we can return to normal"

– Edmund Blackadder, butler to the Prince Regent.

By the time you've had this edition, the annual conference will be but a memory. For some, it will be a curry soaked, beer smelling one, but a memory none-the-less. This conference was special, though. For the first time, it was a total sell out. In all of my time of being a member of the ACCU, I cannot recall such an unmitigated success. My only regret was not being there – judging by the comments and reports back, it was possibly the best it has ever been. Plans are already afoot for the 2007 conference and with a bit of luck, Oxford council will have a brain wave and make travel into the centre of the town much easier by rewinding the clock a couple of years so that you don't need to get back to the car every night to avoid parking fines.

I have a question for you – and it's quite a serious one. How many of you out there look after code written many moons ago when the C++ standard was nothing more than a rumour, a standard C compiler was something you laughed about down the pub and C# was just a note on a piano keyboard?

The reason I ask this is that I've been handed a pile of very old C. The code itself will compile using gcc 4.1 and actually runs quite well. However, it's come time to alter the code and much to my annoyance, I've found that looking at the archive the source is held in is enough to break the software. The question is, what should I do with it? I can't leave the code in its current state as it's no longer fit for purpose; however, I don't have time to rewrite the code in anything more than a quick hack style. If I'm *really* careful, I might be able to bolt something onto the side of the existing program that is correctly coded...

This is the dilemma. If I start to bolt code on, not only do I have to try and find the safest place for it to instantiate from, but I run the added risk of breaking something somewhere else – all because the original coder thought comments and documentation was for losers and besides, the author was immortal and would never lose interest in a 6502 cross compiler...

Of course, there is one more alternative, but it involves the movement of large quantities of foaming nut brown liquid. I wonder if mixing this alternative with bolting on the code will work. Must try it sometime....

PAUL JOHNSON,
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

REGULARS

FEATURES

COPY DATES

C Vu 18.4: 1st July 2006
C Vu 18.5: 1st September 2006

IN OVERLOAD

Overload 73 is stuffed full of relevant, high quality articles including Rachel Davies on Pair Programming and Thomas Guest on the use (or misuse) of TODO. Overload is available to all full ACCU members

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

Hello Groovy

Russel and "Hello World" introduce the Groovy language.

The plethora

Hello World (HW) seems to have first represented itself on page 6 of *The C Programming Language* by Kernighan and Ritchie (1978). It appeared as:

```
#include <stdio.h>
main ()
{
  printf ( "hello, world\n" ) ;
}
```

A short representation, but then short does not mean bad. In fact, these days, for a given functionality, short is definitely considered good, and long bad.

HW has since represented itself in many languages for many contexts. A quick perusal of the website http://www2.latech.edu/~acm/HelloWorld.shtml shows HW in 193+1 (193 listed and 1, Groovy, submitted) different languages – at the time of writing anyway, HW may have represented itself further in the interim. Although HW is hugely infectious, it is not a virus.

Interestingly, HW appears to have a sense of humour. It has represented itself in C for X, Gnome, KDE even Windows, presumably to prove that console output is simple and GUI output hard. For example, *helloX.c* (see http://www.paulgriffiths.net/program/c/hello.html) is a representation of HW in C for the X Windowing System. This representation is 135 lines of code, 25 of which are comment lines. The coding doesn't seem excessive but it is far to long to show here – size-ist maybe but there we are, space is at a premium.

So HW has a habit of representing itself in unusual and often bizarre ways to make a point about a programming language. HW is clearly a teacher. Though not certificated as the possession of a qualification might be considered pretentious – especially amongst some people who contribute to the accu-general mailing list. Being un-certificated doesn't make HW a bad teacher, but it will make it difficult to find a position in a school. Universities though are less strict about teachers having qualifications and so HW may still be able to find a position there, possibly in textbooks.

Unfortunately, despite the various anti-discrimination laws, publishers are now definitely discriminating against HW, generally on the grounds of boredom and "oh no, not that flaming program again". This has not daunted HW, HW is resourceful. Not only does it look for new outlets, this article for example, it is always on the lookout for new programming languages to represent itself in. For this article, HW has represented itself in a number of short ways to show how groovy the Groovy programming language is.

Groovying on

When Sun introduced Java to the world in the mid 1990s, they also introduced the Java Virtual Machine (JVM). Sun is reputed to have stated that the only programming language for the JVM was Java. However, it seems that this may be a myth. Myth or not, many people thought the idea short-sighted and overly introverted and started developing other languages targeted at execution on the JVM.

People ported Python (Jython) and Ruby (JRuby) to work as dynamic programming languages (aka scripting languages) for the JVM. However, whilst these languages have a following, they have not caught on in any big way. It may well be that the lack of catching on is because Python and Ruby are based on different types to those supported by the JVM, leading to the need for type/representation conversion. However, the idea of a dynamic programming language for the JVM has caught on. Big Time. We have BeanShell, PNuts and, of course, Groovy, to name the most well-known three. HW has almost certainly been represented in all of these languages even if not listed on the HW site so that probably means 193+3 rather than the figure quoted earlier.

The success of "scripting languages", aka dynamic languages, working symbiotically with Java has recently caused Sun to publicly state its enthusiasm for mixed language working on JVM-based systems. This shows Sun are not averse to doing and thinking the right thing even if they didn't originate the idea.

So where does HW stand on all this? Actually, it is totally agnostic, it cares not a jot about language wars, it will represent itself in any language. Java is one example that is apposite for this article:

```
public class HelloWorld
{
  public static void main (
   final String [] args )
  {
    System.out.println ( "hello, world" ) ;
  }
}
```

Of course this is horrendously verbose and complicated, especially if this is the first program you see when learning a programming language. It is hard to argue against this point when you observe HW representing itself in Python:

```
print 'hello, world'
```

and Ruby:

```
puts 'hello, world'
```

Compare these to HW's first representation in Groovy:

```
println ( 'hello, world' )
```

and we see that Groovy is as good as Python and Ruby and a lot better than Java, even on Java's own turf.

The many Groovy ways of HW

As mentioned earlier, HW, being a program with a wry sense of humour, has come up with a series of representations of itself designed to show various features of the Groovy programming language. These are not serious representations – we have already seen the serious version above – they are simply presented to show that Groovy is, well, groovy.

The state of variables

Groovy has variables and so the obvious first variant representation of HW in Groovy is to use a variable to hold the string value to be printed and then to print the value of the variable:

RUSSEL WINDER

Having done the Professor of Computing Science and Head of Department thing, I tried the CTO thing. Just as things were going well the accountants closed the company. I am now doing the author, trainer, consultant thing – all reasonable jobs considered.

```
String theString = 'hello, world'
println ( theString )
```

This representation of HW uses static (compile-time) typing. Groovy being a dynamic language does not need compile time type checking, HW can therefore use a variable of type determined at run time:

```
def theString = 'hello, world'
println ( theString )
```

The type of a dynamically typed variable can change during execution since it is the type of the last assigned value that determines the type of the variable. Some people find this hard to deal with and stick to static typing always. However this is not dynamic and not Groovy. Having said this, even in a really Groovy program not all variables are dynamically typed, it is often right and proper to have statically typed variables. Nonetheless, using **def** and dynamic typing is the usual way of working with explicitly initialized variables since it avoids repetition of information - the literal being assigned has a type so why specify the type again? Coupling **def** with the use of **final** to enforce single assignment, leads to a declarative style of programming which is considered Very Groovy.

Languages like Python and Ruby do not require specific declaration of variables, they just have declaration by usage. HW, being a good teacher, answers the question "What happens in Groovy?" by trying the experiment and representing itself:

```
theString = 'hello, world'
println ( theString )
```

The program still works. Why? Each Groovy script has a binding object associated with it, so in this representation of HW **theString** refers to a variable in the binding – if there is no variable of that name already in the binding then one is created. Usually the binding is used for passing information from operating system to script, or embedding system to embedded system, but it can be used as a global shared memory. OK, global shared memory is generally considered bad (quite rightly except in certain very specific circumstances) so using the binding in this way is definitely not groovy even if it is Groovy. HW raises the issue simply to show there is an issue.

The above variables were scalars, HW now represents itself to show that we can have arrays of things:

```
String [] theStrings = [ 'hello' , 'world' ]
println ( theStrings [0] + ',
          ' + theStrings [1] )
```

The literal on the right hand side of the initialisation is not actually an array, it is a list (**java.util.List** – well **java.util.ArrayList** actually but, as in Java, we use the interface and not the class). However, because of the static type of the variable is specified as array there is an automatic coercion from **java.util.List** to array.

What happens if we have a variable of dynamic type? In this situation, to ensure the assigned value is of array type, we have to manually coerce as HW shows in this representation:

```
def theStrings = [ 'hello' , 'world' ]
  as String []
println ( theStrings [0] + ',
          ' + theStrings [1] )
```

The syntax of coercion is clearly very un-Java like. For various reasons which need not concern us here, Java-style casting is not allowed in Groovy.

boring and tedious is generally the view Groovy programmers have about Java

You are probably asking the question: "Why use arrays if we can use lists directly?"

"Exactly." HW responds and represents itself thus:

```
def theStrings = [ 'hello' , 'world' ]
println ( theStrings [0] + ',
          ' + theStrings [1] )
```

Indexing into a list is supported by Groovy ? lists are sequences like arrays so indexing makes a great deal of sense. Using lists rather than arrays is generally considered more Groovy but the ability to have arrays is required for using some Java APIs.

Repeating the groove

Given that Groovy has lists, it must provide facilities for iteration enabling us to parametrise over the length of the list. Here is HW representing itself using the basic for loop in Groovy:

```
def theStrings = [ 'hello' , ', ' , 'world' ]
for ( i in 0..< theStrings.size() ) {
  print (theStrings[i]) }
println ( )
```

The expression **0..< theString.size()** is the sequence of integers starting with 0 and ending with 1 less than the upper limit. Classic idiom for iterating over zero-origin sequences. Of course, this sort of indexed looping is, quite rightly, frowned upon these days as being too low level for the task of iterating through all the element of a collection. Instead, for doing such an iteration, we should use a foreach construct:

```
def theStrings = [ 'hello' , ', ' , 'world ' ]
for ( item in theStrings ) { print ( item ) }
println ( )
```

Actually these two loop constructs are no different in language terms, they are both **foreach** loops. The first is actually *for each i in the range (0, theStrings.size ( ))*. So there is only the one type of **for** loop in Groovy – Groovy does not support the **for (int i; i < theStrings.size() ; ++ i)** variety of loop that Java does.

Well, not yet anyway, it may soon. This is all, obviously, a bit different to the way Java does things, but this is just more Groovy – dynamic rather than static, operations on data structures as a whole rather than detailed knowledge of the underlying structure, declarative rather than imperative.

Of course, we can use the Java-style **foreach** loop with its static typing approach:

```
def theStrings = [ 'hello' , ', ' , 'world' ]
for ( String item:theStrings ) {
      print ( item ) }
println ()
```

but it is considered boring and tedious by Groovy programmers. Actually boring and tedious is generally the view Groovy programmers have about Java. Groovy programmer do know and use Java, much Groovy programming is about using classes and objects from the Java Platform

after all, but they only program in Java when necessary – mostly in the same way that C++ programmers use assembler.

Closing in on closures

Actually most Groovy programmers would probably think all the above was tedious and boring since it doesn't involve the programming construct that really separates Groovy and Java: closures.

Closures are Totally Groovy – despite the idea actually being old and available in many programming languages. The biggest win for closures is that they allow a much more declarative expression of algorithms. HW has to admit that the following is a representation of itself but really doesn't show why closures are just so cool:

```
def theStrings = [ 'hello' , ', ' , 'world' ]
theStrings.each { print ( it ) }
println ()
```

Here HW uses the method each applied to a list and being passed a closure to be executed for each item of the list. It is a special variable name in a closure being the implicit closure parameter.

"Aha", you say, "there is no method **each** on lists in Java."

"Indeed," replies HW, "but Groovy can add methods to standard Java classes using its groovy meta-object protocol."

In this particular instance, Groovy has added the **each** method to Java list types exactly so that this technique of iterating over lists and applying closures can be used.

You could think of the above representation of HW thus:

```
def theStrings = [ 'hello' , ', ' , 'world' ]
theStrings.each ( { print ( it ) } )
println ()
```

but people invariably remove the optional parentheses to avoid having extra syntactic clutter. Actually it is a wee bit more complicated than this but HW has not been willing to come up with the seriously over-contrived representation required. There are limits, even to ridiculousness.

Returning to it , if we want to explicitly name the closure parameter then we can, as HW shows here:

```
def theStrings = [ 'hello' , ' , ' , 'world' ]
theStrings.each { item -> print ( item ) }
println ()
```

This is probably the more usual way of working with closures.

Of course, this algorithm, however it is expressed, involves a lot of output statements. Many feel (and the argument is a good one) that minimizing the number of output statements is generally a good thing. So concatenating the strings before outputting is probably a good thing. Groovy extends the Java list types with a **join** method so we can achieve concatenation without knowing how many items in the list:

```
def theStrings = [ 'hello' , ', ' , 'world' ]
println ( theStrings.join ( ' ' ) )
```

In Groovy, closures are first class entities so we can have variables of closure type and pass closures around as parameters. Here HW presents a fairly gratuitous representation of itself highlighting initialisation of an object that has a closure member and then execution of that closure via the class data member:

```
class ClosureWrapper
{
  @Property Closure action
  def execute ( ) { action ( ) }
}
new ClosureWrapper (
   action : { println 'hello, world' }.execute()
```

The initialization is interesting here. The class declares a data member and a method. The data member is declared to be a property – the member will be a private member and accessors will be automatically generated to fulfil the requirements that the class be a 'bean'. No constructor is declared. On initialising the new **ClosureWrapper** object we provide a constructor parameter. From a Java mindset this is clearly an error that the compiler will detect. Distinctly not Groovy. Groovy is quite happy with this code since it allows a map to be used to initialise all properties of a new instance. In this case the parameter is a single value map: the colon separates the two components of the map value, action is the key and the closure is the value associated with that key. The key must be the name of a property of the class of course! This technique for initialization of properties using a map leads to some seriously useful idioms.

OSification

Many people believe that scripting languages must be able to script the execution of operating system jobs – à la **bash**, **ksh**, etc. Although Groovy is a general purpose dynamic language (like Python and Ruby), it can be used for scripting and so must be able to do the job. Here HW represents itself in a way designed to show what is possible:

```
print ( [ 'echo' ,
        'hello, world' ].execute().text )
```

Applying the execute method to a list of strings creates a process object **(java.lang.Process)** on which Groovy has defined the **getText** method – two more instances of Groovy adding methods to standard Java classes. This means we can write code as though it is accessing a variable when in fact it is calling an accessor: using text as a property accesses the member variable if it exists or if it doesn't then the method **getText** is called if it exists. Metaclasses and beans at work in an extremely constructive way. Very Groovy.

Swinging groovily

All the above are console based but what about GUI based representations? Groovy supports builders, and a builder for constructing Swing/AWT interfaces is part of the standard distribution.

we can write code as though it is accessing a variable when in fact it is calling an accessor

Builders use all the facilities of a dynamic language with a meta-object protocol to simplify building hierarchical systems. Add to this the map initialization of properties construction technique and Swing programming becomes easy:

```
def frame =
 new groovy.swing.SwingBuilder().frame (
 title:'Hello',
 defaultCloseOperation :
 javax.swing.JFrame.EXIT_ON_CLOSE )
{
  label ( " hello , world " )
}
frame.pack()
frame.visible = true
```

Note that we use **frame.visible = true** where in Java you would have to say **frame.setVisible (true)**. As noted earlier, Groovy

allows properties to be accessed as though they were data members rather than having to use accessor calls.

The "wow factor" here is partly how the use of closures and hash parameter properties initialisation shorten the code but is mainly the fact that **`SwingBuilder`** does not actually have any methods such as **`frame`**, **`label`**, etc. The metaclass associated with the **`SwingBuilder`** object knows how to turn what appear to be method calls into the construction of Swing/AWT components. It does this by reflection and understanding the naming conventions rather than by explicitly mapping method call names to component type names. Very flexible. Most Groovy.

Anyone having any experience with Swing/AWT must surely agree with HW that this way of constructing Swing/AWT GUIs is just 'cool' and 'groovy'.

Listing 1

```
class helloWorld_Test
{
  static void main ( args )
  {
    def testClass = ' ' '
    class TestHelloWorld
     extends GroovyTestCase
    {
      private final expected =
       ' hello , world '
      private final shell =
       new GroovyShell ( )
      private final saveSystemOut = System.out
      private final buffer =
       new ByteArrayOutputStream ( )
      private final outputFromScript =
       new PrintStream ( buffer )
      void setUp ( ) { buffer.reset ( ) ;
       System.setOut ( outputFromScript ) }
      void tearDown ( ) {
       System.setOut ( saveSystemOut ) }
      void evaluateFile ( String filename ) {
       shell.evaluate ( new File (
       filename ) ) }
      String getOutput () {
       return buffer.toString().trim() }
      ' ' '
      new File ( '.' ).eachFile {
       s -> def script = s.name.trim()
      if ( script =~
       /^ helloWorld_ [ a ? z ].*\.groovy$ / )
      {
        def methodName =
         ' test ' + script.substring
         ( script.indexOf ( ' _ ' ) ,
           script.lastIndexOf ( ' . ' ) )
           testClass +=
            " void $ - methodName " ( ) {
        evaluateFile ( ' $ - script }' ) ;
        assertEquals ( output , expected )
         "\ n "
      }
    }
    testClass += ' " ;
    return TestHelloWorld '
    def shell = new GroovyShell ( )
    shell.runTest ( shell.evaluate (
     testClass ) )
  }
}
```

1, 2, 3, Testing

Interestingly HW is actually a bit of an advocate of unit testing though it isn't too sure about test-driven development but that may be because HW can be represented in so many ways without having to develop. This is one of the wonders of HW.

So we are agreed that all the representations of HW need to be tested to ensure semantic correctness. Clearly we must separate the console-based representations from the GUI-based representations. For the console-based representations we can construct a program that tests all of the programs in a given directory (see Listing 1).

Wow, this is getting seriously serious.

This program constructs a program (comprising a **`class`** definition and a **`return`** statement) as a string, then evaluates the string (using a **`GroovyShell`**) which compiles the program and loads it into the running JVM. The result of this compilation and load is a reference to the class object which is then used (by a **`GroovyShell`**) to execute the program as a JUnit test. The **`GroovyShell`** is just so Groovy.

Actually this is very true, the **`GroovyShell`** is arguably the single most important class in the Groovy system since it is responsible for executing any and all Groovy programs.

Reactions to all this range from "Wow, just how cool is that." to "OK, isn't that just obviously the right thing to do." depending on whether you are new to dynamic programming languages or an old hand.

As you have probably guessed already, triple single quotes or triple double quotes start a multi-line string, i.e. one that can include end-of-lines as data. The example here is used to present the literal that is the fixed text part of the generated program.

Listing 2

```
class build
{
  private final ant = new AntBuilder ( )
  def init ( )
  {
    ant.taskdef ( name : ' groovy ' ,
     classname :
      ' org.codehaus.groovy.ant.Groovy '
     )
  }
  def test ( )
  {
    init ( )
    ant.groovy (
     src : ' helloWorld_Test.groovy ' )
  }
  def clean ( )
  {
    ant.delete ( quiet : ' true ' )
    {
      ant.fileset ( dir : ' . ' ,
       includes : ' *~,*. class ' ,
       defaultexcludes : ' no ' )
    }
  }
  static main ( args )
  {
    def builder = new build ( )
    if ( args . length == 0 ) {
     builder.test ( ) }
    else { args.each {
     target -> builder.invokeMethod (
     target, null )
    } }
  }
}
```

The generated class **TestHelloWorld** is a subclass of **GroovyTestClass** so that it is treated as a **(JUnit-based)** test class. The variables **(expected**, **shell**, **saveSystemOut**, **buffer**, **outputFromScript)** and methods **(setUp**, **tearDown**, **getOutput)** implement the infrastructure for capturing the standard output of executing a script via a GroovyShell so that the test program can capture the output from executing the various representations of HW - which is achieved using the **evaluateFile** method. All the test methods are written in the closure used in the iteration over all the names of the files in the directory.

As noted earlier, Groovy allows extension of standard Java classes. Here we see use of the **eachFile** method that has been added to **java.io.File** to support using a closure as part of a closure style iteration.

In the closure itself, we are using regular expressions (things that look like regular expressions initiated with a **/** and terminated with a **/**) and the match operator (**=~**) to make decisions about writing the necessary test methods for any given representation of HW in the current directory. The name convention being assumed and applied is that an HW representation will be in a file starting '**helloWorld_**', finishing '**.groovy**' and comprising only lower case letters.

You might have noticed the use of **$ {methodName}** in a string. These are substitution markers and the string is not a standard Java string as the examples have been so far but is a **GString** , i.e. a string in which substitution markers are substituted.

Having said all this, it remains that the single most important thing about this example is the idiom of iteration using a closure.

Building groovily

In the section called 'Swinging groovily', HW represented itself as a Swing application using builder technology. Builders are so Groovy that HW thought it appropriate to show how Ant (the standard Java system build tool) can be scripted in Groovy avoiding all the need to have any XML files at all. What a lovely lack of angle brackets.

Here is the Groovy script that handles all the building for the Groovy representation of HW used in this article:

Build targets are represented as methods which then use builder technology (including property initialisation with maps and nesting using closures) to script calls to the tasks in the Ant library.

Anyone at all familiar with using Ant will very quickly see how this is working and appreciate the benefits of using a scripting language rather than XML to define the build.

It is possibly interesting to note the **if** statement in the **main** method of the class. The **if** branch implements the test target being the default target and the **else** branch, which is a closure being applied in order to all the targets listed on the command line, shows how the metaclass system of

Groovy can be used directly to call methods using the string representation of the method name. This is the Groovy metaclass system being used directly. Very dynamic. Very Groovy.

Interestingly, the success of builders in Groovy has caused Ruby to really take builders on board.

Getting in the Groove

Languages like C gave way to C++ because people could express things more easily. Java supplanted C++ for many because of ease of expression and portability. Python and Ruby allow much more dynamism, as does Groovy. No one language is better than another per se. Having said that, Java people trying Groovy rarely look back at Java as better.

If this article has piqued your interest, and HW hopes it has, the Groovy website home page is **http://groovy.codehaus.org** . Remember though the Groovy documentation is not particularly good yet even though the language itself is excellent. Python and Ruby have had 10 years or more work on their documentation, Groovy is still being built - 2006-07 should see the first formal release of the system itself. Work then starts on making the documentation of good quality. Also work will begin in earnest on the TCK ready to progress JSR-241 which will lead to Groovy becoming an integral part of the standard Sun Java distribution.

A final note: HW has not been able get a look in on Groovy documentation as yet due to the Wallace/Grommit/cheese fixation of the original Groovy authors. See the Groovy website for more on this problem.

The other final note: HW apologies for becoming a mop-head but meta-object protocols just rock its world. ■

References

1. Kernighan, B. and Ritchie, D. (1978), *The C Programming Language*, Prentice-Hall. Page 8.

Article writing competition 2006

Every year the ACCU awards a number of prizes for articles published in our magazines. This is a simple way to recognise the high quality of articles we receive, and to thank people for taking the effort to write for us. Both C Vu and Overload rely on the articles contributed from ACCU members.

At this year's conference, the ACCU committee reviewed all articles published in the last volume of C Vu and Overload, selected the front runners, and voted for the best in each category. The results were announced at the AGM:

- **Best C Vu article**

  Code Monkeys (Professionalism in Programming)
  by Pete Goodliffe, in C Vu 17.1/17.2
- **Best overload article**

  The Curate's Wobbly Desk
  by Phil Bass, in Overload 70
- **Best article by a new writer**

  Sheep Farming For Software Development Managers
  by Pippa Hennessy, in Overload 66

Kudos and untold fame to go these winners! But thanks to *all* writers who contributed in 2005/6. And remember: write an article; next year it could be you!

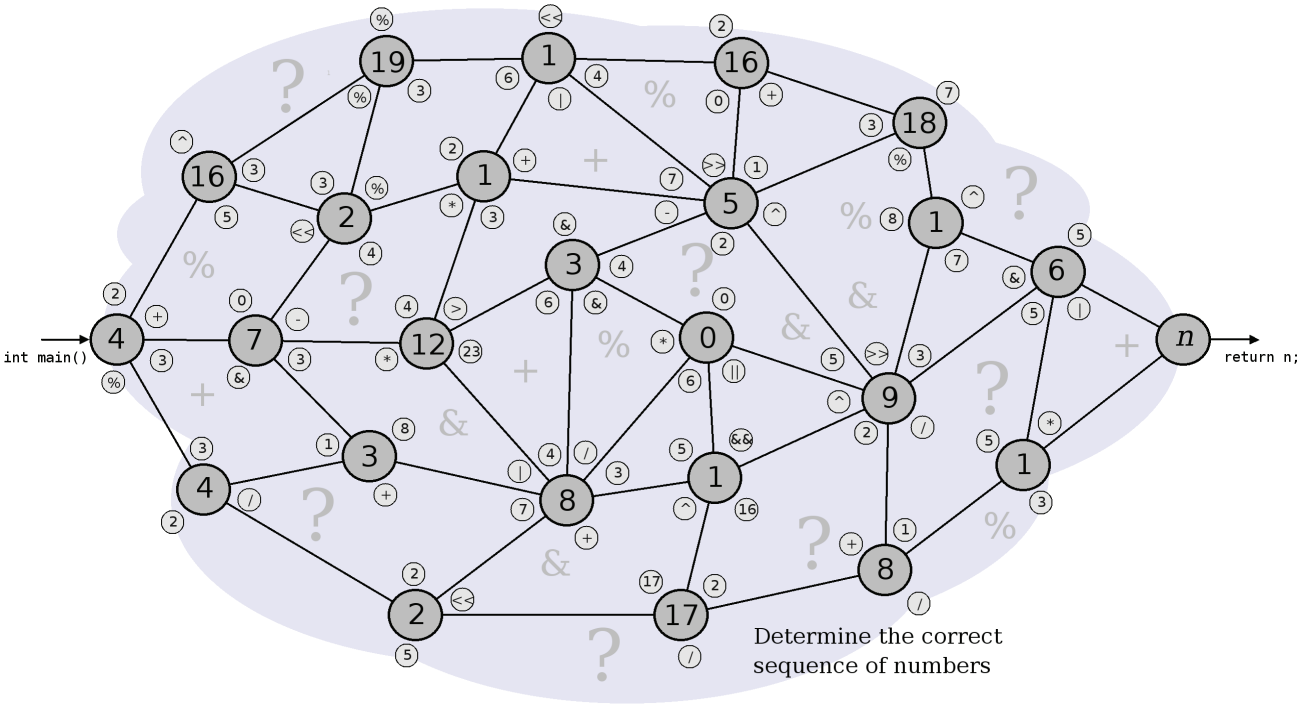
Mental Gymnastics

Pete Goodliffe stretches our little grey cells.

I find that my kids teach me more about the world – about how we learn, think, and act – than I could ever have expected. And it's interesting to see how some programmers still act like a three-year-old when it comes to it.

This issue I want to depart from the norm, and engage the inner child. Who knows, perhaps it'll do you some good! My daughter Alice has become engrossed with puzzle books of late, and to share the joy with you, here are a number of programming puzzles to stretch your little grey cells.

When you have a solution to these problems, mail it to me. The first (or most amusing) will have their names printed in the next C Vu. ■



Spot the difference

There are a number of differences between these two code snippets. Can you find them *all*?

```
#include <iostream>
#include <algorithm>




template<typename Iterator>
void bubble_sort(Iterator first, Iterator last)
{
    for (Iterator i = first; i != last; ++i)
        for (Iterator j = first; j < i; ++j)
            if (*i < *j)
                std::iter_swap(i, j);
}
```

```
#include <stdio.h>

void swap(int *x, int *y)
{
    int tmp = *x;
    *x      = *y;
    *y      = tmp;
}


void bubble_sort(int array[], size_t size)
{
    for (int i = 0; i < size-1; ++i)
        for (int j = 0; j < i; ++j)
            if (array[i] < array[j])
                swap(&array[i], &array[j]);
}
```

Pete's book, Code Craft, is out in August 2006.

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

Value for Money

Alan Lenton queries how cost-effective some projects are.

I was looking at a story about the UK's Project Semaphore the other day and thinking about value for money.

Let me explain. The project is a pilot scheme for the British Government's much vaunted eBorders project, which, it is claimed, will enable us to spot baddies trying to get into the country to blow us all up. It won't, of course, catch disaffected locals, like the ones who bombed the London transport system!

The pilot project has just finished, and the politicians were boasting about how successful it was - six million passengers screened and 140 arrests. The report was fairly straightforward, talking about how the main eBorders program would now go forward and tenders were out for the contract. The main thrust of the article was about whether in the light of all the other major IT projects going on, there would be enough skilled developers.

Much as I like the financial implications for developers like myself, that wasn't what caught my attention. It was the 140 baddies caught that I wondered about. Sadly, I couldn't find out any information about what they were arrested for, but I guess it would be reasonable to assume that had any of them been known terrorists the politicians would have been trumpeting the success of the scheme across the national newspapers. So my betting was that quite a number of the arrests were for relatively minor infringements of the immigration rules.

I was intrigued enough to search for material from when the scheme was started to discover that it was budgeted at £15 million. I couldn't find anything about cost overruns - something made me immediately suspicious! However, we can give them the benefit of the doubt and assume it did only cost £15 million.

So, £15 million for 140 arrests. I make that just under £110,000 per arrest.

That's an awful lot of money to pay to track and arrest someone. In fact it's the sort of Return on Investment (RoI) that would result in mass sackings in most big corporations. And remember that's only the cost of apprehending the person, not prosecuting or anything like that. Significantly enough, figures don't exist for conviction rates, or what sort of sentences were given, or whether those involved were just refused entry, because their papers weren't in order.

And lest our US readers are feeling smug about the UK wasting money, let me mention that a few days after reading about Project Semaphore, I came across a piece about the US government's US-VISIT program in security guru Bruce Schneier's newsletter.

According to Schneier, since January 2004, US-VISIT has processed more than 44 million visitors. It has spotted and apprehended nearly 1,000 people with either criminal or immigration violations. The budget for that phase of US-VISIT was US$15 billion - we'll assume, as with Semaphore, that it didn't overrun on costs – again a very dubious assumption, but we will be generous.

That makes the cost of catching each baddie a cool US$15 million. Wow! Maybe we Brits are, in fact, doing really well. It's interesting to know what the government is doing with your tax pounds, dollars,or other currency, isn't it?

Oh, and by the way have you noticed that politicians never apply value for money criteria to their pet projects, only to ones they want to close down? ■

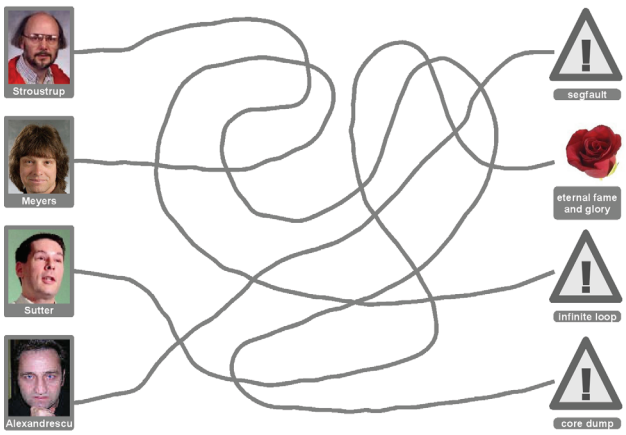
ALAN LENTON

Alan Lenton is a long-standing ACCU contributor. He can be reached at alan@ibgames.com

Mental Gymnastics (continued)

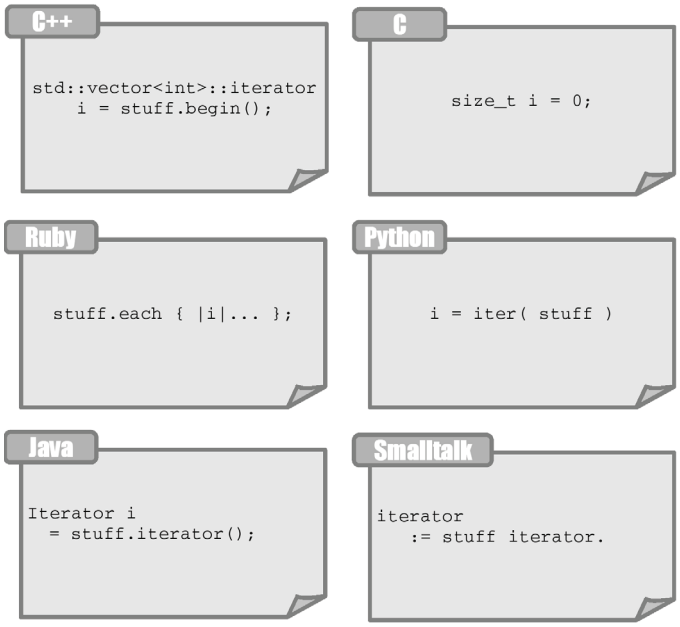
Follow the leader

Which industry luminary's code will achieve eternal fame and glory, and whose code will meet a sticky fate at the hands of a merciless debugger?



Odd one out

Which piece of code is the odd one out, and why?



Trees, roots and leaves #2

Örjan Westin keeps track of the 'adopted' children.

This is the second article of three in which I create a generic, re-usable tree. In the first part, I defined the requirements and made a start at the implementation, putting the value and parent members in place. In this part, we'll see how the management of children brings its own, special headaches, and finish with something that is quite complete and usable.

If it weren't for those pesky kids

With the requirement that I should be able to take a subtree out and treat it as a separate tree, I must use dynamic containment, as a list of pointers, rather than direct aggregation to hold the children. I choose to use an ordinary **std::vector** to keep track of the pointers, and I'll give access to many of its functions through simple forwarding functions (Listing 1).

Listing 1

```
template <typename T >
class tree_node
{
public:
  typedef tree_node<T> node_type;
  typedef tree_node<T>& reference;
  typedef const tree_node<T>& const_reference;
  typedef tree_node<T>* pointer;
  typedef std::vector<pointer> child_holder;
  typedef child_holder::iterator iterator;
  typedef const T& const_value_reference;
  ...
  iterator begin()
  {
    return children_.begin();
  }
  reference front()
  {
    return *children_.front();
  }
  ...
private:
  child_holder children_;
  ...
```

However, this is the first obvious problem with my new container. Because I have a vector of pointers, I have to de-reference what I get from the call to **front()**. So what do you get if you de-reference the iterator above? Right. A pointer to a **tree_node**. That it is a **tree_node** and not the template type isn't a problem – as I said earlier this is a feature, not an implementation detail – but I would ordinarily expect to get a reference out rather than a pointer. For the time being, I will ignore this, as it does not fundamentally affect how the class works, but just makes for uglier code when using it. It's an issue I will return to in the last part, though.

While we all know you should never break encapsulation casually, this class illustrates why. Were I to reveal the storage mechanism of the children, it would allow the uncontrolled adding and deletion of children, something that could, no, make that would, lead to memory leaks and access violations. We must keep track of those relationships.

Liaisons dangereuse

A node serving as branch or root needs to be able to take in new children and get rid of old. Since we're talking about a parent-child relationship, I decided to call these adopt and disown, to get a bit of drama. At first look, they are simple: the first adds a child to an internal list of children while the second removes it. But it is a bit more complicated than that – relationships between parents and children are dangerous and volatile things, as anyone with a teenager at home will tell you.

By necessity, there is a two-way relationship between parent and child – they both know of each other. A node can be made a parent by telling it to adopt another node. This not only adds the new child to the list of children the parent holds, but also tells the child about this new relationship by calling a private function, reparent, on the child. (Listing 2).

I do little more than the barest minimum of safety checking here, just enough to make sure I don't put a NULL pointer in. This is bad enough to warrant an exception. Of course, in the documentation I could just say that the pointer should be valid and bring out the "undefined behaviour" warning flag if it isn't, but since I need to check whether I already have it anyway (as a pointer that occurs twice in the list would be deleted twice on destruction), I might as well give a hint. Should someone try to adopt the same child twice, that's fine with me. At the end of the call, the child is adopted, so conceptually, the function has succeeded.

But what happens if the child already has another parent, if we are in fact moving it from one parent to another? Contrary to literary tradition, it must inform its old parent that it's leaving. This is done internally by the child through a call to **elope()**, which in turn calls the private function

Listing 2

```
  iterator adopt(pointer child)
    {
    if (NULL == child)
      throw std::invalid_argument(
       "Cannot adopt NULL pointers");
    iterator i = find_child(child);
    if (i == children_.end())
    {
      children_.push_back(child);
      i = --children_.end();
    }
    child->reparent(this);
    return i;
   }
private:
  iterator find_child(const pointer child)
  {
    return std::find(children_.begin(),
     children_.end(), child);
  }
  void reparent(pointer parent)
  {
    if (parent == parent_)
      return;
    elope();
    parent_ = parent;
  }
```

ÖRJAN WESTIN

Since 1993, Örjan has worked as a developer and consultant (apart from a return to Mid Sweden University in 1995 where he created and taught a course in "GUI design and Windows Programming"), mainly using C++. He is currently working for Unilog Ltd as a Senior Consultant and can be contacted at orjan.westin@unilog.co.uk

Listing 3

```
  pointer elope()
  {
    if (NULL != parent_)
    {
      parent_->eloping(this);
      parent_ = NULL;
    }
    return this;
  }
  pointer disown(iterator child)
  {
    if ((child != children_.end()) &&
       (this == (*child)->parent()))
       return (*child)->elope();
    throw std::invalid_argument("Cannot find
       child to disown");
  }
private:
  ...
  void eloping(pointer child)
  {
    iterator ch = find_child(child);
    if (ch != children_.end())
      children_.erase(ch);
  }
```

eloping(). This lets the old parent remove the child from its list of dependants and get on with its life, without having to worry about the mysterious disappearance of its offspring. Here, I'm using a return value to provide a handle to the object. If **elope()** is called through an iterator, I most definitely want to have its pointer, since I will have lost it forever if I don't. (Listing 3)

The scenario opposite from adoption, when I take a child from its parent, works the same way behind the scenes. You call the public function **disown** on the parent, telling it which child to give up on. This, in turn, causes the child to shout, "You can't disown me, I'm eloping!" like the ungrateful brat it is.

Admittedly, this seems a bit unnecessary. Couldn't you just inform the child it's been abandoned and remove it from the list? Yes, you could, but this has the advantage that there is only one place where this abandoning of children takes place, in **eloping**. And it looks more dramatic when I write about it. Note that **disown** doesn't erase the child, only its relationship with the parent, which is why I return a pointer to the newly independent child.

Clearance

However, we do need that function – abandon – or something like it. Why? Well, a parent can do worse things to its children than disown them. It can also get rid of them completely. The destructor of a node must inform its parent, if any, that it is no longer a valid child by calling **elope()**. This is right and proper, but inefficient and unnecessary if it is the parent that is destroying it.

Since I figure I should provide functions to erase children anyway, I wrote Listing 4.

This works fine, as far as it goes. But what about when the client wants to erase a range, or even all children? What about the destructor, which needs

Listing 4

```
  iterator erase(iterator it)
{
  pointer p = *it;
  iterator i = children_.erase(it);
  p->parent_ = NULL;                  // abandon
  delete p;                           // delete
  return i;
}
```

Listing 5

```
  ~tree_node()
  {
    delete value_;
    clear();
  }
  iterator erase(iterator it)
  {
    pointer p = *it;
    iterator next = children_.erase(it);
    p->erase_silent();
    return next;
  }
  iterator erase(iterator first, iterator last)
  {
    child_holder backup(first, last);
    iterator next = children_.erase(
     first, last);
    std::for_each(backup.begin(), backup.end(),
     std::mem_fun(&node_type::erase_silent));
    return next;
  }
  void clear()
  {
    erase(begin(), end());
  }
  ...
private:
  bool erase_silent()
  {
    parent_ = NULL;
    delete this;
    return true;
  }
```

to do just that? While I could call the above function manually for each one, this would be error-prone and inefficient. A better idea is to use separate function to do the actual deletion, giving the clean-up code in Listing 5.

There are some points worthy of notice in this code. Again, I only have one place in which the action happens – the **erase_silent** function – just like in the eloping and disowning discussed earlier. This is a sound design principle, both to avoid code duplication, and therefore maintenance problems, and to increase clarity and brevity. Remember that sword – there should only be one purpose, whether it's a class, function or variable. Conversely, any purpose should only be expressed once, and all other instances – of functions, classes and variables – should be expressed in terms of that one.

I am also careful not to delete things until I have managed to take them out of my list of children. Had I done it the other way around, I might erase children that are not mine, without letting the other parent know this had happened (since the parent pointer is reset before deletion) before failing to get them out of my own list of children. The other parent would then have pointers to children that didn't exit, causing all sorts of unpleasant surprises.

This way, **std::vector::erase** will protest before I do anything I might regret. Admittedly, this means I'm relying on the **erase** function to raise a fuss, and since the standard says that erasing an iterator that is not valid causes undefined behaviour, I can't be sure what happens. I feel it's sufficient to emulate this, though, and if nothing else it will be consistent with how your implementation of the standard library does things.

In the range version of the erase function, I take a copy of the range first. This might fail if the range is invalid, as it falls under undefined behaviour, but even if it is valid it might not be mine. If it isn't, the call to **children_.erase** should protest, and I won't have deleted anything.

If it works, I still have a record of the objects I need to delete, even though they've been erased from the list of children.

Making the function **erase_silent** appear as a functor, or function object, by putting it in the **std::mem_fun** adapter also saves me the chore of writing a loop, since I can simply use the one provided in the algorithm library. Many programmers I know only use the containers from the STL, and consider things like algorithms and function adapters to be strange and esoteric, but really it is quite simple. The **std::mem_fun** just makes a member function look like a functor. Unfortunately, any function passed to the standard function adapters has to return a value. They shouldn't have to, but few compilers and libraries accept this construct yet, which is why **erase_silent** always returns true.

Copy right

That's the erasing part done, but I also promised a function to copy the contents of a **tree_node**. For this, I'll use a similar approach, and add an adoption function that creates a new child from a value instead of taking an existing one. This is a bit more complicated, so I will need some helper functions, both private and static (see Listing 6).

First of all, in order to maximise exception safety I use the "create a temporary and swap" idiom. If the creation of the temporary works, I can swap safely. The same goes for the value, which is a simple pointer copy operation and guaranteed not to throw. Of course, I can't use the vector's swap function, since the parent needs to be set, but the idiom is the same. The adopt function is safe, since it's only doing pointer assignment and a push_back on a vector which is guaranteed to have the capacity.

Unlike the erasing, I can't easily use **std::mem_fun** on a member function to get a functor. The reason for this is that I need to bind one of the parameters – the destination – since **std::for_each** only works on unary (single-parameter) functors. For **std::bind1st** to work, it needs the functor to have some type definitions you get for free from **std::binary_function**, which also explains why it quite pointlessly returns a boolean – it is required to have a return type.

The functor **AdoptCopy** takes pointers to what would be the left and right sides of an assignment operator, i.e. the destination and source, respectively. Its purpose is simply to call a static function that creates a copy of the source and ask the destination to adopt this copy. In both the **copy_children** and **create_copy** functions, it is used in the **std::for_each** algorithm, with the destination bound to it.

Note that this is a recursive setup. **AdoptCopy** calls **create_copy** which calls **AdoptCopy**... There have been many arguments about the benefits and/or horrors of recursion over the years, and I have to say that I am usually not fond of using it. In this case, however, I think it's the best solution, as it makes a much more elegant solution than a non-recursive one. Trust me, I tried.

On a final note, the snippet in listing 6 also features the only **try-catch** block in the code. That's because this is the only potential memory leak I can control directly. Should the copying of children fail – because of lack of memory, or an exception thrown in the value constructor – I abandon the whole thing, delete the node I was assigning children to and rethrow the exception, making sure I do not leave any unadopted children lying around.

To finish the **tree_node**, I'll add some call-through functions for the list of children – like **back, end, empty** and **size** – and put in **const** versions of existing functions. With that, this class is complete. It is as exception-safe as **std::vector** and the value type allows, it's without memory leaks, it works, and it's useful. Wonderful. But before I dislocate my shoulder trying to pat myself on the back, I must remember that it's not quite finished.

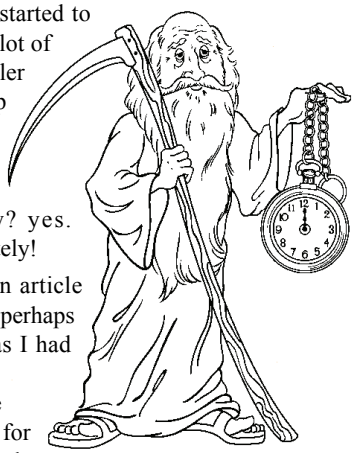
In the next, and final instalment, we'll take a look at what is required of a container, and do what we can to fulfil those requirements. ■

Listing 6

```
  void adopt(const_value_reference child)
  {
    adopt(create_value_node(&child));
  }

  void copy_contents(const_reference node)
  {
    node_type temp();
    temp.copy_children(node);
    if (node.has_value())
      temp.assign_value(node.value());
    // Make room
    clear();
    if (temp.size() > children_.capacity())
       children_.reserve(temp.size());
    while (temp.size())
      adopt(*temp.begin());
    value_ = temp.value_;
   temp.value_ = NULL;
 }
  ...

private:
  void copy_children(const_reference node)
  {
    clear();
    std::for_each(node.begin(), node.end(),
     std::bind1st(AdoptCopy (), this));
  }
  static pointer create_value_node(
   const_value_pointer child)
  {
    if (NULL != child)
      return new node_type(*child);
    return new node_type();
  }

  static pointer create_copy(
   const_pointer pnode)
  {
    pointer p = NULL;
    try
    {
      p = create_value_node(pnode->value_);
      std::for_each(pnode->begin(),
       pnode->end(),
       std::bind1st(AdoptCopy(), p));
    }
    catch (...)
      {
      delete p;
      throw;
    }
    return p;
  }

  struct AdoptCopy :
     public std::binary_function<pointer,
     const_pointer, bool>
  {
    bool operator()(pointer lhs,
                    const_pointer rhs) const
    {
      lhs->adopt(create_copy(rhs));
      return true;
    }
  };
```

Blasting from the Past

Paul Johnson asks when everything started to get so complex.



Way back in issue 17.1, I started to look back – with quite a lot of fondness – at a far simpler time. A time when you could fire up a computer in a couple of seconds and through the built in BASIC interpreter, could construct some pretty neat bits of code. Slow? yes. Uninspiring? possibly. Fun? definitely!

In 17.2, I followed this up with an article which proposed the argument that perhaps things haven't changed as much as I had thought. Sure, it was now a case of hunt the library and refer to online manuals, but the basic mechanism for sound is just the same – the only difference now is that instead of the software working on (say) a BBC B, it would work on a range of hardware architectures – the only caveat being that they ran the same operating system. It is even possible to compile code against an older library and have the results be the same.

.NET and Java just makes things even easier in so far that as long as the system has a compliant runtime environment, code doesn't even need to be recompiled.

I'm sure we're all aware of this, being coders and programmers all.

It's time to throw a spanner in the works – a spanner in the form of the processor. For those who still have RISC OS machines, they will be all too aware of this problem (especially those who worked for Acorn before its demise). The ARM processor (originally) though it was 32 bit processor, it was actually a 26 bit addressed chip with the remaining bits being used as registers.

This was never really a problem, until the number of these 26/32 chips began to run out. The likes of the ARM9 and XScale processors (and newer StrongARM SA1110 chips) are 32 bit only. This caused quite a lot of problems for RISC OS software, especially where a company who had produced software had vanished for whatever reason. Code would die on the newer processors. Not a problem if you had the source and a 32 bit compiler, but a pain in the backside if you didn't.

For those on PCs, this same problem is now coming to the surface with the advent of 64 bit processors and the relatively small difference in price. Not all code written for 32 bit machines can be just compiled for 64 bit architecture.

Take a simple example of the data model of the processor (table 1). Not only is there a difference (as you'd expect) between the 32 bit and 64 processors, but even with the different types of 64 processor, there is a difference in the non-pointer data types. When the width of one or more of the C data types changes from the different models, applications may be affected. The effects are in two main catagories, as shown in Table 1:

- Size of data objects. Compilers align data types on a natural boundary (32 bit data types are aligned to 32 bit boundaries). Upshot: the size of data objects (such as a structure or union) will be different on the architecture.
- Size of fundamental data types. You can no longer use the common assumptions about the relationships between the fundamental data types.
  `sizeof (int) = sizeof(long) = sizeof(pointer)`
  for the 32 bit data model, but not for the others.

Table 1

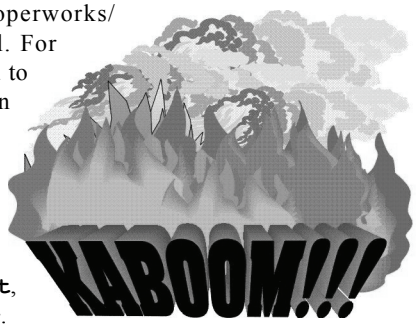
	ILP32	LP64	LLP64	ILP64
`char`	8	8	8	8
`short`	16	16	16	16
`int`	32	32	32	64
`long`	32	64	64	64
`long long`	64	64	64	64
`pointer`	32	64	64	64

Of course, the compiler will do the aligning itself (padding). A practical demonstration of this can be seen here in table 2:

Table 2

structure member	size on 32 bit	size on 64 bit
`struct test {`		
`int i1;`	32 bits	32 bits, 32 bits filler
`double d;`	64 bits	64 bits
`int i2;`	32 bits	32 bits, 32 bits filler
`long l;`	32 bits	64 bits
`};`	`sizeof(struct)` = 20 bytes	`sizeof(struct)` = 32 bytes

I'm not going to go into this in detail, but those nice chaps at IBM have a fair bit of documentation available at http://www-128.ibm.com/developerworks/linux/library/l-port64.html. For Win32 users, I'd advise you to have a look as the next version (Vista) is supposed to be 64 bit only, though we'll have to wait and see if that is the case!



If you're wondering what ILP et al stands for, `I = int`, `L = long`, `P = pointer`.

Why am I on about with this and more over, what has it got to do with blasting from the past?

You will have all of realised by now that I love my old 8 bit days – and going by the hit counters on the likes of worldofspectrum.org (and other 8 bit emulation sites), so do quite a lot of other people. Emulation is all well and good, but for the life of me, I don't want to get out my 6502 and Z80A manuals to program in machine code again. I'm way too used now to my gcc tool chain plus with the likes of Oric and Spectrum, there wasn't a built in assembler, so everything was the raw hex and one mistake...

Thankfully, those days are well gone now and there are two pieces of software I use to compile my Spectrum and Oric code

PAUL JOHNSON

Paul Johnson works at the University of Salford where he sometimes teaches, but mostly fixes computers and attempts to keep members of staff happy. Contact Paul at paul@all-the-johnsons.co.uk

with – z88dk and OSDK. Actually, that should read "used to compile … with" as neither of them will build on a 64 bit system with a modern compiler (such as gcc 4.1)

Closer examination of z88dk shows it's really not a big thing to fix the code to make it neutral, but OSDK – that's a different matter. The original code was written (from what I can make out) in the original K&R C

Now, if you've been programming since Stroustrup was knee high to a grasshopper, then K&R will hold no fear for you. However, if, like me, you've only been programming for about 8 to 10 years, K&R, though understandable (after all, it is still C when all is said and done) is baffling and what's worse, is a pain to follow.

For example, below is completely acceptable C then and now

```
char foo(a, b, c);
int a, long b, char c
{
/* */
}
```

It's not hard to understand either. However, when you have declarations at the start of the source file (and the source file is not a trivial one) of the style

```
DECL char foo(int a, int b, char c);
```

and all you then have at the function:

```
char foo(a, b, c)
char c
{
/* */
}
```

you will find some who will just switch off.

Now, assuming that the code has been re-written to be more C99 (or C89) compliant, there then comes the problem of making the code to be processor neutral after it's compiled. That's not so hard. The hard part though is ensuring that the code generated by the compiler hasn't been broken by the port and it still runs on your favourite emulator as native code!

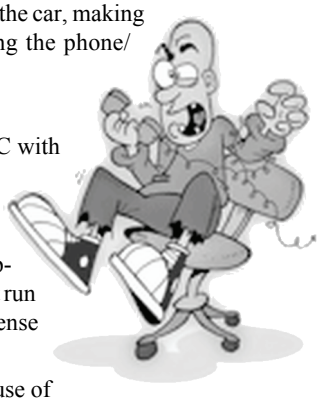
In theory it should run – and for a large degree of the time, it does, but for some reason, graphics and sound tends to fail. Text, the (emulated) save and load and even switching (on the Oric) to HIRES seem fine, but try to get it to use the ZAP or SHOOT sound effects and it's good night sweetheart time.

It appears that there is another bogey in the system which wasn't apparent. Predefined types used within the source (remember, everything is 8 bit on the target machine) and some very bizarre **`#define`**s which just break on the newer 64 bit systems. At this point, you start to worry not so much

.NET and Java just makes things even easier in so far that as long as the system has a compliant runtime environment, code doesn't even need to be recompiled

The moral of this piece is to document. And make sure the documentation is good

about the sanity of the author, more your own sanity; why on earth are you fixing someone else's code to compile code on a target machine long since gone! Could this time be not better spent on sorting out that odd knocking noise at the front of the car, making another coffee or just plain answering the phone/ walking the dog?



What a silly question!

I mean, what is your 2.8GHz 64 bit PC with more memory in it that the original creator of the target machine could be dreamed of and a sound system which in would embarrass even the most up-to-date cinema of 1985, for if you can't run your old 8 bit software on it? Talk sense man!

The problem now though is that because of a macro, which calls another macro which relies on some predefined 8 bit integer type which in turn is part of a structure assumed to be a particular size which isn't because of padding now apparent on 64 bit systems, sound and video is effectively shot on the generated code.

There is a simple fix – tell the compiler to compile as if it was a 32 bit compiler (in other words, use **`-m32`** on **`gcc`**). Only problem there is that this now means that if you're using Intel C++ (or another non **`gcc`** derived commercial compiler on a native 64 bit system), you need to change the flag – or worse, that ability to re-target is not there.

It would be simpler all round to spend a week fixing the problem so that future users don't have the problems currently being encountered.

The key to sorting the problem though isn't in the code so much. The problem is actually in what isn't there!

As with so many packages, documentation in both the source and extra to the source is virtually nil. By spending a good couple of hours (depending on how large the code base is) in documenting what does what, connects to what, depends on what and other such trivial things, the fixing of the code should be a great deal simpler.

The moral of this piece is to document. And make sure the documentation is good. It saves time in the long run and better still, means loonies like me can enjoy writing code for our pet 8 bit machines on our whizz bang state of the art boxes.

Of course, the fixing of z88dk and OSDK are not earth shattering, however, larger pieces of industrial code are and the following pieces demonstrate greatly differing strategies on how people deal with maintaining legacy code. ■

Building on a Legacy

Anthony Williams provides some tips for inherited code.

The term "Legacy Code" has been imbued with meaning by the software development community, and evokes images of masses of tangled, hard-to-change code, with bad variable names, 1000 line functions, misleading comments, and unfathomable dependencies.

Taken literally, it means code "handed down by an ancestor or predecessor" [1], which doesn't sound so scary, and it needn't be, provided suitable care is taken. I've found the following techniques to be useful when I've had legacy code to work with, and I hope they can be useful for you too.

What, no documentation?

Many developers bemoan the lack of documentation when presented with some legacy code to maintain, but this often belies their real concerns – they find the code hard to understand, and hard to change, without introducing bugs.

Michael Feathers asserts [2] that the best way to maintain legacy code is to get it under test, so you don't have to worry about breaking it as you make changes, and I'm inclined to agree. If you haven't got a copy of Michael's book, I really recommend you buy one.

Bearing in mind the problems of maintaining hard-to-understand code, it's well worth taking the time to refactor, to make it clearer. I've written about what I consider makes for maintainable code before [3], and you should bear this in mind when making changes. However, it is important not to get carried away and try and rewrite everything – that way lies stress, as you're not making any progress in the mean time.

Baby steps

The key to making any changes to legacy code, whether in order to add features or to get it under test, is to make a series of small changes, rather than one big one, and then verify that everything works after each small change. Obviously, the best way to verify that nothing has broken is with a set of automated tests, which is Catch-22 if you need to make the changes in order to add the tests – in this case, you have to rely on making the smallest changes you can in order to add the tests, and manual testing to verify the behaviour.

Stay focused

When faced with a big ball of mud, it's tempting to dive in and refactor like crazy, tidying up the code all over the place. This is not a good idea, since you're not adding new functionality whilst you're doing this, and you are modifying this code for a reason – you've got a bug to fix, or a new feature to add.

Instead, the way to deal with the mess is to focus on the area that needs to be changed. If you're fixing a bug, hunt it down ruthlessly, then add tests to that specific area, so that (a) you'll know when the bug is fixed, since you've got a test case that traps it, and (b) you'll know that you haven't broken any of the desired behaviour in that area. Once the tests are in place, tidy up this corner of the code-base – split that 1000 line function, by extracting small, self-contained functions with good names, and clear responsibilities; rename a few variables; group related data into structures and classes.

If you're adding a new feature, work in a similar way – find the parts of the code that need to change in order to add the new feature, add tests to the existing code, write tests for the new feature, and add the new code. Again, once the tests are in place, tidy up the affected areas of the code-base. Sometimes it is best to do this before adding the new code, in order to make it just that little bit easier to add the new feature, and sometimes it is a good idea to do it afterwards, to eliminate some duplication, and make it easier to change next time.

Pay back technical debt

Unmaintainable code-bases, with a lack of tests, are often said to have accrued a lot of "Technical debt". The consequence of accruing the technical debt, and not keeping the code clean and well-tested is that you have to pay "interest", in that adding new features and fixing bugs takes longer.

By adding tests, and refactoring to improve the design, as you fix bugs and add new features, you are paying back some of this debt. The code should be cleaner after your modification than before, with less duplication, and more tests. Working this way, the areas of code that change frequently will become well-covered with tests, and will gradually become better designed over time. Adding new features will therefore get easier as time goes on, rather than harder.

faced with a big ball of mud, it's tempting to dive in and refactor like crazy, tidying up the code all over the place

The areas of code you haven't had reason to change will remain just as untidy as before, but if you don't need to change them, this isn't a problem. Also, if you're fixing all the critical bugs you know of, and you *still* don't need to change a bit of code, then either it's never run, or it works as intended, however unclear it may be.

Delete, delete, delete

No code has fewer bugs than *no code*. If some of the code is not used, delete it. Some people have a fear of deleting unused code, in case they need it sometime, but this fear is unnecessary – if you're using a version control system, then the code will be there if you need it. Tag the last version before you delete it, to make it easy to find again, if you wish, but do delete it from the current version of the code. Unused code just makes it harder to understand what the rest of the code does.

Whilst unused classes and functions cause clutter, the worst offender in the unused code stakes is an unused branch of a function that *is* used. Every time you have to read the function to try and understand it, this code gets in the way. Once you've worked out that it cannot be called, **delete it** and save yourself from having to work it out every time you look at the function.

Code coverage tools will help with this analysis, but they require that every code path actually used is exercised, which requires extensive testing, whether with automated or manual tests. Sometimes static analysers will

ANTHONY WILLIAMS

Anthony is the Managing Director of Just Software Solutions Ltd. He is a strong believer in the benefits of Test Driven Development, Refactoring. He can be contacted at anthony@justsoftwaresolutions.co.uk

Subversion of the C language

Mark Easterbrook explains that what you see is not C.

There are many problems with maintaining legacy code, and a good understanding of the language is essential because it is the only factor that is likely to be well documented, and therefore the only real "known" in the system. In this first article I will look at areas where even this island of sanity in a sea of complication is not the safe haven it should be. There are some programmers who, for reasons best known to themselves, prefer to modify the language itself before they start coding. Here are just a examples.

Forever and a day

Many embedded systems (and some not so embedded systems) contains code along the lines of:

```
FOREVER {
/* stuff */
}
```

The intention is fairly clear, so you hope that when you find the definition of `FOREVER` it is one of:

```
#define FOREVER for(;;)
```

or

```
#define FOREVER while(1)
```

and you wonder why they didn't just write one of these two well-known idioms straight in to the code in the first place.

In practice, those coders who think `FOREVER` is neat don't extend neatness to the definition, and thus you have to work hard to be sure that the loop does actually loop forever:

```
#define FOREVER while(TRUE)
```

Which of course depends on the local value of `TRUE` (more about values of `TRUE` later on).

```
#define FOREVER while(!FALSE)
#define FOREVER while(!0)
```

These probably work, but they are just slightly obscure enough to make you doubt the coder's ability and you have to double check everything to be sure.

Now to something evil, especially if you don't read it carefully:

```
#define FOREVER for(;;);
#define FOREVER while(1);
```

Normally this would not make it past the first testing stage because all the tasks just get stuck in a tight loop and nothing much happens. But in an environment where definitions are copied into every source file, sometimes with a slight modification such as a trailing semi-colon, and the compiler has a "feature" where it optimised away empty loops completely, the error can remain undetected for a long time. The "loop" only executed once and then the tasked exited. The process monitor would then notice the "failed" task and so re-start it ready for the next message.

MARK EASTERBROOK

Mark is a software developer specialising in technical domains. In his day job he works with embedded systems, high performance/reliability/availability systems, operating systems, and legacy code. The rest of his time is split between motorcycles, genealogy, linux, and food and drink, but never at the same time. Mark can be contacted at mark@easterbrook.co.uk

Building on a Legacy (continued)

be able to identify unused code, and sometimes you can tell just by looking, e.g.

```
n=3;
if(n==2){ ... }
```

Sometimes it's worth using a cross-reference tool, or even plain **`grep`**, to find whether a function is called from anywhere. If it's not called, delete it.

Version control

I said deleting code is safe because it'll still be in your version control history; this rather presumes that you're using a version control system. If you're not, start now – download CVS or Subversion, and set up a repository for your source code.

Assuming you do have a version control system, make sure you use it to full effect – check in code frequently, and label important versions, such as before a refactoring, or when you make a release. Every time you make a small change to your code, and everything still works, check it in. After each baby step, check the code in; check your code in many times a day, sometimes after only a few minutes. When you're making changes to legacy code, you'll be glad of the safety net, knowing that you've got working code to fall back to, from just a short time ago.

Stay focused, take things slowly, and work step by step, with frequent check points

Conclusion

Legacy code needn't be scary, but it does require careful handling. Stay focused, take things slowly, and work step by step, with frequent check points. Add tests as you go, and pay back a little technical debt every time you work on an area.

We should all bear in mind the problems of legacy code when we're developing, and do our best to avoid them before they're an issue. We should strive to leave a legacy we can be proud of. ■

Notes and references

1. *OED, meaning 5b for Legacy*, n.
2. *Working Effectively with Legacy Code*, Michael Feathers, published by Prentice Hall PTR, 2005.
3. *Writing Maintainable Code*, Anthony Williams, C Vu 16.2, April 2004.

The only symptom was regular "task failed – restarting" entries in the log, as as there were no other failures nobody could be bother to investigate.

somewhere someone will be relying on the broken behaviour

If you are allowed to, the best way of dealing with this nonsense it to remove the `#define`(s) and then fix the compile errors by putting in the correct idioms. If this is not permitted, you will need to remove the `FOREVER` one by one whenever the code changes for another reason and hope the process police don't catch you.

Open to extension, open to change

There are those for whom C doesn't have enough keywords, and it is so easy to add a few more, for example:

```
until (condition) {
  /*stuff*/
}
```

and if you are lucky this will be:

```
#define until(x) while(!(x))
```

and if you are unlucky:

```
#define until(x) while(!x)
```

Whatever you do, don't fix this by changing the latter to the former, because somewhere someone will be relying on the broken behaviour:

```
until (match_found && !user_cancel) {
  /*stuff*/
}
```

The Wrong Trousers (apologies to W&G)

There are others for whom the problem is not enough keywords, but the wrong keywords – because it is the wrong language:

```
#define BEGIN {
#define END }
#define FOR(b,e,s) for(i=b;i>e+1;i=i+s)
#define THEN
```

which allows the C-challenged coder to write:

```
FOR(1,9,2) THEN
BEGIN
    /*stuff*/
END
```

I once came across this in a coding standards document as a suggestion to avoid bugs caused by incorrect use of the **`for`** statement. The author of the document was promoted to CIO, but not fast enough to avoid other collateral damage on the way to the top.

This madness is probably from a closet BASIC programmer, but I have also come across the lower-case version indicating a closet algol programmer.

It's illogical Captain (with apologies to S-T)

The purveyors of the above nasties can easily be shown the error of their ways (the application of sharp implements often helps), because it can be argued that they are changing the code language, but everything else is game, right?

```
#define FALSE 1
#define TRUE 2
```

Amazingly, the mindset that created this can maintain the delusion by not using (or knowing about) the `TRUE` and `FALSE` that the rest of the world uses:

```
all_done = (((x_records == 0) &&
           (y_records == 0)
           && (cleanup_needed == FALSE))
           ?(TRUE)
           :(FALSE));
if (all_done == TRUE)
  can_exit = TRUE;
else
  can_exit = FALSE;
```

If you come from a world where `TRUE` and `FALSE` are the way the creators intended them to be, you are on your way to introducing hard-to-see bugs with a little bit of code clean up.

```
if (all_done) {
  return;
}
```

someone who read the first entry in the ASCII table and didn't know that the difference between `NUL` and `NULL` is significant

Of course, having off-by-one errors in the definition of boolean is not the only fruit:

```
#define FALSE 0
#define TRUE -1
```

then making it portable for different word sizes (their excuse, not mine):

```
#define FALSE 0
#define TRUE ~FALSE
```

and one that got spotted during testing (oh how we laughed when we tracked this one down, not!):

```
#define FALSE 0
#define TRUE -FALSE
```

and finally a rare but unfortunately not extinct species:

```
#define FALSE 1
#define TRUE 0
```

Various definitions of nothing

The other common target for alternative definition is that old favourite, `NULL`:

```
#define NULL '\0'
```

or

```
#define NULL (void*)0
```

The first I can only assume came from someone who read the first entry in the ASCII table and didn't know that the difference between `NUL` and

NULL is significant. The second is possibly the result of being stung by passing NULL to variable length argument lists on a machine where `sizeof(int)!=sizeof(void*)`. However, neither introduces bugs into the code most of the time. This means they are impossible to "fix" if someone plays the "if it ain't broke don't fix it" card. The NULL keyword and its look-alikes turn out to be extremely resilient to abuse, allowing all sorts of interesting combinations that shouldn't work, but do:

```
char* mstring = NUL;

if ((dynamic_pointer=malloc(CHUNK_SIZE))==NUL)
  exit(1);

char etc_path_str[]={'/','e','t','c','/',NULL};

if (newname[0] == NULL)
  strcpy(newname, DEFAULT_NAME);
```

Swap, swop, and SWAP again

We all know the difficulty of writing a swap macro in C, but only a few know a good solution (hint: it does involve the pre-processor). The problem is that usually you need to define a temporary, and that is really difficult to do so you have to cheat a little. This leads to some creativity.

There are temporary-less solutions:

```
#define SWAP(a,b) (a) ^=(b); (b) ^=(a);
  (a) ^=(b);
```

(by the way, I don't know if this works or not, because I didn't wait to find out – when I found this I just deleted the line and addressed each compile error manually).

partly generic solutions:

```
#define SWAP(a,b) {long t; t=a; a=b; b=a;}
#define SWAPP(a,b) {void* t; t=a; a=b; b=a;}
```

and non-portable solutions:

```
#define SWAP(a,b) do {typeof(a) t; \
   (t)=(a); (a)=(b); (b)=(a); \
  } while (0)
```

The author of the last one would argue that he has read all the books on safe macros and "knows" his solution is right, after all, he has dealt with

the developers decided to make all the compiler warnings go away

the two big gotchas.

As an experienced legacy code maintenance programmer you know you've found your bug when you get to:

```
/*swap left and right values and move to next
entry*/
SWAP(*left++, *right++);
```

Was it the comment that tipped you off? The author obviously wasn't sure what he was doing so added the comment the make sure the code couldn't be mis-interpreted! Or perhaps it was the **SWAP** shouting at you. Unfortunately in my experience, the broken

```
  #define SWAP ...
```

flavours are outnumbered by about 2 to 1 by the much sneakier

```
  #define swap ...
```

The NULL keyword and its look-alikes turn out to be extremely resilient to abuse, allowing all sorts of interesting combinations that shouldn't work, but do

family, which doesn't shout at all.

But, it's already been fixed...

If you are working on code that is not too old, you might be lucky to find some of the above coding techniques in the form the original programmer coded, but more than likely, someone has tried to fix them at some point.

In one case the developers decided to make all the compiler warnings go away. This is a dangerous activity unless approached with the right attitude. Typically, getting rid of all those pesky warnings about re-defines results in:

```
#ifdef FALSE /* avoid compiler warning */
#undef FALSE
#endif
#define FALSE 1
```

Another flavour:

```
#ifdef FALSE
#undef FALSE
#define FALSE 1
#endif
```

A third flavour, to give you real confidence about programmer knowledge and ability:

```
#ifdef FALSE /*if we included the standard header
*/
#undef FALSE
#define FALSE 0 /* use the standard definition */
#else
#define FALSE 1 /* else use in-house definition */
#endif
```

Sometimes these errors are picked up at code review time, and the source code revision system recorded the change:

```
before: #define FALSE 1
after:  #define FALSE (1)
SCCS: Code review change - Coding standards say
all #defines must use parentheses around all
variables and all expressions.
```

You might note that this seems a particularly enlightened development team:

1. They use code reviews.
2. They use a source code control system
3. They have coding standards containing useful advice
4. They add comments at code check-in time.
5. The comments are meaningful.

With all this going for them, you might be tolerant of a slightly dodgy **#define**!

Your def or mine?

You have probably noticed that many of the above, as well as generating compiler warnings, are sensitive to include order, so:

```
#include "/home/dave/project/includes/global-
stuff.h"
#undef FALSE /*need to remove our defs to stop
compiler warnings*/
#undef TRUE
#include <stdlib.h>
#ifdef FALSE
#undef FALSE
#endif
#ifdef TRUE
#undef TRUE
#endif
#define FALSE 1 /*need to put our defs back*/
#define TRUE 2
```

Dave is no longer with the company, but they cannot delete his account because to do so breaks the build. Also, nobody is allowed to change `globalstuff.h` and they won't tell me why (I think I know, and I think they don't!). I'll excuse them the misunderstanding about the pre-processor vs. the compiler error message.

Once a constant, not always a constant

The following code always used to work, but the last few releases have had problems with channels that won't start properly:

```
for (channel=0; channel<MAX_CHAN; channel++)
  init_channel(channel);
```

There is not much to go wrong with a compile time constant is there, and none of the channel initialisation code has changed in any way at all, so why do a random number of channels get initialised? How can a constant not be a constant? - this is how:

```
#define MAX_CHAN GetMaxChan()
```

You see, the great thing about `#define` constants is that when they change, you don't need to go through the code changing all instances, just change the single definition and recompile. If it changes into a value read from a configuration file instead of a constant, you don't need to go through the code changing all instances, just change the single definition and recompile. It's still a constant, just one that cannot be used before it is read from the file.

Fixing this should be easy, just make sure that the configuration file is read as the very first initialisation step. However, I forgot to mention that file was not local, if this is the slave card it obtains the configuration from the master card, and communication to the master is, you guessed it, over channel 0. ■

Are we done yet?

(slightly tongue-in-cheek) guidelines:

1. Never "fix" the `#define` by changing it. It must either remained unchanged or be deleted completely. Obviously removing it means changing everywhere it was used. (but see 2)
2. Never "fix" the `#define` by removing it if this won't work. `#define FALSE 1` falls into this category.
3. If someone thinks it is a good idea to modify the language, they probably don't understand the spirit of the language and certainly don't understand the letter of the language (accidental pun), so don't trust any of the code they write.
4. `#define BEGIN {` and similar travesties is like a big flashing neon sign saying "get off this project ASAP" – probably time to get the CV up-to-date.
5. Never ever change the order of `#includes` – the code is almost certainly relying on the correct order of `#(re)define`.
6. Don't believe anything you read, especially if it is written IN CAPITAL LETTERS.
7. Code reviews are useless if they consist of the blind leading the blind – there must be at least one good programmer present!
8. Don't allow Coding Standard Lawyers in to code reviews – they will use all the time to correct the code and leave no time to spot the errors.

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.

QEMU as a means of software distribution

Silas Brown introduces a cross-platform solution.

If you are writing software for others to use then it would help if it runs on their computers (unless it can be done entirely on a network server, which is not always feasible).

If you use a Unix-like environment for development, you will have many command-line tools and other software that can easily be used and/or customised in your own program, saving you time, but when it comes to distributing it, you'll find it's difficult because the user does not have the same environment. And thus you'll either have to re-implement the functionality of the tools yourself, or find equivalent tools that have suitable redistribution licenses (perhaps by porting them yourself); both of these approaches can take a lot of time and the result is not always as good as the original, particularly if you are using esoteric functionality.

QEMU is a free cross-platform x86 emulator which can run any operating system you like without needing administrator privileges or installation

You could require the user to load a suitable environment, but this can be a hassle, even with such products as Cygwin and live Linux CDs available. Even if the user has time for it, he or she might not have administrative access to the machine, or the machine might be an awkward laptop that cannot boot live CDs, or something.

In many cases it suffices to restrict your choice of tools as far as possible to what is available in the language interpreter or portable libraries. But when you need to go much further, an option which has recently emerged is to distribute a virtual machine under QEMU.

QEMU is a free cross-platform x86 emulator which can run any operating system you like without needing administrator privileges or installation. Ready-made disk images are available, some of which are relatively small and can be customised relatively easily (and you can do this customisation while using any operating system under which QEMU runs; you don't have to run the user's operating system yourself). For example, try going to www.damnsmalllinux.org, choosing a download mirror and looking for the `dsl-<version>-embedded.zip` file (to run this in Linux you will need libSDL on your system; Windows should be able to run it as-is, and it is also possible to run on Mac OS but you need to download the Mac version of QEMU separately).

The file `qemu/harddisk` appears as `/mnt/hdb` and can be accessed from outside the emulated machine by using the Unix mount **`-o loop`** command. You can use this to move large files (such as software installation files) onto the emulated machine (remember to unmount before running). Alternatively you might be able to access the network from the emulated machine, but this doesn't always work. Once the required files are there, it's best to do any actual installation from within the emulated machine, to prevent configure scripts from linking to an outside library. Remember to install to `/mnt/hdb` not `/usr/local` as the latter is not preserved. If you need more space on that virtual disk (which is fairly small to begin with), you can shut down the emulator and use standard Unix commands to do the job, for example for 500M:

```
cd qemu
dd if=/dev/zero of=hd2 bs=1048576 count=500
/sbin/mke2fs hd2
mkdir a ; mount harddisk a -o loop
mkdir b ; mount hd2 b -o loop
cp -pr a/* b/
umount a ; umount b
mv hd2 harddisk
```

The virtual disk also contains a file `backup.tar.gz` which has the home directory, but since it will be loaded into RAM it cannot be very large. It is useful for storing configuration options and startup scripts.

It is possible to start the emulator from a saved state so that it does not have to emulate a machine startup. This is less useful if you want the user's actions to change the virtual hard disk (unless you are clever about mounting). But if it is to be a "read-only" system then you can get it into the state you want, press Control-Alt-2 (if QEMU recognises Control-Alt on your system) and type **`savevm filename`**, then close the emulator and edit the DSL script so that **`-loadvm filename`** is specified on the qemu command-line.

In many cases it is useful to read input from and write output to the user machine's filesystem rather than to a virtual hard disk. If it's possible to run a separate server of any kind, then this can be reached from the emulated machine at IP address 10.0.2.2 even if networking is not otherwise available (Samba sharing can also be set up if you have the server). However, this approach means more end-user setup and it is different across platforms. A small Python script can make an adequate cross-platform server (*py2exe* can eliminate the need for Windows users to install Python, and other platforms likely already have it), but be careful to allow connections only from localhost, and be especially careful on a multi-user machine. For batch jobs it might be better to use QEMU's **`-nographic`** option and to have the scripts within the emulated machine interface with the serial port, which will appear as QEMU's **`stdio`** to the outside Python program.

if you are writing software for others to use, then it would help if it runs on their computers

QEMU does run more slowly than other approaches, but as long as the user has sufficient hardware this is normally more than outweighed by its not requiring any installation or setup or even rebooting. Users with old hardware are more likely to be willing to set up an appropriate environment anyway (although this is not always the case!)

Incidentally, the lead author of QEMU, Fabrice Bellard, is also responsible for QEmacs (an editor which although not being as customisable as Emacs still has some customisability and has the advantage of being able to edit files much larger than the physical RAM without being slow) and TCC (a small, fast C compiler) among other things. ■

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

ACCU Conference 2006: Retrospective

Pete Goodliffe introduces a round-up of the ACCU conference.

Another year, another conference. For many people the ACCU developers' conference has become an unmissable part of their calendar: a must-attend event full of high-quality presentations and the chance to meet like-minded individuals who love to code *well* and want to constantly improve. For many `accu-general` mailing list lurkers, it's a chance to meet up with the people we only know by email address. Usually, the shock isn't too bad!

This year's event followed in the grand tradition of its forebears: it was exceptional. It provided high quality industry-renowned speakers (including Herb Sutter, Scott Meyers, and Guido van Rossum) and talks covering many aspects of software development. There was a strong focus on C++ and Python, distributed computing, and dynamic languages, nestling amongst many, many other topics. The presentations were excellent, and I don't think anyone can say they didn't come away with new techniques, perspectives, and insights into software development under their belt.

But the conference is about much more than that: it's a wonderful opportunity to bump (sometimes literally) into other programmers (and often into the industry luminaries themselves) over a coffee, a beer in the bar, or at whichever restaurant we've taken over that night. I still feel sorry for the Pizza Express staff who coped admirably; Chutneys (the local curry house) are used to us by now!

Highlights for me included the lively debate between Herb Sutter and some C++ standards committee members about the naming and design of Microsoft's C++/CLI (including Lois' classic quote: *stop talking whilst I'm interrupting you*) [1], the quality of prop used in Alan Kelly's session, and the number of people who wished they were on stage during the "Grumpy Old Programmers" panel [2].

To impart more of a flavour of the sessions, a number of attendees have written up their experiences. Fasten your seatbelt.

Mark Easterbrook

Mark (mark@easterbrook.org.uk) made these remarks about a couple of interesting sessions.

Only the Code tells the truth (Peter Sommerlad)

Although there was only a small audience – the cartoon room was about half full – this presentation generated considerable discussion over the next few days. The core of the presentation was a list of things we do that lead to less understandable code such as comments, type safety, design before you code and never change a running system. The first of these, comments, generated the question: "Is the total value of all the comments in all of the code out there, greater or less than zero?". (If I remember correctly, vocalised by Russel Winder). Enough participants considered the answer to be "less than zero" resulting the phrase "Comments are Evil". This was more of a workshop and left the presenter and audience with lots of ideas to take away and work on.

Understanding Security With Patterns (Peter Sommerlad)

Like Peter's earlier presentation this was interactive involving group discussions that moved away from the Patterns and concentrating on Understanding Security.

The groups picked a subject area and evaluated it with respect to Threats, Vulnerabilities and Risks. The two most significant issues that resulted were the difference in the value of a data asset depending for read only and read-write access, and the significant change in value when composition happens, either upwards such as combining a credit card record with the customer record that includes the address, or downwards when abstraction also occurs.

Steve Love

Steve (steve@essennell.co.uk) comments: the conference is a highlight of my year. It gives me a chance to mix and meet with a community of programmers who have at least one thing in common: the desire to be better at what they do. Of course, this isn't the only reason to attend! The ACCU has an unmatched technical programme presented by a wide selection of speakers, from jobbing programmers to language designers, and it's a real treat to be able to learn from the brightest in the business. Every day of the conference was a full one for me, but the following three presentations all stood out for one reason or another.

Effective Version Control (Pete Goodliffe)

Pete Goodliffe was his usual ebullient self in this talk about making the best use of version control systems in software development. Humour in the slides and Pete's ever-present enthusiasm brought light and sparkle to what many consider to be a very dry topic indeed. Those of you who've ever seen Pete give a presentation on anything will know that there is always much more than a delivery of the slides, and this was no exception (slight hangovers notwithstanding). There was even bingo to ensure everyone paid attention!

pick the simplest version control system you need, not the most advanced one you can afford

Without delving too deeply into the details of the system administration associated with version control systems, or focussing too sharply on the excellence or shortcomings of specific products, Pete conveyed a general sense of what's required of a good version control system by using examples of what developers need from it.

Basic principles started the show: the differences between models of how edits and check-ins are handled gave Pete a good excuse to make some comparisons between version control systems, and the importance of repository/project planning was tackled well. Simple diagrams gave a clarity to this which even talking around it probably couldn't achieve. Advanced techniques followed, with clear examples of managing different lines of development through branching and merging, and moving on to managing third party libraries and product releases.

Throughout, Pete's wide-ranging experience was always ready with anecdotes and practical tips. Of all the pieces of advice given, the one that struck chords with me was to pick the simplest version control system you need, not the most advanced one you can afford.

In his allotted 90 minutes, Pete gave good coverage to this huge topic, and that is not to say it was skimmed in any way. Obviously all details cannot be covered in a presentation like this, but the important stuff that matters to developers' everyday lives was handled with Pete's easy confidence.

CONFERENCE ATTENDEES

The people who attend ACCU conferences are a well-adjusted set of individuals with excellent judgement and top-notch programming skills. They can be contacted at accu-general@accu.org

A Design Rationale for C++/CLI (Herb Sutter)

Herb Sutter gamely volunteered to be grilled by the audience and a panel of experts over the design of C++/CLI. If you've read Lois Goldthwaite's 'Standards Report' in the previous two issues of CVu, you'll be aware that there is some tension between Microsoft's and ECMA's wish for a fast-track ISO standard for C++/CLI, and the BSI C++ Panel. Quite a few members of the BSI C++ Panel regularly attend the ACCU Conference, and so this session promised to be, well, entertaining, anyway. Now you might be thinking that concerns over fast-tracked ISO acceptance (or concerns over the very name C++/CLI – more later) might not be quite the same thing as a Design Rationale. You would, of course, be right. But the discussion became one of why the BSI C++ Panel hold such strong objections to both the fast-tracking, and the name, and at least in part why Microsoft, and Herb in particular, feel these objections are ungrounded.

the design choices made by any language or language binding are extremely important

In fact, Herb Sutter, having obviously guessed that the discussion would head in that direction, came with a presentation of only 3 slides. The final slide invited questions, and a small impromptu panel was formed of Kevlin Henney, Francis Glassborow, Roger Orr, and Lois Goldthwaite. The ensuing discussion effectively revolved around the difference between a language binding – which is what the C++/CLI is considered to be by Microsoft and the ECMA standard - and a new language altogether.

The discussion was lively, often alternating between Lois, effectively leading the panel, writing examples on a whiteboard, and Herb clarifying points of interest from the FAQ section of the *Design Rationale for C++/CLI* paper (available at http://www.gotw.ca/publications/C++CLIRationale.pdf for those interested). In particular, the audience was reminded by Herb of Microsoft's commitment to C++ as a first-class language in .Net, and by Lois that Microsoft's commitment to an open standard is to be applauded and not underestimated.

There was certainly a political, rather than technical, debate, but the design choices made by any language or language binding are extremely important because they affect the everyday lives of the people on the ground-floor – the programmers themselves. And the design choices are about much more than just syntactic arrangements of code.

Concurrency Requirements (Hubert Mathews)

Hubert Matthews gave the final keynote presentation of the conference. Delivered with his customary authority and good humour, this was a talk about changing the way we think about concurrent systems. What set this apart from, say, Herb Sutter's keynote on writing concurrent programs, which was about changing the way we think about writing concurrent code, Hubert's keynote focussed on concurrency in the requirements - the Real World being represented.

The key point was that this concurrency has no real technological solution. One example given was that of borrowing a book from a library, which indicates two aspects of the problem. The first aspect is a race-condition, because between asking whether a book is available and going to the shelf to obtain it, the book may have been taken. The second aspect is one of system integrity. The solution to the book not being available is not to go and put the book on the shelf, but to update the 'system' to reflect the real state of affairs.

It was a very thought provoking talk, and one point that stood out for me was Hubert's presentation of the ubiquitous traffic light problem. On a crossroads, if all the sets of lights go green at the same time, chaos will ensue. Generally when we model this in software, these are the terms in which we think. Hubert suggests a more goal oriented approach to the problem. The goal is to prevent scratched paintwork – an entirely real-world one – not to prevent all the lights going green.

Anonymous attendee

A writer who wishes to remain anonymous made a set of interesting notes on some of the sessions he attended:

The Future of Python (Guido van Rossum)

Guido van Rossum, the creator of Python, opened the conference with a talk about Python 3000 (AKA Python 3.0, AKA Python 3k). This will be the next major release of Python, a release that is allowed to break backwards compatibility with Python 2.x. This is not as bad as it sounds, as Guido takes backwards compatibility very seriously and a major part of the Python 3000 work will be helping to smooth the transition.

Guido covered many specific changes, which are documented in PEPs 3099 and 3100. Briefly, the feel of the language is not going to change. The main objective is to clean up some clutter and inconsistencies that have crept in along the way.

Migration from Python 2.x to Python 3000 is considered very important. 2.x versions will continue to coexist with Python 3.x versions up to something like version 2.9. Python 2.x versions already warn about features that are scheduled for deprecation, and this functionality will be expanded. Tools will be available for inspecting code and helping identify areas where changes will be required. PyChecker was mentioned several times in this regard.

All in all, it was a good talk and encouraging to hear that so much effort is going into smoothing the migration from Python 2.x.

Five Considerations in Practice - Kevlin Henney

This talk follows on from a keynote talk from 2005 titled simply *Five Considerations*. This year Kevlin expanded on the ideas and presented practical examples. Given the many heads in the room nodding regularly in agreement, the subject matter was familiar to many people's experience even if they hadn't codified it so well as Kevlin did.

The five considerations, an inevitable product of philosophy and beer, are: Economy, Visibility, Spacing, Symmetry and Emergence. Each is a large subject, but I'll somewhat arbitrarily pick out some concepts that appealed to me particularly.

From the Conference Chair

Ewan Milne (ewan.milne@btopenworld.com)

As I (finally!) relaxed at home on Saturday after the conference, I felt tired, but very happy with how the whole event had turned out. I won't pretend that I haven't put in a great deal of work over the past few months, and so feel a justifiable sense of pride in the success of the event. But conference organising is a collaborative process: the efforts of Julie Archer in particular are invaluable, not to mention the conference committee, sponsors and speakers. Finally it is the delegates who provide so much of the atmosphere which makes our conference such a favourite event of so many top speakers.

While I may still be a little too close to the whole event to pick out favourite sessions, a few of my favourite moments were: the discussion on C++/CLI with Herb Sutter and panel taking off after a shaky start which showed its late addition to the programme, Allan Kelly's incorporation of a sound asleep Pete Goodliffe into his session, the continuing ability of the conference audience to keep speakers of all levels on their toes, and the highly appropriate final collapse into chaos of the Family Fortunes end-note - Jez and I rehearsed it just like that, you know.

And where else but Oxford in springtime can a leading C++ author get off the coach from Heathrow and approach the nearest passer-by for directions, to be met with the response "I know you - you're Scott Meyers! Come with me, I'm heading round to the Randolph myself..."

Visibility: Much of what developers and technical managers deal with is non-concrete. So unlike a building project, for example, it can be very difficult to encapsulate artefacts in some form that is mentally digestible. This affects many aspect of the development process, including project completion estimation and reasoning about project components.

Visibility is concerned with helping developers and technical management to see aspects of the software and development process that are non-concrete. Agile methods can help with this. So can *concretising* implicit concepts by making them explicit. For example, if you find you always pass day, month and year values around together, then consider creating a `Date` object that concretises this implicit concept.

Economy: One aspect of this is keeping code simple, and therefore more readable and maintainable. This point was illustrated by the following actual real world code example:

```
if(enabled)
  enabled = false;
else
  enabled = true;

if(!(a == p || b == q))
  return true;
else
  return false;

if(loaded() != false)
  if(valid() != false)
    return true;
  else
    return false;
else
  return false;
```

Which really just does this:

```
  enabled = !enabled;
  return a != p && b != q;
  return loaded() && valid();
```

threadsafe pieces of code cannot be guaranteed to be threadsafe if used together

A new expression was also coined: *decremental development* is the art of reducing code volume without decreasing functionality.

Symmetry: This covers many ideas. Two examples are:

- Making behaviour consistent across multiple objects or domains. e.g. Eclipse allows you to right click on a unit test and run that test, or right click on a suite and run all the tests in the suite, or right click of a whole project and run all the tests in the project. The functionality is reflected symmetrically across all levels.
- Putting resource deallocations at the end of the code block where the allocation occur, rather than relying on knowledge of external flow and putting the deallocations in other related functions.

PyPy - A Progress Report - Michael Hudson

A very interesting presentation about an area of study that is quite complex.

PyPy is an effort to re-implement the Python compiler, bytecode interpreter and command shell in the Python language. This is expected to result in many benefits, including:

- Coroutines.
- Elimination of the global interpreter lock.
- A just in time (JIT) compiler for Python.
- Support for new platforms such as the JVM and the .NET framework.

The last of these means that PyPy may one day be an alternative to Jython and IronPython. However, PyPy is applicable to any platform whereas Jython and IronPython are tied to the JVM and .NET frameworks respectively.

The presentation covered an architectural overview of the project and a status update. The architectural overview was far to complex to replicate here, see the PyPy website if you want to know more.

Regarding the status update, there is quite a lot to report:

- The infrastructure is becoming quite mature.
- The Python bytecode interpreter, compiler and command shell are all now fully implemented in RPython. (The PyPy tools only build a subset of Python called Restricted Python or RPython. This greatly simplifies the tools.)
- There are three back ends that allow builds to C, JavaScript or LLVM. Of these, the C back end is very interesting because it allows RPython code to be built directly into a native binary.
- Consequently, there is a completely new C implementation of Python which is generated by the compiler tools from the RPython source. This implementation performs slightly slower than the original Cpython.
- An immediate practical consequence is that one can write any application in RPython and build it to a native binary that runs of the order of 100+ times faster than running directly on the Python interpreter.
- Longer term the project is now beginning to look into a JIT compiler for Python.

Working with C++ as if Unit Testing Mattered - Michael Feathers

Michael Feathers is the author of *Working Effectively with Legacy Code*. This presentation also dealt with legacy code, focussing on adding unit tests to legacy C++ code. Since legacy code affects many of us this talk held much promise.

Unit testing legacy code has all the obvious advantages and can also be very useful in figuring out just what the code actually does.

The main body of the talk covered techniques for breaking dependencies to enable unit tests to be added to code. Low risk is watchword as there are no regression tests in place to ensure that the dependency breaking process doesn't break any functionality at the same time.

Interspersed with this information were some additional handy hints. One that springs to mind is getting the compiler to help with refactoring tasks by deliberately introducing specially structured errors into the code, then following up on the subsequent build errors and warnings.

In summary, this was a session full of useful, practical advice. If it was any indication of what you can expect from *Working Effectively with Legacy Code*, this book will be great for anyone working on legacy code.

C++0x, Concur and the Concurrency Revolution (Herb Sutter)

The starting point for this talk was the observation that making faster processors is becoming difficult. Instead, we're moving towards vastly more parallelism with multicore processors and multiprocessor machines. This fundamentally changes scalability strategies and is forcing us to write concurrent applications.

Herb discussed the sort of technologies and language features that will be required before we can hope to write concurrent applications simply and reliably. The current technologies, threads, mutexes and semaphores are far too low level and difficult to use (not surprising since they're about 40 years old). In particular, thread safety is not composable. This means that two known threadsafe pieces of code cannot be guaranteed to be threadsafe if used together in some fashion.

We looked at several ideas that are evolving in compilers and operating systems, focussing particularly on some work that Herb's team has recently been doing. The efforts described attempt to move ideas and tricks that are currently being implemented manually into the OS and language

levels. This can be likened to the OO revolution where, for example, OO concepts like vtables which were being manually implemented in C were moved into the compiler in C++.

In particular, two things to look out for are transactional memory and C++0x's memory model which will take concurrency into account.

The Keyhole Problem (Scott Meyers)

Scott Meyers has identified a common bad practice amongst developers which he has called the *keyhole problem*. He is hoping the name will catch on and consequently reduce the incidence of the bad practice.

The keyhole problem is responsible for many of the irritations we experience daily while using computers. It occurs when developers make arbitrary design decisions that result in a unnecessary restrictions being imposed on their users.

An example is web pages that pop up windows which are too small and cannot be resized. Another is combo boxes whose drop down list is too small. Not all instances of the keyhole problem are GUI related, but there certainly seems to be an epidemic in the GUI space. Interestingly quite a lot of the examples were familiar to the audience, including Visual C++'s infamous C4786 compiler warning.

The talk was based fairly directly on a 2002 paper by Scott, which you can find at http://www.aristeia.com/TKP/draftPaper.pdf.

Producing Better Bugs – Roger Orr

This was a semi-satirical look at software development as a bug writing process. We looked at a variety of bugs types and the stages of development where they tend to be produced. Always, the focus was on the elusive "good bug" which makes it all the way through the development process into production. All in all, instructional and a good laugh.

Allan Kelly

Alan (allan.kelly@actix.com) made these comments on his blog:

Most of the last week was at the ACCU conference in Oxford. Although ACCU has its roots in the C and C++ community the whole organization has been trying to move away from this for some time. This year I really felt we're succeeding.

Yes I went to some C++ talks, yes a lot of people talked C++ but a lot of other subjects were covered too. People are just as happy to talk about Agile development, products, dynamic languages and about anything else to do with software development. Overall I'm left with a lot of new information and ideas, plus things to think about and understand – I've got to update my mental maps.

For example, Peter Sommerlad did a session called "Only the code tells the truth" in which he set out challenge our assumptions about what is "good programming practice". A few slides then a goldfish bowl discussion that really opened up some dark corners and said "Is accepted wisdom right?" One of Peter's questions was "Do comments help code maintainability" – this got a lively reception. I must admit I'm tending towards the view that comments in the code don't.

The conference was full of interesting people presenting or just attending. During the last hour or so I got talking to Gunter Obiltschnig of Applied Informatics in Austria. They have one of those libraries (C++, portable components) that sounds really useful, just I don't have any need for it right now. One-day maybe.

Paul Grenyer

Paul (paul@paulgrenyer.co.uk) added these thoughts on some memorable sessions:

The keyhole problem (Scott Meyers)

The presentation that stood out the most for me was Scott Meyers' The Keyhole Problem. Although this non-technical talk would have been better presented as a key note, it made me realise that there are different ways to present things and not everything needs to be technical.

The presentation mostly consisted of examples of poorly designed GUIs and websites where the best use of the available space was not made or scrolls bar were missing and therefore prevented content from being read. For example drop down boxes that hold the months of the year, but only have room for 11 items and licence agreements with no scroll bars preventing the licence from being read. There were a large number of these, but Scott presented each one in a humorous way and suggested what he would do better, sometimes including how to change the markup.

One of the later examples showed how an engineers "the user will never want to do that" assumption almost cost someone their life due to a buffer overrun on a log sawing machine.

This, however, was the only serious moment in an otherwise very entertaining session. Kevlin Henney is well known for dealing with hecklers well, Scott is certainly comparable and it would be interesting to see them head to head. I missed Scott's more technical presentations, so from a purely entertainment point of view I would highly recommend attending one of his sessions.

Threading 101 (Tim Penhey)

Tim Penhey has a big personality and, with several of the rest of us, is normally in the thick of things at any ACCU event he attends. I spoke to Tim following his Threading 101 presentation and was interested to find out this was one of the first times he had presented. Threading 101 was very well thought out, pitched at a level everyone could understand and Tim transferred his big personality extremely well.

Most people in the room, including myself, had a reasonable amount of threading experience and, I suspect, were there to pick up any of the basics they'd missed up to this point. I for one finally understand what a semaphore is for and how one works. This was something the university professors failed to get through to me, but that's another long story.

Tim evidently knew his stuff inside out from the questions he was fielding successfully. I especially enjoyed his "this slide is intentionally left blank" slide at about the point in the presentation where brain overload is reached and wished I'd have thought of it for my presentation. It was too late of course by then. I was also glad to take him up on his suggestion of standing up and turning round at this point.

My only regret is that I suspect we'll loose Tim to the Python track next year. It will certainly be their gain and our loss.

Gail Ollis

Gail (gail.ollis@roke.co.uk) was amused by the message displayed by the screen at the bus stop just outside the Randolph on Tuesday morning: `Error: Unmatched { and }` (what exquisite timing!), and concludes our round-up with a short quiz to see how well conference attendees were paying attention...

Q: Is SCRUM:

(a) An agile methodology?

(b) What forms around the food & drink during breaks?

(c) What happens when a bus arrives at an Oxford bus stop?

A *All three*

Q: Are comments in code:

(a) The devil's work?

(b) A Good Thing?

(c) A necessary evil?

A *Don't know. The people who attended Peter Sommerlad's 'Only The Code Tells the Truth' are still arguing about it.*

Francis' Scribbles

Francis Glassborow sets another cryptic puzzle.

Education

Anyone who took the trouble to check the website for my book, 'You Can Do It!' would know that I have never taught in Further Education (roughly College level in the US). I think that taking time to do a little basic research is important before speculating about a writer being influenced by the popular media.

I do not think that readers should agree with me but I do think they should take a little more time thinking about their responses than seems to have been the case for the writer in the last issue decrying my previous column. Doubly so if they choose to remain anonymous.

In my opinion, anyone who believes that all is well with mathematics teaching is being complacent. One of the consequences of the level of maths teaching that is currently considered adequate is revealed if you look at what is happening in the Physics curriculum. If you do not understand what I am getting at, it is time that you did a little research.

Please note that I am not having a go at teachers, but at a system that is preventing good teachers from preparing the next generation for the technical world they will live in.

There is a fundamental problem with pervasive assessment coupled with a requirement for objective justification for such; we finish by only assessing those things that can be assessed in a provably objective way. However, many attributes do not submit to objective assessment. That does not make them undesirable. How do you measure kindness? Thoughtfulness? Consideration? Tact? and so on.

You can program in C++

Yes of course you can, but that is the title of my latest book which will be on general sale this side of the Atlantic by the time you get this issue of C Vu. It will be released in the US shortly after you get this issue. It is already being translated into French and Slovakian. In case you are interested, I received more royalties for the French version of 'You Can Do It!' last year than I did for the English version.

This book is written for the reader who can already program in some language and wants to learn the basics of C++. I try to identify the places

FRANCIS GLASSBOROW

Francis is a freelance computer consultant and long-term member of BSI language panels for C, C++, Java and C#. He is the author of 'You can do it!', an introduction to programming for novices. Contact Francis at francis@ronbinton.demon.co.uk

ACCU Conference 2006: Retrospective (continued)

Q: Is Von Neumann architecture:

(a) Dead?

(b) Poorly?

(c) Alive and well?

A *Herb Sutter's diagnosis suggests we should start writing the obituary!*

Q: Is Python:

(a) Indented?

(b) Invented?

(c) Indispensible?

A *According to his much misread T-shirt, it's Programming as Guido indented it.*

Q: Is SOA:

(a) Hope?

(b) Hip?

(c) Hype?

A *According to Nico Josuttis, (c) - fuelled by the fact that it uses the right vocabulary to press management's buttons.*

Q: How many web application frameworks does Python need?

(a) One?

(b) More than one?

(c) Lots?

A *A look at the schedule suggests it's (c)!*

Q: Is good software:

(a) Simple?

(b) Simplistic?

(c) Complex?

A *Read what Giovanni Asproni had to say. He crystallised this BRILLIANTLY. From a {cvu} point of view, it's worth noting that he cited writing as a good means of learning for the writer as you have to marshall your thoughts. Ironically, this presentation clashed with the panel discussion on 'writing for ACCU' so no-one got to hear both. (If anyone can offer a summary of that panel, I'd be pleased to see it published).*

Q: What fosters better communication:

(a) Email?

(b) Telling people to communicate?

(c) Having a supply of sweeties on your desk?

A *In Tuesday's tutorial, Nico & Jutta may well have increased UK sales of gummy bear sweets! Apparently people are more likely to drop by & talk to you if there are sweeties on your desk!*

That's all, folks

Once again, all conference attendees owe a debt of gratitude to the conference organisers and committee, who clearly worked very hard to make an excellent event. Plans are already under way for next year's event, and if you've never been to an ACCU conference, hopefully this article will have given you a taste of what you're missing. See you in 2007! ■

Endnotes

1. Unfortunately, we'll have to wait for another conference to see Herb and Lois arm-wrestle!
2. Apparently the ACCU conference is well-known amongst speakers for the quality of its hecklers. Sorry, that should read: the quality of the *delegates*!

where the reader maybe surprised because of the differences between C++ and the previous programming experience.

For example, those coming from a functional programming background are likely to be surprised by the degree to which C++ code relies on assignment. That will just be a surprise but one they will generally overcome. More subtle is that their instinct will be to use recursion for iteration. That will work in C++ but C++ compilers are not generally optimised for recursion. Rather more important is that C++ programmers will not understand source code that uses recursion for iteration.

That is a special case of a more general problem; programmers who do not write idiomatic code make their source code much harder to maintain because the code will be harder to follow.

Symbian C++

I am interested in any readers who have experience in programming in C++ for the Symbian OS. I have an ulterior motive because Symbian have asked me to write an introductory programming book using their OS and C++.

At the moment it is giving me some concern because it seems to me that Symbian C++ has a strong local idiomatic style all the way down to special naming conventions (which seem to conflict with those used by Microsoft). They also seem to be relying on a very old C++ style and I wonder just how inherent that style is. Some of it is the result of the original being developed for use in a very constrained environment. However, my feeling is that some of it is a result of those writing the guidelines not understanding how C++ was going to change between 1995 and 2005.

I would be grateful for readers comments and possibly guidance. In particular I would very much like to contact anyone who is using Carbide C++.

Book reviews

I have just finished sorting out books that have been waiting more than 18 months for a reviewer. As I packed them away ready to be donated to Oxfam, I was struck by the number of apparently good books that were among them.

Please check the books that are currently available for review because I would not want to be 'remaindering' another batch of good books on November 1st.

Problem 26

```
#include <iostream>
int main() {
  int i(0);
  std::cout << "Please type in a number "
  "between 0 and 255: ";
  std::cin >> i;
  std::cout << i * i;
}
```

Please identify all the possible problems with the above program (both compile time and execution time).

Problem 25 commentary

Here is my problem (assumes at least Oracle 9i):

```
INSERT INTO my_table VALUES (
(SELECT my_sequence.NEXTVAL FROM DUAL), ...);
```

The intent is to insert a non-repeating auto-generated value into a primary key field of a table. The problem is that this method is simply naïve.

Yesterday was 1st May and my email box has remained empty so I guess that the SQL experts in ACCU considered the above problem was too simple for their attention. As no one else responded, I think they were mistaken. Here is William's commentary which he supplied with the problem (Thanks William. How about contributions from other readers.)

It is a problem for the following reasons:

1. It does not require that the sequence by used every time something is entered into the table
2. It does not assure that one will not enter a previously used value (although, presumably, a duplicate key error would be raised by virtue of the primary key)
3. It does not assure that the primary key value cannot, at some later time, be changed inducing any number of errors (not the least of which is the possibility that the newly updated key would be some future value of the sequence) .

Resolution is also not clear. Normally, this problem will be resolved with the creation of an insert trigger:

```
CREATE TRIGGER bi_my_table
BEFORE INSERT
ON my_table
FOR EACH ROW
BEGIN
  SELECT my_sequence.NEXTVAL
  INTO :new.my_primary_key
  FROM DUAL;
END;
/
```

While this problem addresses and properly resolves (1) and (2), it fails to address (3) which must be handled by an update trigger:

```
CREATE TRIGGER bu_my_table
BEFORE UPDATE
ON my_table
FOR EACH ROW
BEGIN
  :new.my_primary_key := :old.my_primary_key;
END;
/
```

Cryptic clues for numbers

Last issue's clue

'All for one', no, 'Musketeers for musketeers'. You will need some powerful luck to get this one.

This time I had only two responses. Unfortunately the first one had misunderstood my clue and the second produced an alternative clue that had me baffled for almost a day even though I knew what the answer was. Clues must be such that when the solver has the answer they can see it is correct.

In the above clue the first sentence is supposed to suggest 343 (3 musketeers + the sound of 'for'). The second sentence is intended to provide confirmation – the cube (a power) of 7 (a lucky number) is 343.

This issue's clue

Jeans? Maybe. ISO Labour Day? Definitely in Japan. (3 digits) ■

Standards Report

Lois Goldthwaite reports on the international standards committee meeting held in Berlin during April.

The mood was all business at the April meeting of the international C++ standard committee (WG21) in Berlin. All of the subcommittees buckled down to the serious work of refining the specification of new features to be included in C++0x.

The current working draft can be found at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2009.pdf.

With the large UK delegation, we were able to have a representative in each subgroup.

Among the new features intended to make C++ programming more pleasant, and now included in the current draft, are a new use for the **auto** keyword, to allow the compiler to deduce the type of a variable from its initialiser:

```
int foo();
auto x1 = foo();           // x1 : int
const auto& x2 = foo();    // x2 : const int&

float& bar();
auto y1 = bar();           // y1 : float
const auto& y2 = bar();    // y2 : const float&
```

This should greatly simplify some variable declarations for uses of the standard library:

```
std::map<std::string, std::string> mymap;
auto it = m.begin();
  // it: std::map<std::string,
  // std::string>::iterator
```

Code using **auto** with its current meaning – "this variable has automatic storage duration, meaning it lives on the stack" – would still compile as now, but such usage is extremely rare in real-life code, because **auto** is nearly always completely redundant.

Another change would eliminate the compiler error which now results from a forgotten space character in a nested template parameter:

```
std::list<std::vector<int>> mylist
  // look ma, no space!
```

the UK delegation had an opportunity to discuss with other participants our concerns about the Ecma C++/CLI document

This is more in the category of eliminating a nuisance than enhancing the power of C++, but many programmers regard the required space as an aggravating wart in the syntax.

Another new feature would allow a constructor for a class object to invoke a sibling constructor, as is possible in Java. The aim is to prevent code duplication and possible errors of inconsistency. The original proposal to add this feature came from Francis Glassborow of the UK delegation.

Two other UK papers, these from Paul Bristow, were adopted into the library section of the draft standard, to facilitate handling of floating point numbers:

- The new **digits10** member of **numeric_limits<FloatingPointType> >** shows the number of decimal digits guaranteed to be correct, after rounding.
- A second proposal adds a new manipulator to **iostreams** called **defaultfloat**, which would cancel previously-specified fixed- or scientific-format manipulators.

With the exception of mathematical special functions, all of the features of the first *Technical Report on Library Extensions* were adopted wholesale into the working paper, and will be moved into namespace std. A second *Library Extensions Technical Report*, targeting application programmers rather than library authors, is in preparation. One of its features would provide standard interfaces to file systems and directories, for greater code portability.

More revolutionary changes to C++, such as the addition of 'concepts' constraining template parameters, and a full-fledged lambda function capability (the UK's Valentin Samko is one of the authors sponsoring this), are in the pipeline but not yet fully specified in formal standardese language.

While in Berlin, the UK delegation had an opportunity to discuss with other participants our concerns about the Ecma C++/CLI document now undergoing a fast-track ballot within ISO (and discussed at length in previous standards columns here). There was general agreement that these concerns are well-founded. ■

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

Student Code Critique Competition

Set and collated by Roger Orr.

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

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

Before we start

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

Student Code Critique 39 entries

The student wrote: "I wanted to learn how to use STL in my own code.I've got a data structure that currently has 'start()' and 'size()' methods returning the start address and the size of some internal data structure.So I decided to try and write it an iterator so I can use the standard algorithms.

"I'm getting a bit stuck – so I've simplified it down as much as I dare but the test code still fails to compile if I uncomment either of the lines marked 'ERR:'.Please help me get my iterator class working.

"The real class is much bigger than 'tester' and doesn't use 'int' but does have the start and size methods.

```
#include <algorithm>

template< typename T >
class iterator
{

public:
  // construction
  iterator(T* p) : mPtr( p ) {}
  iterator& operator=( const iterator& rhs )
  { mPtr = rhs.mPtr; return *this; }

  // Comparison
  bool operator!=( const iterator& rhs )
  { return mPtr != rhs.mPtr; } const

  // iterator operations
  T& operator*() { return *mPtr; }
  iterator operator++() { return ++mPtr; }
  iterator operator++(int) { return mPtr++; }

private:
  T* mPtr;
};
```

```
class tester
{
public:
  tester()
  { for ( int i = 0; i < size(); )
    data[ i++ ] = i; }

  int * start() { return data; }
  int size() { return thesize; }
  iterator<int> begin()
  { return iterator<int>( data ); }
  iterator<int> end()
  { return iterator<int>( data + thesize ); }

private:
  static const int thesize = 10;
  int data[ thesize ];
};

#include <iostream>

void print( const int & i )
{
  std::cout << i << std::endl;
}
void incr( int & i )
{
  i++;
}

int main()
{
  tester t;
// ERR: std::for_each( t.begin, t.end, incr );

  iterator<int> begin = t.start();
  iterator<int> end = t.start() + t.size();

  begin++;

  std::for_each( begin, end, print );
// ERR: std::for_each( begin, end, incr );
}
```

From Balog Pál

pasa@lib.hu

A fast scan of the iterator class reveals plenty of suspicious things:

- No default ctor
- A non-explicit converting ctor
- **operator=** present without copy ctor and dtor
- **operator !=** present without **operator ==**
- **operator*** present without **operator->**
- strange signature on preincrement
- odd implementation of postincrement

But none of those should cause a compilation error in **main**, so let's nail that down first.

ROGER ORR

Roger Orr 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

Uncommenting the last line we indeed get error: the compiler can't convert `const int` to `int&` stopping in `for_each`, line `_Op(*_F)`. Strange. To corner the problem let's use `incr(*begin)` instead of `for_each`. Yeah, same error. While `print(*begin)` compiles fine. Our `operator*` looks fine, is it the one actually used by the compiler? I put a breakpoint on `print(*begin)` and single-step from there.

Yeah, there we're in our `operator*`. That returns `T&`. But wait, what's that hanging `const` 2 lines above?

Very funny, Scotty. Now beam down my clothes.

Well, the `const` in `operator !=` got misplaced by one line, and the compiler correctly reads `operator*` to return `const T&`. Thus the issued error makes sense after all...

Moving the `const` one line up `incr(*begin)` compiles fine, and so does the original `for_each`.

Now let's see the first. It doesn't compile, saying:

"`error C2664: 'for_each' : cannot convert parameter 1 from 'class iterator<int> (void)' to 'class iterator<int> (__thiscall *)(void)' None of the functions with this name in scope match the target type`"

Yeah, sure. So many words, so little clue. Bring up deja vu too, if we ever used templated code....

The actual problem is a simple typo: we forgot the `()` after `begin` and `end`. So the compiler uses the function itself instead of the result of calling the function. And we can feel lucky that we get a compilation error sometime somewhere downstream, in other cases the code could just compile and do some nonsense. I got burnt by such missing `()` several times, especially with predicate functions used in `if()`. Interesting, that in some cases there is a warning but not for others:

```
if(vec.empty) ...   // warning C4551: function
                    // call missing argument list
if(0 == vec.size) . // no warning!
```

So we have to look out even if we get some compiler feedback, a function name without `()` can convert to a pointer to function and that pointer can fit into the expression fooling the compiler that can't read our minds. Back to the point, we insert the missing parens and the code example finally compiles. Time to get back to that iterator class.

I never wrote an iterator before but if I needed one my first visit is to the standard and look up the requirements. That's section 24.1. By the content I guess the mark was a forward iterator, so we look at table 74.

Lines 1 and 2 requires the iterator be default constructible, so we'll need such a `ctor`.

Lines 3, 4 and 7 requires to be copy-constructible and assignable. Our state has just one non-owning pointer, so the auto-generated stuff will be alright. We better remove the hand-written operator= to avoid redundancy and confusion.

Lines 5 and 6 ask for operators `==` and `!=`, so we'll need that `==`.

Line 8: `operator*` looks right. (Really-really!)

Line 9: `operator->`: we need to add one.

Line 10: preincrement shall return `T&`

Line 11: postincrement: an implementation is present that is different to ours in shape. Does it have the same semantics? I think yes, but have to think hard.

Line 12: `*r++` returns `T&` looks OK.

Reading the standard further we discover `iterator_traits` will rely on some `typedef`s inside the iterator class, and that we have some framework ready to help us define our own iterators. There's a class `std::iterator<>` supposed to be used as a base class that will provide those `typedef`s. Then we can leave `iterator_traits<our_iterator>` alone. The name also suggest we better pick up another one for our special iterator. Here I'll use `arr_iterator`, the author who knows the actual purpose shall pick a better name.

The only open issue of the front list is the `ctor` taking a raw pointer and its being implicit. The usage in `tester` suggests that construction from pointer is intended, but explicit is probably OK. The general guideline is to make all converting `ctor`s explicit unless there's a clear documented purpose doing otherwise.

Having all that I came up with this version:

```
template< typename T >
class arr_iterator : public
std::iterator<std::forward_iterator_tag, T>
{
public:
  typedef arr_iterator tSelf;
  // my usual typedef
// construction
  arr_iterator() : mPtr( 0 ) {}
  explicit arr_iterator(T* p) : mPtr( p ) {}
// Comparison
  bool operator==( const tSelf& rhs ) const
    { return mPtr == rhs.mPtr; }
  bool operator!=( const tSelf& rhs ) const
    { return !(*this == rhs); }
// iterator operations
// could use reference_type and pointer_type as
// return
  T& operator*() const { return *mPtr; }
  T* operator->()  const { return  mPtr; }
  tSelf & operator++() { ++mPtr; return *this;
  }
  tSelf operator++(int) { tSelf tmp(*this);
    ++(*this); return tmp; }
private:
  T* mPtr;
};
```

It almost compiles with the provided test framework: inside `main` the copy-construction of variables begin and end does not fit with explicit, it shall be converted to direct-initialisation; or if such use is actually intended explicit shall go.

From Michael Brunton-Spall

mib@mibgames.co.uk

At first glace most of these issues seem to be syntactical issues, which I'll deal with first. The first compile fails because of the `std::for_each`. I look and realise that your first issue is that things like `std::for_each` normally require an iterator, not a pointer to a function that returns an iterator. I see you looking at me blankly so I'll show you what I mean. You wrote this `std::for_each( t.begin, t.end, print);` you actually meant to write the following:

```
std::for_each( t.begin(), t.end(), print);
```

That's quite a common mistake. I can only assume here that you weren't trying to do anything funny or special, and that is what you really meant. I also notice that your call to `incr` in the `for each` is also missing the parenthesis. So I add those and uncomment the first line marked `ERR`. Now I try to recompile your program and I get a different error, it looks like this :

```
/usr/lib/gcc/i486-linux-gnu/4.0.3/../../../../
include/c++/4.0.3/bits/stl_algo.h: In function
'_Function std::for_each(_InputIterator,
_InputIterator, _Function) [with _InputIterator
= iterator<int>, _Function = void (*)(int&)]'
num39.cpp:81:   instantiated from here
```

```
/usr/lib/gcc/i486-linux-gnu/4.0.3/../../../../
include/c++/4.0.3/bits/stl_algo.h:158: error:
invalid initialization of reference of type
'int&' from expression of type 'const int'
```

Now this looks complicated, but it's actually telling you exactly what the problem is. Your print function takes a `const int&`, and `incr` takes an `int&`. It looks like the reference being returned by the iterator is a `const` value, which works fine for print, but won't work for `incr`. When I realised that the iterator had an issue, I looked at the iterator `operator*`, which you define as `// iterator operations`:

```
T& operator*()
{
  return *mPtr;
}
```

This looks fine, however looking in that area of code, I noticed the hanging `const` at the end of the previous function. Ah ha, this looks like you meant to say that `operator!=` is a `const` function but weren't sure where to put the `const`. The cost for a `const` function goes after the declaration but before the definition. Which is to say before the bit in the parenthesis `{}`. The compiler, confused by your hanging `const` is attaching it to the next block, which is the return value for iterator changing it to `const T&` from `T&`. Removing the `const` and putting it in the correct place seems to fix your code, and does what I believe to be the correct thing.

My final `main` looks like this:

```
int main()
{
    tester t;
    std::for_each( t.begin(), t.end(), incr );
    iterator<int> begin = t.start();
    iterator<int> end = t.start() + t.size();
    begin++;
    std::for_each( t.begin(), t.end(), print);
    std::for_each( begin, end, incr);
}
```

and the only change further up is to change the iterator `operator!=` definition like this...

```
bool operator!=( const iterator& rhs ) const {
  return mPtr != rhs.mPtr;
}
// iterator operations
T& operator*()
{
  return *mPtr;
}
```

Now on a stylistic point of view, in *The C++ Standard Library* by Nicolai Josuttis, the recommendation is that you should write an iterator by inheriting from the `std::iterator` template, passing in the iterator type tag and type as necessary.

The reason for doing this is that to create an STL compliant iterator, you need to provide a number of standard type definitions. Instead of doing these by hand, inheriting from `std::iterator` will provide these for you based on the template parameters.

From Nevin Liber

nevin@eviloverlord.com

Why create an iterator class at all? You already have a perfectly good iterator: `int*`. Pointers are iterators, and can be used in the STL algorithms (some implementations of `std::vector` use pointers as the iterator type, for example.) Unless you need to either restrict or extend the functionality, just use a pointer.

It is a lot of work to create an iterator that matches the STL requirements. It is not for the faint of heart.

Heck, most of `tester` can be replaced with a `std::vector<int>` or, better yet (if you have access to it), a `boost::array<int, 10>` (or a `std::tr1::array<int, 10>`, which is essentially the same thing).

There is a bug in `tester::tester()`: `data[i++] == i` has undefined behaviour, since you don't know when the increment of `i` will happen.

Taking all this into account, using `std::vector`:

```
#include <vector>
class vtester
{
  typedef std::vector<int> container_type;
  container_type data;
public:
  typedef container_type::iterator iterator;
  typedef container_type::pointer  pointer;
  typedef container_type::size_type  size_type;
  typedef container_type::value_type value_type;
  vtester() { for (value_type i = 0; 10 != i;
    ++i) data.push_back(i); }
  pointer start()        { return &data[0]; }
  size_type size() const {return data.size();}
  iterator begin()    { return data.begin(); }
  iterator end()      { return data.end(); }
};
```

To use a `boost::array` instead, all that needs to change is the container_type and the constructor:

```
#include <boost/array.hpp>
class atester
{
 typedef boost::array<int, 10> container_type;
 // rest as above, changing vtester to atester
```

Now, on to the issues in `main()`:

```
std::for_each(t.begin, t.end, incr);
```

There are two problems with this The first is that `begin` and `end` are member functions of `tester`, and need to be called with parentheses.

```
std::for_each(t.begin(), t.end(), incr);
```

The other issue is because a temporary (caused by dereferencing the iterator deep in `std::for_each`) cannot be bound to a non-`const` reference, yet `incr` expects a non-`const` reference. If you substitute `print` for `incr`, it works fine. No easy fix; an ugly one is with a `const_cast`, as in:

```
void incr(const int& i) {
  ++const_cast<int&>(i); }
```

but I wouldn't recommend it.

```
iterator<int> begin = t.start();
iterator<int> end = t.start() + t.size();
```

`t.start()` returns an `int*`. If we use your class, `int*` is implicitly convertible to an `iterator<int>` (via the constructor), in general that conversion does not exist for iterators. A possible fix:

```
int* begin = t.start();
int* end = t.start() + t.size();
begin++;
```

Generally, preincrement is preferred to postincrement, as it is never slower and usually faster (since it doesn't have to produce a temporary).

```
++begin;
```

Looking back at your iterator class, **iterator operator++()** should be really be declared as

```
iterator& operator++() {++mPtr; return *this;}
```

as those are the expected semantics.

If you really want to make your iterator class compliant with the STL, it would end up looking something like:

```
template<typename T>
class iterator : public
std::iterator<std::input_iterator_tag, T, int,
 T*, T&>
{
public:
  typedef std::input_iterator_tag
iterator_category;
  typedef T value_type;
  typedef int difference_type;
  typedef T* pointer;
  typedef T& reference;
  iterator(T* p) : mPtr(p) {}
 bool operator==(const iterator& rhs) const {
    return mPtr == rhs.mPtr; }
  bool operator!=(const iterator& rhs) const {
    return !operator==(rhs); }
 iterator& operator++() { ++mPtr; return
    *this; }
  iterator operator++(int) { iterator
    tmp(*this); operator++(); return tmp; }
 T& operator*() const { return *mPtr; }
private:
  T* mPtr;
};
```

From Seyed H. HAERI (Hossein)

shhaeri@math.sharif.edu

Scanning the graphical layout, I would say, yeap, good horizontal indenting. I wonder if the lack of vertical indenting is due to shortage in (C Vu) space…

Considering the error lines the programmer has reported, I'd say that one of them was trivial to me right at the first glance, and for the other, I would like to share my experience toward demystifying it with the others. As soon as I do this, I will come to the guidelines I may want to offer the student.

The first ERR line is:

```
std::for_each(t.begin, t.end, incr);
```

The trivial mistake I spoke about is here. A common mistake I always hint my students on: **begin** and **end** are both member functions, and not data members. This means that they should be called upon use. Like this:

```
std::for_each(t.begin(), t.end(), incr);
```

The second ERR line was not this easy to me to decipher. I couldn't spot that with merely reading the code. However, at the very beginning even, the following line seemed quite odd to me:

```
bool operator!=( const iterator& rhs )
{ return mPtr != rhs.mPtr; } const
```

I first thought this should be an error. However, when I typed the program to make sure, I didn't get any error messages on compilation. So, I started digging up the Standard to see if I'm making a mistake, and – oddly enough – it is that the Standard says this is also a correct way of **const**-qualifying member functions. No surprise that I couldn't find any such thing. [Roger: correct – it isn't there]

Then, I started to uncomment the second ERR line. I ended up with this error message (GGC 3.3.1):

```
In function `_Function
std::for_each(_InputIter, _InputIter,
_Function) [with _InputIter= iterator<int>,
_Function = void (*)(int&)]': instantiated
from here could not convert
`(&__first)->iterator<T>::operator*() [with T
= int]()' to `int&'
```

This was really odd to me because as you can see

```
T& operator*() { return *mPtr; }
```

defines its return type to **T&** which, given that we've instantiated **iterator<>** with **int**, means **int&**. I then noted that the only difference between **print** and **incr** is that the former takes a reference **const**. So, to make sure that the problem is on the return type instead of the member function itself, I tried:

```
int& ir = *begin;
```

Odder even, I got this:

```
69: error: could not convert `(&begin)-
>iterator<T>::operator*() [with T = int]()' to
`int&'
In function `_Function
std::for_each(_InputIter, _InputIter,
_Function) [with _InputIter = iterator<int>,
_Function = void (*)(int&)]':
64: instantiated from here could not convert
`(&__first)->iterator<T>::operator*() [with T
= int]()' to `int&'
```

I wondered then that: "if this stupid compiler is right, and **int&** is not the type of what **operator*** returns, then what that type is?" To understand that, I tried forcing the compiler to call the **operator*** with the syntax I want. Here it is:

```
typedef int& (iterator<int>::*Op) ();
Op op = &iterator<int>::operator*;
int& ir = begin.op();
```

And, got:

```
70: error: cannot convert `const
int&(iterator<int>::*)()' to
`int&(iterator<int>::*)()' in initialization
```

Therefore, I realised that the compiler was assuming the return type of the only **operator*** was **const int&** as opposed to **int&**. Trying to find out why, I first guessed that this may be inheritance of **iterator<>**.

`iterator<>` however doesn't inherit from anything. So, I then commented the only `operator*` I could see. I ended up seeing that the compiler couldn't find any other instance. Afterwards, I wondered what if I disambiguate things? And, it actually worked! I tried adding the following operator overloading (which is the only rational `operator*` one may choose for returning a `const`):

```
const T& operator*() const { return *mPtr; }
```

And, the error message I received then was:

```
70: error: no matches converting function
'operator*' to type 'int&(class
iterator<int>::*)()'
22: error: candidates are: const T&
iterator<T>::operator*() [with T = int]
23: error:                 const T&
iterator<T>::operator*() const [with T = int]
```

I soon realised that the only thing which differs across the two overloads is that the latter is qualified with `const`. Neglecting that, one would observe that both the versions return `const T&`'s. So, I started double-checking the return type of the first overload. It was right here that I understood that the only thing which was causing me this bother was that nasty `const` after the definition of `operator !=`! C++ does not care about the white spaces, OK? So, what would it do with that `const`? Yes, it would patch that to its next statement which is `T& operator*()`. What this would mean is that it would consider the return type of `operator*` to be `const T&`, and not `T&`. Yuck! This is the whole senseless bother!

Back to the critique, and scanning top-down, I first see that the student has placed the assignment operator in the constructors. Maybe this has been mistakenly considering this the `copy` assignment constructor, and trying to implement The Rule of Three. However, I can't see any `copy` constructors, and this means violation of the above rule. And, by the way, the first constructor seems to be better to make explicit. (Note that this depends on the application, and can't be decreed on with our current information.) So, one negative mark in appropriate commenting.

Then I see that for `iterator<>`, the student has offered `operator !=`, but hasn't done that for it counterpart – `operator==`. Not good enough. The same holds for `operator*` vs `operator->`. If the programmer has admittedly deleted the above (missed) member functions to shorten the demonstration, then the student is perhaps smart enough (in C++) to realise that the implementation of `std::for_each` may have used either of the prefix or postfix versions of `operator++`. I will have a little surprise if this programmer who doesn't spot the mistake in the first `ERR` line, would know enough about STL to decide on the latter upon their knowledge that `std::for_each` needs InputIterators, and InputIterators are needed to provide both versions.

Next, I come to `tester`. For the case of the `for` loop in the constructor, stylistically, I would like to recommend him to write `i != size()` instead of what has been written. This will enable the student to avoid a common trap whilst looping using iterators. Changing the incrementing of the loop to prefix, not only will save a temporary, but also will cause the loop to read more clearly.

As I reach to `start()`, I can't understand why the member function has been written. I don't find any acceptable reason, yet I find several reasons why this is a bad decision. Firstly, all the STL containers have `begin()`, but none has `start()`. So, caring consistency only, I would suggest to remove that. This is not the only reason I suggest it. The way to use `start()` is no different from how one could have been using `begin()`. So, secondly, there is no point in retaining that. Thirdly, he/she is providing `iterator<>` in resemblance to the Standard. One reason why the Standard provides iterators is to keep the user away from the internal type really used for the storage of the containers. Yet, `start()` is returning something of the internal "unguarded" type.

Finally, I would recommend changing the two remained postfix `operator++`'s to prefix because there semantically is no difference in either case.

From Simon Sebright

simon.sebright@ubs.com

I sat down to continue reading my C Vu and read some of the answers to the previous competition. I feel, like Balog Pal that the format is difficult. My personal take is that there is no point going into the infinite details of "correcting" the student's code if the design is like mud. I suppose the question is what approach will most benefit the student, and that depends on the student, and their relationship with the teacher. So, this critique will firstly address the student's specific question, then go on to take a step back and work out what is actually required here.

So, why won't this compile:

```
std::for_each( begin, end, incr );
```

when this does:

```
std::for_each( begin, end, print );
```

So what is the difference? The difference is that the functions `incr()` and `print()` have different signatures, with `incr()` taking a non-`const` reference, `print()` a `const` reference. According to my current source of documentation (MSDN Jan 06), `std::for_each()` expects as its third argument a function/functor, and "This function does not modify any elements in the sequence", i.e. it takes its parameter as a `const` reference.

What does this mean? It means that we can't use `for_each()` to change the elements in the container. `incr()` was trying to modify the contents, namely by incrementing each integer in the array. Why this is not possible might seem rather odd. I think it is to do with the fact that changing the contents of a container may invalidate certain properties of it. For example, if it is a sorted container, then the resultant sorting cannot be guaranteed. Such are the issues when dealing in very generic terms. Personally, I feel it should be possible to use an algorithm to modify a container's contents. In fact, another part of my documentation says "The algorithm `for_each` is very flexible, allowing the modification of each element within a range in different, user-specified ways". Rather confusing!

Let's leave that one for now by saying that there probably is a `boost` algorithm, or we could write our own to make changes to elements.

So, on to the more important part, I think. What should the student have done to get his class to work with stl? Firstly, stl is a large library, so what does s/he mean? It looks as though, very diligently, they noticed that the class was acting like a container. They also know that containers are used in conjunction with iterators, so that they can be exposed to common algorithms. Great!

But, our student seems to have dived in and just decided to write an iterator. Without thinking how it will be used in proper detail.

I'm currently reading *Test-Driven Development*, so thought I could write the rest in that style, that being to first define a small test, see it fail and add some code to get it to pass. Refactor if appropriate, and if the test passes, do it all again. But I won't use an actual test harness, rather rely on the code being compilable and watching the console output.

Right, the student wants to be able to use algorithms with the container class. Specifically `for_each()`. As Kent would say, that's a big leap. Let's start with the simplest step on the way. `for_each()` is going to need two inputs from the container, so here is one possibility:

Main:

```
Tester t;
t.begin();
t.end();
```

Tester (public):

```
void begin();
void end();
```

So, we have compiling code. A good start, better than our student, except that it doesn't do anything yet. The next test case has got me a little stumped. We cannot pass void to **for_each()**, so we have to make a decision about what to return pretty quickly. Here, we will follow the other containers' examples. That is that the container contains a nested type, iterator. So, the test code becomes:

Main:

```
Tester::iterator start = t.begin();
Tester::iterator end = t.end();
```

To get that to build, we have to provide an iterator type in **Tester**. We can either **typedef** this, or create a new type. We'll go with the latter for now, as we can add only what is strictly necessary to it as we go, and refactor later.

Tester (public):

```
class iterator {};
iterator begin() { return iterator(); }
iterator end() { return iterator(); }
```

Now, we have to take a bit more of a leap, and put in the **for_each()** call in **main**:

```
std::for_each( begin, end, func );
```

This now requires a **func**:

Global:

```
void func( const int & i ) {}
```

I got rid of the compiler errors by adding to the nested iterator type, so that **Tester::iterator** becomes:

```
class iterator
{
public:
  int operator*() { return 1; }
  bool operator != (const iterator& other)
   const { return false; }
  iterator operator++ () { return *this; }
};
```

This compiles, but obviously does nothing yet. Let's make **func()** output something so that we can see what is going on, now that we are going to proceed in the absence of a test harness.

```
void func( const int& i ) {
  std::cout << i << std::endl; }
```

Running this now reveals nothing, as the algorithm terminates immediately due to the implementation of **operator !=();**

OK, what next? Well, my **operator*()** returns an **int**, the type referenced in the container, so we'll have to add a bit more to the implementations, to **int**ify it. **Tester** contains an array of **int**s, so we'll make the iterator iterate over such an array, which is what the student did in the iterator class. Here is the minimum we need (for my stl implementation):

```
class iterator
{
public:
  iterator( int* p ) : m_p(p) {}
  int operator*() { return *m_p; }
  bool operator != (const iterator& other)
   const { return m_p != other.m_p; }
  iterator& operator++ () { ++m_p;
   return *this; }
private:
  int* m_p;
};
```

The **Tester** functions now become:

```
iterator begin() { return iterator( data ); }
iterator end() {
 return iterator( data + size );
}
```

Running this in a console window, we see the printout of the initialised array of integers.

Hooray! We have finished. Or have we? Our student's iterator class is more rich that mine. First, it's a template class. Why? So it can iterator over an array of any type. I have made mine a nested type of **Tester**, so there is no point in it exposing more than **Tester** needs. If **Tester** becomes a templated type, then so should its iterator. That is for another day.

Second, the student has got more functions in their iterator. They seem to be sensible, and I am sure that trying my example on other algorithms would produce a need for some of them.

But, if you look at the iterator, its member functions do nothing other than delegate to the **m_p** pointer member. So, we can replace **Tester::iterator** with:

Tester:

```
typedef int* iterator;
```

The code compiles and gives the same output.

Now, how about that increment function? Well, we could use something like **std::transform()**:

```
int incr( const int& i ) { return i + 1; }
```

and we can add to **main()**:

```
std::transform( start, end, start, incr );
std::for_each( start, end, func );
```

It is slightly contrived that we use the same range for the source and destination, but when using a library, you have to roll with what it provides. Running this gives us a second output of incremented integers. The finished code is:

```
#include <algorithm>
#include <iostream>

class Tester
{

public:
  typedef int* iterator;
  iterator begin() { return iterator( data ); }
  iterator end() { return iterator(
   data + size ); }
  Tester()
  {
  for ( int i = 0; i < size; ++i )
      data[i] = i + 1;
  }

private:
  static const int size = 10;
  int data[size];
};

void func( const int& i ) {
  std::cout << i << std::endl; }
int incr( const int& i ) { return i + 1; }
```

```
int _tmain(int argc, _TCHAR* argv[])
{
  Tester t;
  Tester::iterator start = t.begin();
  Tester::iterator end = t.end();
  std::for_each( start, end, func );
  std::transform( start, end, start, incr );
  std::for_each( start, end, func );
  return 0;
}
```

Commentary

At the ACCU conference panel about "writing for ACCU publications" several of those attending asked to see the student code critique earlier than the current schedule to give more time to compose a reply. I'm not sure making it *too* early is good thing but (starting with this issue) I posted the critique question onto accu-general to give an extra month of thinking time.

Can I also encourage anyone finding suitable candidate code to send it to me for possible inclusion!

The entries cover a variety of approaches to the student's problems. These are at two levels – syntax problems, such as the trailing **const** and the undefined behaviour of the **for** loop in the tester constructor; and the harder problem of writing an STL-like iterator.

The code first took my attention because of the trailing **const**, which modifies the return type of the *next* method as the white-space is not significant to the compiler. This sort of problem is hard to see, especially for new users of C++, and the compiler errors rarely help.

My preference for the STL-specific part is to recommend that the student gets more practice at *using* the STL before they try to *extend* it. Nevin's answer hints at this: "Why create an iterator class at all?" It seems to be a common mistake to assume that learning the STL involves writing lots of template classes. Using the existing classes and functions provides the best initial learning, and when, later on, they reach the point of writing their own iterators they will have a clearer idea of how to do this (and more experience of decoding the somewhat scary error messages typically generated by the compilers).

At this point Matt Austern's book *Generic Programming and the STL* might be useful; or more specifically pointing the student at boost's iterator façade and adaptor classes (which are also in the first C++ technical report).

The Winner of SCC 39

The editor's choice is:

Balog Pal, with a commendation to Simon Seabright.

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

Student Code Critique 40

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

The student wrote:

"I'm having problems printing out a tree data structure – the code doesn't crash and I don't get any compiler warnings but the tree doesn't seem to get shown properly. It's based on some (working) Java code. This test program shown below only prints the head node and not the children. I've tried both C and C++ but the program behaves the same for both."

Please try to help the student understand *why* their code is broken as much as *where* it is broken.

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

struct binaryNode
{
  int value;
  struct binaryNode *left;
  struct binaryNode *right;
};

struct binaryNode createNode( int value )
{
  struct binaryNode *newNode;

  newNode = (struct binaryNode*)malloc( sizeof(
   struct binaryNode ) );

  newNode->value = value;
  newNode->left = 0;
  newNode->right = 0;

 return *newNode;
}

struct binaryNode addChildNode(
 struct binaryNode parent, int value )
{
  struct binaryNode tempNode;
  tempNode = createNode( value );

 if ( value < parent.value )
    parent.left = &tempNode;
  else
    parent.right = &tempNode;

  return tempNode;
}

void printNodes( struct binaryNode head,
 int indent )
{
  int i;
  if ( head.left != 0 )
    printNodes( *head.left, indent + 1 );
  for ( i = 0; i < indent; i++ )
    printf( " " );
  printf( "%i\n", head.value );
  if ( head.right != 0 )
    printNodes( *head.right, indent + 1 );
}

int main()
{
  struct binaryNode head;
  struct binaryNode node1;
  struct binaryNode node2;

  head = createNode( 3 );
  node1 = addChildNode( head, 1 );
  node2 = addChildNode( node1, 2 );

  printNodes( head, 0 ); // Only prints 'head'

  return 0;
}
```

Prizes provided by Blackwells Bookshops & Addison-Wesley

Bookcase

The latest roundup of book reviews.

In the recent past, I've had to take the somewhat unfortunate position to drop book reviews in order to keep to the page count and balance the number of articles to the number of reviews. In this instalment, I should have made a large chunk out of the back-list I have sitting on my hard drive. Thanks for being so patient.

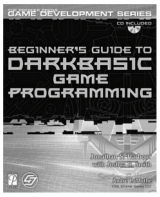
Remember, if you submit a book review you are contributing to the greater knowledge of the membership. Books are expensive and the last thing anyone wants it to spend upwards of 30 pounds on a book which is an utter turkey!

That said, if you decide to review a book, the worst that will happen is you lose a fiver – and if the book has the "Not Recommended" rating, your next book is free. What can be fairer than that.

As always, the ACCU must thank the Computer Bookshop, Blackwells and a range of other publishers for providing us with the review books.

Games Programming

Beginner's Guide to DarkBasic Game Programming



by Jonathan S. Harbour, ISBN 1-59200-009-6, 711pp + CD , Prima Tech

reviewed by Mark Green

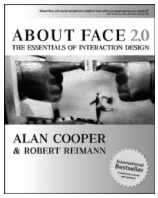
The premise behind DarkBASIC is to give, in a BASIC like language, the power that is needed to access DirectX (easily) and so make games. It is obviously tied to the Microsoft Windows world and in my opinion the amateur/hobbyist games market. The book has three main sections: "The Basics of Computer Programming", "Game Fundamentals: Graphics, Sound, Input Devices, and File Access" and "Advanced Topics: 3D Graphics and Multiplayer Programming".

As the title of the book suggests it is a "Beginners Guide", it does try to handhold the reader as the print is large and there are lots of pictures. Even so, the first part of the book is a strange mix it introduces all the basic procedural programming topics, in examples. But it does not make the user think and extend the work, just follow along the exact path given, it does not promote true understanding of how to link these ideas together into a game (or any program).

Part two could be subtitled the functions called in DarkBASIC which are used to access the different parts of DirectX, (not very snappy subtitle but part two is not very snappy either at 300 pages or so). It is a fair reference and it does show the power and extent of DarkBASIC, but it does not tie all the bits together into anything but little noddy gamelets. It does not cover any real game design, story progression, production process (design, coding, testing). True this is only one book but it does not really hint at all the other parts that go into the creation of a game. The final part quickly looks at the advanced topics of 3D and multiple payers but only in an introductory way.

My verdict, it is as the title states a BEGINNER's guide to DarkBASIC, only those people who have little or no (or very rusty) procedural programming experience will find this book useful. True it will give them a leg up into DarkBASIC programming, but they still have lots to learn about programming and even more about developing games and the games industry. Overall an average to weak rating.

About Face 2.0; The Essentials of Interaction Design



by Alan Cooper and Robert Reimann, ISBN 0-7645-2641-3, 540pp, Wiley

reviewed by Christopher Hill

I think back to a programming conference in Cambridge in the mid 1990s. I had planned to attend one session by Alan Cooper and spend the rest of the day doing more programming things. I spent the rest of the day in Alan's sessions. It was all a revelation! He was describing problems that people have when using software in everyday terms, while explaining the underlying problem.

It is like putting a handle on the side of a door that you need to push. It is not the end of the world. When you first come to that side of the door you pull the handle (you don't push handles – well I don't). The door does not give, so you push. Next time you know to push the door. But sometimes the handle on the wrong side still catches you out.

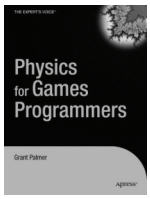
OK you can complete your trip with just a little delay, but by jolting the door, your journey has been disturbed. People are at their most effective when they enter a state of 'flow' – everything just works, you know where you are going and what you are going to do. Then you hit a menu option that you have to think about or does not do what you expected. You lose flow and you become less effective.

About Face highlights the issues that 'jolt' when using computers. You write in a notebook and place it back on the shelf. When was the last time it asked you if you want to keep your jottings? Yet when I finish writing this review, the software will ask if I want to save the changes? This is putting a programmer's problem (reconciling copies on hard disk and in memory) into the users' domain – where it does not belong!

If you have the first edition, I would encourage you to also get the second edition of this book. User Interface Design is a moving target, and more issues are addressed. The second edition has more psychology of interaction and covers acquiring requirements (ideas like talking to people in their place of work, maybe while they are doing the job).

The second edition is a major re-write, very informative and challenging. It ought to be read by any one who makes any claim to design user interfaces. Highly recommended.

Physics for Game Programmers



by Grant Palmer, Apress, ISBN: 1-59059-472-X, 472pp

Reviewed by Carl Bateman

This book tries to cover everything: from basic kinematics through ballistics to sport simulation, aerodynamics and lasers (even integration and differentiation are covered) and all in 472 pages.

Clearly the author knows his stuff and has a pleasant conversational style that is quite engaging. Unfortunately he covers a lot of material, very quickly, perhaps too quickly. If the reader can't follow the explanation she's lost

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 us know). 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

and will need to refer elsewhere to understand the topic being covered.

There are demo applications throughout the book, these are self contained but very primitive. While they usually illustrate the principle in question they fail to engage the user for more than a few minutes. It may have been better to build a project or two, integrating the various principles covered.

There are brief digressions, a biography of Newton for example, which add little. Given the intensity and sharp learning curve of the book these pages would have been better spent focusing on the topic at hand.

Despite proclaiming itself to be "Physics for Game Programmers", the principle language used here is Java (C and C# versions are available via download). The majority of modern games are written in C++ making this a strange choice. Java is used for mobile games, but the state of that particular art leaves little room for painstakingly accurate physical effects.

The book exhorts the benefits of using physics to improve the fidelity of games and thus their marketability, and does so to the point where one would be forgiven for thinking that all games would benefit mightily from the introduction of accurately modelled physics. However, this view fails to take into account games that have no need of physics (e.g. chess) or games that distort or ignore physical laws (e.g. Mario).

Certainly, there are many games where accurate physics are desirable, flight sims, racing games and the like, but frequently accurate physics are not needed or wanted because it adds unnecessary complexity or, ironically, because the effects they create don't look realistic. Unfortunately, none of these potential issues is mentioned, but then there is scarcely room to do so.

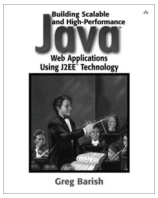
This is not a bad book, it is simply over-ambitious. It would certainly have value for someone with a basic familiarity of the topics covered and who wished to refresh their memory, but seems too terse to be a first learning guide.

Java

Building scalable and high-performance Java web applications using J2EE technology

by Greg Barish, Addison-Wesley, ISBN: 0-201-72956-3, 392pp

reviewed by Michel Greve



According to the author, the book is written for "any engineer or architect who is proficient with Java and wants to build Java-based Web applications for performance and scalability, but does not yet understand how J2EE can be used toward the goal or how all of its underlying technologies work."

The book has the following goals:

- To define and identify the challenges;
- To provide you with a J2EE roadmap;
- To describe concisely key J2EE technologies;
- To fill in the gaps of Web application design that the J2EE spec. leaves out;
- To demonstrate the benefits of various specific J2EE design decisions.

The book is well laid out, though the chapters are a bit long: about 30 pages. In the beginning I had some trouble reading it, though I think the problem was the font. The book has a lot of short listings and though I didn't read all the listings and still understood what the author meant. The listings are explaining at a low level what the author has written e.g. the author talks about beans and the listings show you how the bean looks like in Java.

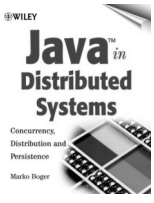
The author not only explains how to build scalable and high-performance web applications, he also explains how a web application works internally. He explains HTTP, beans, request processing, messaging, database design and architectures of web applications. The bottlenecks in a web application are explained and possible solutions are given (it depends on the problem, which solution will work with a given trade-off). The index of the book is good.

The things I don't like about the book are the font and the rather large chapters. The good points are the explanations of how a web application works and the optimizations with their trade-off. Unfortunately, if you're only interested in learning how to optimize a web application this isn't the book for you. This is more a book for somebody who wants to know how a J2EE web application works and how to optimize it. For that kind of reader I would give it a recommended.

Java in distributed systems

by Marko Boger, ISBN: 0-471-49838-6, 393pp, John Wiley & Sons

reviewed by Michel Greve



This book is about distributed systems in Java. The book doesn't include a CD, but it refers to the different websites where the necessary software can be downloaded. The book is aimed at developers who want to deepen their knowledge of Java in distributed systems.

The book has a nice layout and reads well. The text is clear and the listings are short and contains all the exception handling, but have no comments. I like my listings in a monospace font. Unfortunately that is not the case in the book.

There are a number of subjects that reappear everywhere in the book: concurrency, distribution, persistence, communication and method calls.

The book is divided in two parts:

- Java in distributed system
- A distributed Java

The first part is an overview of a number of solutions in distributed systems. The subjects are not only the well known solutions like: Corba, RMI, Jini etc, but also about the Voyager framework, Pjama and TupleSpaces.

The second part of the book is a possible solution to the problems the author mentions in the book. The author directed a research project on the Hamburg University. The university developed a framework called Dejay. This framework is explained in this second part.

In the last chapter the author gives three examples that use the Dejay framework; distribution, concurrency and persistence. These examples and explanations are short and to the point.

At last we have a good index.

The things I dislike about the book are de monospaced fonts for the code and that the book is a bit dated.

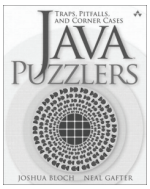
The good things about the book are the overview of the problems and solutions in distributed systems and the very good (high level) explanation of the rather technical and difficult topics. Personally I am amazed what was possible five years ago. Although you may not want to use the frameworks used in this book, you will get a feeling of what is possible nowadays.

Because this book is a little bit dated (1999 written, 2001 the English translation), I give this book a RECOMMENDED.

Java Puzzlers: Traps, pitfalls, and corner cases

by Joshua Bloch and Neal Gafter, ISBN 0-321-33678-X

reviewed by Derek Jones



If, like me, you enjoy figuring out the behaviour of code snippets that involve programming language corner cases then you will enjoy this book. The 95 puzzles are simply expressed and the solutions written in an easy to comprehend way without being overly simplistic. You don't need to be a Java expert to enjoy this book and you will probably learn something from many of the puzzles.

The authors have tried to frame coding guidelines on the back of the surprising behaviour of the some of the puzzles. Unfortunately little thought has been given to these guidelines, which mostly say something along the lines of "avoid this usage". Fortunately the guidelines does not form a substantial amount of the material and can be ignored without distracting from the main contents.

For some puzzles the authors have reused the same problem in a slightly different way, or resorted to relying on (lack of) knowledge of the behaviour of not-so-common library functions.

There are enough interesting puzzles that these cases don't detract from the enjoyment.

Coming from a C/C++ background, I was surprised by how often the edge case behaviour differed in Java. It is a pity that the authors did not include any discussion on how other languages deal with the corner cases.

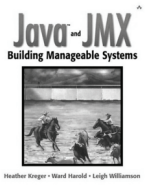
The material also includes quiet a few black and white visual illusions scattered among the puzzles. Some people might find these add to the charm, I found them annoying (although I found the appendix notes on the various illusions very interesting).

Taking the Amazon discounted price gives a cost of around 20p per puzzle. Well worth the price.

Java and JMX programming

by Heather Kreger et al, ISBN 0-672-32408-3, Addison Wesley

reviewed by James Roberts



This book provides a great deal of interesting description of the support for JMX with Java. It covers a great deal of ground from the motivations of using JMX through to details of how to produce manageable applications using Java. The writing style is clear throughout (although I found the bibliographic references a bit over the top, no harm done was done in including them) and is logically organised, introducing the generalities first before covering implementation details.

There were a few things that I did not like. Firstly, there was rather too much quoted code. 10 pages of largely uncommented code is over the top from my point of view. As the book's web-site includes the source code, this seems completely unnecessary. I would have much rather had some more focussed code examples to read, and then used the downloaded code to fill in the gaps for a complete implementation.

The book is somewhat implementation-neutral. I felt that there is a gap in the presentation, as it would have helped my understanding if the book covered an example of an implementation of a JMX solution. The examples on the web-site describe the implementation based on the TMX4J product, which is unfortunately not free – this will limit the number of people that could follow the examples.

Overall I felt that it was a reasonable introduction to the subject, and also will be a useful reference book – but could have been improved.

User Inferface

Interaction Design for Problem Solving

by Barbara Mirel, ISBN 1-55860-831-1, Morgan Kaufmann

reviewed by Francis Glassborow

This book is aimed at the member of your software development team who is responsible for the Human-Computer Interface. Many applications are seriously damaged by the lack of expertise in this area. The importance of well-designed HCIs extends across the entire range of computer applications. It is all very well having highly qualified people working on safety critical, high integrity or mission critical software but a poor HCI design can undo all their careful work. For example, a medical program that correctly calculates the drug dosage required in the treatment of a patient is a menace if the result of poor interface design is that the nurse misreads the screen and give the patient the wrong dose.

The reason that I raise this issue is because I have seen far too many applications where the quality of the programming has been made irrelevant by the poor quality of the interaction between the user and the software.

In this book Barbara Mirel tackles all aspects of the way software interacts with the user. If your team already has someone responsible for the HCI of your products, encourage them to read this book and discuss the resulting issues with you. In the context of a team it might also be helpful to make this book a team study effort so that you all better understand those extra constraints on the total product that are necessary to convert excellent software into excellent usable software.

You will get some idea of the detail with which the author tackles her task when you realise that the final section of the book (a single chapter) deals with the politics of getting the earlier chapters accepted and implemented in a work environment.

In the earlier chapters the reader will learn about appropriate mechanisms for data capture, suitable ways to display data (the author is an expert on information visualisation techniques) and how to help the user to use an application correctly. Years of experience with computers often lead software developers into a state where they have an implicit understanding of overly complex displays. Just as we should be careful to test software on typical hardware rather than some bleeding edge computer we are using for development (you should be so lucky) we should also be careful to test our software on the typical user. However one problem here is that the typical user is often unable to articulate what it is about the HCI that is proving unhelpful. Worse, when such comments are made the tester often explains how the software is intended to be used, thereby making the test subject more knowledgeable than the typical user. I wonder if we should be using double blind testing of interfaces.

Despite the quality of the presentation provided by the publisher, this book will not be a quick study. Nonetheless I believe that either this book or one like it should be on the reading list of most aspiring software developers. This book has a strong theoretical basis but focuses very much on the practical needs of the commercial practitioner.

User Interface Design

by Jenny Le Peuple & Robert Scane, ISBN 1 903337 194, Crucial

Visual Programming

by David J. Leigh, ISBN 1 903337 11 9, Crucial

reviewed by Francis Glassborow

I am choosing to review these two books together because my main focus concerns presentation and editorial issues that are, in my opinion, serious flaws in both volumes. They highlight the very real risk that modern printing technology is in danger of supporting the under prepared and poor presentation that already mars a great deal of web-based publishing.

My printer could turn out these 'perfect bound' books with full colour laminated covers with black and white contents for under £4.00 for a print run of a single book (I know, because I asked them). Just because it can be done does not mean that it should be done. There is a great deal more to producing a marketable book than simply being able to print and bind it.

No competent production editor would allow such poorly laid out material to leave their machine. Actually, even the average amateur might shudder at pages that seem to be devoid of the concept of a bottom margin. Given a modern DTP package and someone with even a basic understanding of presentation and we would have something better than that which might be found in your typical parish magazine.

An editor is part of the production chain for a book. One of the jobs of an editor is to help and persuade inexperienced authors (and even experienced ones) to present their technical knowledge in a readable form. It is my opinion that this has not been done for these books. In both cases the text has a feel of a first draft. The content is fair, and good enough to justify publication but the authors have not been persuaded to put in the hours needed to turn good ideas into well-executed books.

I suspect that the new generation of equipment that allows printers to produce entire books as a single integrated process is going to increase the prevalence of books like these. That is sad because the new technology has the potential for producing something so much better. Laying out the text for a hundred and fifty page book is not much more arduous than laying out a couple of issues of C Vu and Overload. Modern computers and communications coupled with modern printing and binding equipment should allow many more people to earn an income working from home on many of the stages of book preparation and publication. All that should be achievable without any loss of quality.

Well done, the first of these books would have been a fair alternative to Interaction Design for Complex Problem Solving as an introduction to

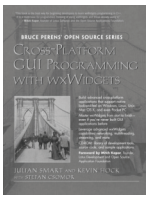
this important subject. However, as is, the fact that the latter book is two and a half times the price is more than compensated for by both the quality of the content and the quality of the presentation. On a scale of one to ten I would give Interaction Design for Complex Problem Solving a very good eight, and User Interface Design would get a poor two.

In case you are interested, *User Interface Design* addresses issues of appropriate design of UIs (User Interfaces) without giving much consideration to the many practical problems that result in accidental misuse of software or irritations caused to users. As an introduction it would be acceptable if the writing and presentational issues were addressed. Visual Programming uses Visual Basic to introduce the reader to the style of programming that focuses largely on selection of standard elements from a visual palette.

Cross Platform GUI Programming with wxWidgets

by Smart, Hock and Csomor, Prentice Hall, ISBN 0-13-147381-6, 662pp

reviewed by Paul F. Johnson



Recently, I turned 35 and as a present, I bought a book that I'd only seen in draft form a while back. You may have realised by now that I am not a great fan of any GUI, but the ability to write the code once and let the library do the grunt work is one I completely agree with.

The problem is that there are not that many cross platform GUI libraries out there. Qt is good, but bulky and the licence is not that pleasant – until Qt 4 hit the streets, it was quite clunky. GTK just isn't nice under Windows and System.Windows.Forms isn't mature enough. wxWidgets is different. It is very close (in terms of the event and message system as well as windowing) to MFC and yet is available for a very large number of operating systems.

Up until now, wxWidgets had a problem in that the documentation was not that amazing. Sure, there are a couple of websites with some examples, but nothing much else. This book changes all of that with a detailed explanation of what everything does (complete with plenty of code examples).

It is set out in a very logical style – it starts with the equivalent of "Hello World" and works right the way through the library – and boy! – it is one extensive library!

If you're an MFC programmer who wants to reach a wider audience or someone who wants everyone to see their wares, but doesn't want to go the .NET route and doesn't have the time to go into Qt, then wxWidgets is for you. It is quick to learn, easy to understand the object model and most of all, costs nothing to use.

I can't recommend this book highly enough.

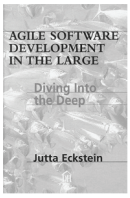
Highly Recommended.

Methodologies

Agile Development in the Large

by Jutta Eckstein, ISBN 0-932633-57-9, 216pp, Dorset House

reviewed by Alan Griffiths



Over the last few years, an increasing number of software development methods have claimed to be "Agile". Probably the best known of these is Kent Beck's "Extreme Programming" [XP] which is like all such methods: focussed on ensuring that all participants in a project (including the "customer") get good feedback quickly and are able to respond to it. This is clearly easier when all participants know each other, and a lot of such methods prescribe putting everyone into the same room. Such approaches do not scale and most methods state that the maximum team size for which they are proven is in the range of 10-20 project members.

There are larger projects, and when projects get larger, the approach to communication and feedback needs to change. To give an example: with several hundred participants finding room for a "daily stand-up meeting" (as prescribed by SCRUM, XP, etc.) is problematic, and the time needed for everyone to give a 30 seconds summary of their activities prohibitive. Consequently, some authors (such as Alistair Cockburn in "Agile Software Development") have speculated that as the size of the project grows then it will start looking more like "traditional" processes whose descriptions focus more on their intermediate work products (e.g. design documents) than on the feedback they support. (IMO it is this focus on solution/work product and not problem/providing feedback that leads to misinterpretation of these methods.)

All this raises the question: what does happen when the Agile priorities are applied to large projects? I, and no doubt others, would like to know! Jutta Eckstein is someone that has been there, done that and has now written a book. So I was delighted when she asked me to review the book and sent me a signed copy.

This book attempts to cover a lot of ground. It includes a summary of the Agile movement and justifications for many of the practices common to Agile (and other) methods. It discusses the differences the organisation will see from "linear" development processes and how to "sell" the approach to different parts of the organisation. In addition, as the title suggests, it covers the problems presented by "large", offers some possible solutions, and gives a report on one large project.

One of the problems of covering a lot of ground is that some of it is not covered very well. For example, while there is much that is valid said about testing, on the different types of testing and who should write each, there is the following statement (about automated testing): "A common and reasonable fear is that testing will slow development down. While this is definitely true at the very start of a project, having tests available will accelerate the development later on, when the system requires some changes."

My experience is that having automated tests in place during development always increases the speed of delivering working software. It is only later that there may be a cost to maintaining both production code and tests (this can be significant if the tests are poorly designed). If this topic were to be covered in more detail, then I would know how to reconcile the above with my experience: does "the very start of the project" refers to a period when there is no working software?

This is typical of much of the earlier part of the book, much that I agree with interspersed with occasional statements that make me want to interrupt the author and ask for clarification, rationale and/or evidence. To be fair, some of this evidence is probably not available – much of the material derives from one large project and extrapolation based on experience of a range of organisations on other projects of varying sizes.

As with many books the value of this one depends upon your starting point and objectives. If you run large projects and are wondering what Agile methods could offer you then this is a great introduction and provides both examples of the differences it might make and pointers to the literature. You could steal the ideas that best suit your situation.

If you are not new to the Agile movement then it is not clear how important some of the material on existing Agile practices is. While I do understand the desire to make the book self-contained, and appreciate that having this material to hand could be useful to the reader, I doubt that anyone contemplating applying Agile methods to large projects would be ignorant of this background. In view of this, placing it as the first section of the book gives it undue prominence. (On the other hand, I keep meeting people that think XP is the only Agile process, so maybe this is for them.)

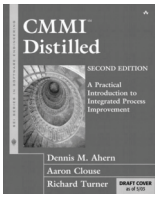
If you are looking for the definitive survey of applying Agile methods to large projects, then it does not exist (yet) and this is not it. It does, however, provide a valuable data point towards such a survey. Every project is different, and every organisation too – we need more examples to map out the territory or even to be sure which are the fixed landmarks. In this book you will find pointers to some of the problems to look for, and some solutions then this will be of help.

Finally, if you are looking for proof that "Agile" can be done for large projects: this is for you.

CMMI Distilled 2ed

by Dennis M. Ahem et al, ISBN 0-321-18613-3, 310pp, Addison-Wesley

reviewed by Greg Billington



This book should be useful to anyone new to process improvement or those with a CMM background wanting to find out more about CMMI. The subject of CMMI is large and complex but the book conveys it in an easy and understandable format while adding more depth and detail as the chapters progress. This makes it good for the novice but also offers enough information and words of wisdom for more experienced readers. The book is split into four main sections that are sensibly organised and can be read sequentially or accessed directly.

The first section gives a short background to process improvement, and shows the roadmap of process evolution from the various standards.

The second section explains the history of the CMMI, how it was created and the two models – staged and continuous. The book gives a clear explanation of the two, highlighting pros and cons which is helpful to someone who comes with the knowledge of the older staged CMM and helps put it into context. The process areas are explained in a concise but clear fashion with good supporting flow diagrams that help the reader understand how these processes fit in the real world.

The third section is shorter than the others and explains reasons for picking the various models. This may be a short section but it conveys a lot of wisdom.

The fourth and final section is a brief look to the future.

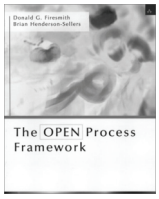
The last third of the book contains appendices showing staged and continuous representation summaries – useful for reference only purposes.

This book certainly does distil a lot of information, and it does it well by focusing on some key essentials, providing comment and advice from the authors in a structured and readable format.

The OPEN Process Framework

by Donald Firesmith & Brian Henderson-Sellers, ISBN 0 201 67510 2, 330pp, Addison-Wesley

reviewed by Matt Pape



This book provides an introduction to the OPEN Process Framework (OPF), which is a framework for defining software development processes. It differs from the Rational Unified Process (RUP) and Extreme Programming (XP) by being less prescriptive in the activities to be performed and their sequencing. This allows for greater tailoring of the process but consequently increased effort for the engineer charged with defining the process.

The book begins by providing a fairly standard definition and justifications for software processes that can be found in many books on software development.

It quickly becomes apparent that the authors are deeply involved in the development of the OPF and this comes across by their strong sales pitch for the OPF within the introduction.

The majority of the book describes the process components (e.g. development activities, roles and products) and this is generally well covered although there is some duplication with the appendices that provide definitions for the components.

The Appendices comprise half the book and much of this information can be found online and felt like padding in an introductory book.

The final chapter, on usage guidelines, is frustratingly short (at just 10 out of the 300 pages) and provides insufficient explanation or examples for defining a process based on the OPF.

Overall, the book provides a good introduction to the OPF (mostly in 30 pages in the second chapter) and should be of interest if you are involved in the task of process definition.

However the book disappoints by failing to provide sufficient implementation guidelines or clear direction of how to proceed if you wish to adopt the OPF.

Real World Software Configuration Management

by Sean Kenefick, ISBN 0-59059-065-1, 439pp, Apress

reviewed by Derek Graham

The role of SCM is often shrouded in a fog created by tool vendors in an effort to make their software solutions essential. This book aims to demystify the role that SCM should play in software development. The start of the book talks about why you need configuration management. There is not a lot of theory but illustrates points with lifelike examples.

Part 2 deals with source control. Chapter 4 gives an overview of source control tools. This may not be very useful to a new graduate just given the SCM job with no control over which tool they use and no use at all to anyone who has been involved with software for any length of time.

Chapters 5 and 6 deal with the use of CVS and MS SourceSafe respectively. Again this might be of some use to a new recruit but I would prefer they learn under supervision as part of a team than from one chapter in a book.

The best part of the book is part 3 which describes the setting up of an SCM environment from scratch and creating scripts to perform a build. The author starts with a bullet list of steps for a build and refines it through pseudo code to an eventual build script. This includes a section on make and Ant.

The front cover says that it "covers both Windows .NET and Linux" but aside from mentioning `rpm` packages (in about a dozen pages in the very last chapter) there is little about Linux. The chapter on Windows .NET discusses using the MS IDE and Perl or VBScript but I would have liked to see tools like NAnt and NUnit mentioned.

The scripts are available from the author but I have not tried using them and cannot comment on them.

Overall a well laid-out and readable book but I feel that the two chapters devoted to using CVS and SourceSafe were unnecessary.

IDE

Official Eclipse 3.0 FAQs

by John Arthorne, Chris Laffra, ISBN 0-321-26838-5, 385pp, Addison-Wesley

reviewed by Silas Brown

If you need to customise or extend the Eclipse platform but you are not entirely sure what you are doing then this book is likely to be helpful. The 361 questions and answers seem well thought-out to give you some pointers no matter what you are trying to do, and it is particularly useful when you are not well practised at navigating around Eclipse, its documentation and all the Eclipse-related information out there.

The book is well indexed and navigable and many questions are cross-referenced. It also comes with a CD-ROM that contains the complete text of the book in HTML, packaged up in a form suitable for installing onto Eclipse as a plugin. This adds itself to the Eclipse help system, hence giving you the choice of accessing it that way if you prefer. This kind of dual-format publishing should perhaps be done more often; it gives you the choice of reading online or paper depending on the circumstances. As you might expect, the CD-ROM also has the source code to the examples, and a nice touch is the inclusion of a mirror of the eclipse.org downloads and supporting documentation as well.

There is also a reference to the book's website, which is heavily dependent on scripting and I found it does not work at all on my browser just because I have set it up with different fonts and colours because I am partially sighted. Therefore, I cannot comment on the content of the website, but that is beyond the scope of this review.

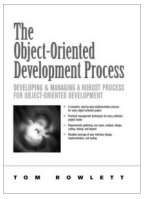
Apart from the website, I would recommend this book to those that need it – developers who need to extend or customize Eclipse and are not entirely sure what they are doing. Others (including ordinary users of Eclipse) might benefit a little from this book but it probably would not justify a purchase in their case.

Process & Patterns

The Object-Oriented Development Process

by Tom Rowlett, ISBN 0 13 030621 5, Prentice Hall

reviewed by James Roberts



This book is a guide to using a particular development process for an object-oriented environment. The author seems to have developed it himself, and named it (modestly) 'The Object-Oriented Development Process'. The process itself is very much the usual kind of thing (use case, state diagrams, class diagrams etc), explained unusually clearly and logically. It also includes non-Object-Oriented techniques (truth tables for example), which, as the author points out, are just as applicable to Object Oriented projects.

The book's strengths are the clarity of the writing, and the way that the author manages to maintain a logical thread of the process from initial requirements through to testing and maintenance (the very mention of which is rather a novelty). I particularly enjoyed the early sections of the book, where the suggested process for generating the analysis model is documented

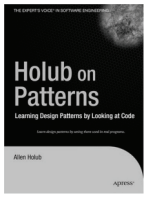
I am not sure whether I would like to take the process defined in this book 'lock, stock and barrel' onto a project. However, I would not hesitate to use this book to help explain to colleagues what we are trying to achieve with existing processes.

One feature of this book is a detailed example of a development project ('video store' implementation), which the author repeatedly returns to for illustration. This would probably be an irritation for someone skimming the book. However, it does illustrate how the process holds together, and so is probably worth the space taken up as an aid to more detailed study.

My only complaint about the book is the title – a reader might be confused into buying this book thinking that it was a review of OO development processes, which this book is not.

Recommended.

Holub on Pattems



by Allen Holub, ISBN 1-59059-388-X, 412pp, Apress

reviewed by Alan Lenton

This interesting, if rather pretentiously titled, book sets out to overcome that vexing gap between the abstract concept of a pattern and how exactly you use it in real life code. The language used for the code is Java, but a competent programmer should be able to cope with the code, even if not a Java expert.

The book uses two full-length programs to achieve its ends. The first is a vastly over-engineered program of Conway's Game of Life, and the second is a production grade embedded SQL implementation. Between them the programs implement all the Gang of Four (GoF) patterns.

I found this book useful. It is a while since I last looked at the GoF book, and this raised enough issues to make me want to go back and take a fresh look from a different angle. I learned two important things from reading the book.

The first was the usefulness of patterns in communicating between programmers. Using patterns can be a very succinct and accurate way of communicating your intent and requirements.

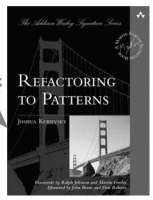
The second was that an object in a program can have different roles within several different patterns. I guess I had known this, but not really thought about it before, and certainly not articulated it.

I also picked up a lot of useful tips and techniques from reading through the code presented in the book, and the introduction on objects contained some interesting ideas that I had not really considered in any depth before.

I have only one caveat. The Life program is on the verge of being so complex as to defeat its own purpose. It would probably have been better to have had three different programs rather than trying to stuff so many patterns into one program. There is a suggestion at the end of the section that the program is an example of the problems that can arise with hard pattern oriented design. Nice try, but no cigar, Mr Holub!

A useful adjunct to the GoF book.

Refactoring to Patterns



by Joshua Kerievsky, ISBN 0-321-21335-1, 367pp, Addison-Wesley

reviewed by Anthony Williams

Refactoring To Patterns brings together the Patterns movement, and the practice of Refactoring commonplace in the Agile community. Whereas the original Gang of Four book told us what patterns were, what sort of problems they solved, and how the code might be structured, Refactoring To Patterns illustrates how, why and when to introduce patterns into an existing codebase.

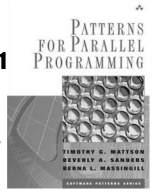
The opening chapters cover the background, introducing both refactoring and design patterns, and the context in which the book was written. This gives the reader a clear overview of what is involved in Refactoring to Patterns, and paves the way for the refactoring catalogue that makes up the bulk of the book.

The catalogue is divided into chapters based on the type of change required – is this a refactoring to simplify code, generalize code, or increase encapsulation and protection? Each chapter has an introduction that gives an overview of the refactorings contained within that chapter, followed by the refactorings themselves. These introductions clearly illustrate the principles and choices that would lead one to follow the refactorings that follow.

Each refactoring starts with a brief one-sentence summary, and before and after structure diagrams with reference to the structure diagrams for the relevant pattern in the Design Patterns book. The sections that follow then cover the Motivation for using this refactoring, step-by-step Mechanics, and a worked Example, relating back to the steps given for the Mechanics. Finally, some of the refactorings finish with Variations on the same theme. The examples are all pulled from a small sample of projects, which are introduced at the beginning of the catalogue section, and help illuminate the instructions given in the Mechanics section. The mechanics themselves are generally clear, and broken down into small steps – sometimes smaller steps than I might take in practice, but I think this is probably wise, as large steps can easily confuse. Finally, the Motivation sections do a good job of explaining why one would choose to do a particular refactoring, and any pitfalls to doing so – the "Benefits and Liabilities" tables provide a useful summary.

This book is well written, easy to read, and genuinely useful. It has helped me put some of the refactorings I do into a larger context, and given me insight into how I can integrate patterns with existing code, rather than designing them in up front. As John Brant and Don Roberts highlight in their Afterword, this is a book to study, the real benefit comes not from knowing the mechanics, but by understanding the motivation, and the process, so that one may apply the same thinking to other scenarios not covered by this book. If you are serious about software development, buy this book, inwardly digest it, and keep it by your side. Highly Recommended.

Patterns for Parallel Programming



by Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill, Addison-Wesley, ISBN 0-321-22811-1

reviewed by Anthony Williams

This book gives a broad overview of techniques for writing parallel programs. It is not an API reference, though it does have examples that use OpenMP, MPI and Java, and contains a brief overview of each in appendices. Instead, it covers the issues you have to think about whilst writing parallel programs, starting with identifying the exploitable concurrency in the application, and moving through techniques for structuring algorithms and data, and various synchronization techniques.

The authors do a thorough job of explaining the jargon surrounding parallel programming, such as what a NUMA machine is, what SPMD means, and what makes a program embarrassingly parallel. They also go into some of the more quantitative aspects, like calculating the efficiency of the parallel design, and the serial overhead.

Most of the content is structured in the form of Patterns (hence the title), which I found to be an unusual way of presenting the information. However, the writing is clear, and easily understood. The examples are well though out, and clearly demonstrate the points being made.

The three APIs used for the examples cover the major types of parallel programming environments – explicit threading (Java), message passing (MPI), and implicit threading

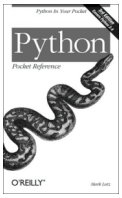
from high-level constructs (OpenMP). Other threading environments generally fall into one of these categories, so it is usually straightforward to see how descriptions can be extended to other environments for parallel programming.

The authors are clearly coming from a high-performance computing background, with massively parallel computers, but HyperThreading and dual-core CPUs are becoming common on desktops, and many of the same issues apply when writing code to exploit the capabilities of these machines.

Highly Recommended. Everyone writing parallel or multi-threaded programs should read this book.

Python

Python Pocket Reference 3rd edition



by Mark Lutz, O'Reilly, ISBN: 0596009402, 148pp

reviewed by Ivan Uemlianin

Recommended

This is a nice little book, very easy to navigate using the just-explicit-enough contents and index, and written nicely enough to browse. I recommend it as an aide memoire type desktop or pocket reference.

The book runs bottom-up through the language, starting with command-line options, through built-in data types and statements, through some commonly used standard library modules (e.g., os, re, sys) and up to sections on Tkinter and the database API and a final section on 'Python idioms and hints'.

As even the most basic topic gets a paragraph, and is easy to find, this would be a good keyboard-side prop for someone newish to Python (or with intermittent memory), and these basics are covered in enough depth to make it worth looking (e.g., print redirection, `else` clauses in `for` statements, etc.).

PPR obviously does not cover the standard library comprehensively, and here the book's small size works in its favour: no-one is going to think this is comprehensive, and if you can't find what you want here your first response will be to look at the online documentation. For this reason I'm tempted to recommend PPR over Python in a Nutshell, which is not comprehensive either but is big enough to make you think it should be. In fact PPR covers things silently left out of PN (for example, `os.samefile()`).

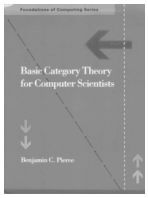
The concluding 'idioms and hints' section has a breezy 'and finally, ...' feel but contains tips on idiomatic usage that are useful and slightly out-of-the-way.

This book exceeded my admittedly low expectations. I'm always glad I've dipped into it, but I use the online docs for real help.

Is it worth £7? If you need quick reminders, or if you're a dipper: yes.

General Programming

Basic Category Theory for Computer Scientists



by Benjamin C. Pierce, ISBN 0-262-66071-7, MIT

reviewed by Francis Glassborow

This book was published a dozen years ago. The reason that I have a copy is that some highly technical discussion of the C++ type system referred repeatedly to Category Theory. Despite my advancing years, I spend a good deal of time trying to bring my knowledge and understanding to the level where I can understand issues relating to language design and implementation. (One of the advantages I have over those who are fully employed in software development is that I have much more time spend on personal development.)

Category Theory is a relatively young branch of pure mathematics derived form algebraic topology. Now it is unlikely that the reader of this review has such a background so be warned that this is tough mathematics for the newcomer; it is tough for me even though my main tutor for my Oxford degree was an algebraic topologist and my own specialism was mathematical logic. This is the kind of book that you study a few pages at a time and that older brains such as mine find hard to turn the acquired knowledge into working knowledge.

Chapter 1 introduces the reader to the basics of Category Theory. There is no reasonable way for me to summarise that. Perhaps the best advise I can give is to suggest you type 'Category Theory' into Google and start looking at some of the early hits from the almost four million you will get.

In chapter 2 we move on to functors, transformations and adjoints. When you have finished studying (just a reading is insufficient preparation) you will be ready to read chapter 3 which looks at various ways that CT can be used in the computer science domain.

Chapter 4 is entirely concerned with further reading. It is badly dated and the reader would probably be better served today by an intelligent use of Google or some other search engine.

Whilst I would recommend this book to someone with an appropriate background and an interest in theoretical aspects of computer languages, it is not a book for the working programmer or software developer. This is a book for those that are either very interested in or who need to study the foundations of computer language design.

Working Effectively With Legacy Code

by Michael C. Feathers, Prentice Hall PTR, ISBN 0-13-117705-2

reviewed by Anthony Williams

Michael puts a new spin on the term "legacy code", identifying it as code without automated tests. For those of us used to the more traditional meaning of the term, referring to old codebases full of cruft and quick fixes on top of hasty modifications on top of quick fixes, this is a somewhat unusual attitude – we like to think that our new code is clean and not "legacy code", even if it doesn't have automated tests. However, it doesn't take long for the former to turn into the latter – leave it 6 months whilst you work on another project, so you've forgotten the details, then rush through a bug fix because it needs to be in production tomorrow, and you're well on the way.

A lot of the book is spent covering different techniques for getting nasty convoluted code under test. These techniques often focus on breaking dependencies, so you can instantiate an object in a test harness, or call a function without it sending an email or talking to the database. In many cases, these are sensible recommendations for improvements to the codebase, but in some cases, Michael recommends techniques you wouldn't expect to see in production code, such as defining preprocessor macros with the same name as functions to avoid calling them, or writing dummy implementations of API functions. This is not to say that the book recommends such techniques wholesale – Michael is keen to point out that these techniques should only be used to get the code under test, so that it can be refactored safely.

Chapters are helpfully named, with titles like "This class is too big, and I don't want it to get any bigger", and the book gives good advice on how to deal with the nastiest codebases. The key recommendation underlying it all is "*get the code under test, so you can refactor safely*", and this is borne in mind with those techniques that require changing the production code in order to get it under test – these techniques provide step by step instructions to help you make the required changes without breaking anything.

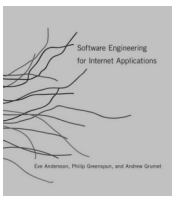
The content of this book is excellent, and the writing clear, so it is unfortunate that it is marred by numerous minor errors, such as spelling mistakes, or using the wrong identifier when referring to a code example. However, this minor niggle is not enough to stop me recommending it – every software developer should have a copy.

Highly Recommended. This book is a must-have for anyone who has to maintain code – which is pretty much every software developer on the planet.

Software Engineering for Internet Applications

by Andersson, Greenspun and Grumet, Wiley, 0-262-51191-6, 390pp

reviewed by Paul F. Johnson



If you're a student at MIT, please completely ignore this review. For everyone else, please read on.

The blurb on a book should give you an indication of what the book is about. Okay, in some circumstances (such as anything by Schildt and a few others), it is as misleading your average politician, however they are mostly accurate and at least give an indication of the book's content. The blurb on this book looks fantastic – it says that by the end of the book you will be able to build an internet application to rival Amazon.

Sounds promising and the first chapter doesn't disappoint. The first chapter is probably the best out of the lot. Why? It doesn't actually contain very much in the way of usable information.

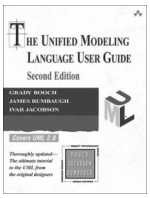
From then on in, the book just goes down hill – rapidly. It is a book which is based on course notes, delivered to MIT students. The problem is though, that is all they are. There are sections which go into detail over what you need to be doing with your "lab partner". Fine if you're an MIT student, pretty much useless if you're not.

Because of this 'course book' style, very little of it makes sense as the linking arguments are just not there. There is no explaination of how the internet application works with the backend, very little on the front end and the database accessing is limited to postgreSQL, Microsoft's SQL system and another relational database system.

By the end of chapter 4 you're thinking why did you every pick up this book and midway through chapter 5, that pile of dishes suddenly looks like a good idea.

Not recommended (unless you're an MIT student)

The Unified Modelling Language User Guide – 2nd Ed.



by Grady Booch et al , Addison-Wesley, ISBN 0-321-26797-4

reviewed by Stephen Foreman

I want to start by saying that I like this book. The technical accuracy and content of this book cannot be questioned, however the writing style of the book is not the best. I found the heavy and often over the top style of this book very similar to the GOF's Design Patterns book.

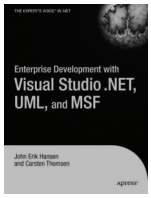
Another thing that troubled me about this book is the fact it is called a 'User Guide' but has so few examples of how to use the UML. This book is one of three that have been written by the Amigos in this series and the prefix of this book points out that the User Guide is not either of these other books, but in my opinion they have tried too hard to separate the subjects of the books and in doing so have missed out some valuable content.

My final negative is that the editing in some places is sloppy.

The typeset of the book is good and the chapters all follow a similar structure that becomes familiar. The chapters themselves are ordered logically with the book being split into basic and advanced parts. Each page of this book is packed with cross references to different chapters, which makes navigating around the book easy.

I would recommend this book and would say that anyone who uses UML needs access to it.

Enterprise Development with Visual Studio .NET, UML and MSF



by John Erik Hansen and Carsten Thomsen, Apress, 1590590422, 955pp

reviewed by Ivan Uemlianin

This book (ED) is organised into six parts. The bulk of the book is part 2 on UML and part 3 on the Visual Studio .NET Enterprise family of development applications, including Enterprise Template Projects, Visual SourceSafe, Visio, Application Center Test, and Visual Studio Analyzer. Later parts look at worked examples, a comparison of Microsoft and IBM enterprise development frameworks, and the Microsoft Solutions Framework.

ED is apparently thorough and is clearly written. It carefully guides the reader through the MSF analysis and design process, and the use of the applications (which are included on a CD, not). Exercises give the reader plenty of opportunity to test their grasp of the text (if the reader has Enterprise Developer). Gathering requirements, writing documentation and maintaining the code are all given a prominent place in the process. If this were my textbook for a course on using MS design tools I would feel very safe.

As you can probably tell from the title, ED lives entirely within the Microsoft world – making reference, for example, to "unsafe code, such as pointers" (p. 217) – and consequently will only be of use to people who are happy to stay within that world. This is a slight shame with regard to the part on UML, which is quite good. If this book is in a local library and you're struggling with UML it might be worth a look.

Another suggestion given by the title is that the book is about enterprise development. It is about enterprise development to the extent that it is about using VS.NET Enterprise Developer (and Enterprise Architect). It's pitched at a very elementary level – the simple language, gentle pacing and generous use of screenshots add to this impression. This book does not tackle weighty issues. I am left with an image of the human developer (or "architect") who is an appendage of the development framework.

If you have just bought MS VS.NET ED/A for yourself, you can afford this book as a user's guide. If you are teaching or studying a course based on VSED, this will be an undemanding and comforting textbook. If you are or have a new junior member on your VSED-based development team, you might want something a bit more snappy.

Miscellaneous

New thinking for the New Millennium

by Edward de Bono, ISBN 0-670-88846-X.

reviewed by Ian Bruntlett.

de Bono considers that traditional thinking, only deployed when a problem is noticed, leads to complacency. This ignores constructive thinking, creative thinking, design thinking in which we seek to improve an existing situation, even if we are not aware of any faults.

de Bono states that, during expansion phases (as in the settling of the American West or during the expansion of the British Empire), action was more important than concepts. After the expansion phase was over, design thinking is better – rearranging things to get more value out of what is already available. In this kind of thinking, concepts play a key role. Where an eye on the past keeps us out of trouble, making a habit out of design thinking will open up new opportunities.

de Bono is scandalised by the poor thinking encouraged by the education system and the media. He states we ought to have a strong sense of value in order to design change based on a sense of "what can be". And our vocabulary should be expanded to describe creativity better.

Parts of this book is full of unstructured thoughts – so, in parts, this book could be titled "Ruminations on thinking".

Verdict: Read this book if you want to chew the cud about thinking. If you haven't read "Lateral Thinking" or "Six thinking Hats" then you might want to read them as well.

Dancing Barefoot

by Wil Wheaton, ISBN 0-596-00674-8, O'Reilly

reviewed by Francis Glassborow

This is another of those odd books that O'Reilly publish from time to time. It is a collection of five short autobiographical stories from the keyboard of Wil Wheaton. I have to confess to my puzzlement that O'Reilly should be the publisher. The stories are well written and enjoyable but no more so than many of the anecdotes that get shared after hours at many conferences, meetings etc. that I have attended. I am sure that such excellent raconteurs as Andy Koenig could fill ten times as many pages with material of equal quality and authenticity.

Yes, I do believe that books such as this one have their place and that we should make publishing of them easier. There are many people with writing talent who never get published (and there are many with far less talent whose works clutter the shelves of book retailers).

If you want a light read (that may trigger deeper responses in you) for a journey or for a winter

evening by the fire this book would be a good candidate. The teacher in me also would use the stories as subjects for discussion among teenagers.

Fearless Change

by Mary Lynn Manns, Ph.D. & Linda Rising, Ph.D., ISBN 0-201-74157-1, 273pp, Addison-Wesley

reviewed by Francis Glassborow

We live in a world where the context of our work and the tools we have for doing it change ever more rapidly. In the distant past, a change that had serious impact on individuals would have been rare. Things like the printing press had life changing impact on those who were alive. However, the norm was that you would still be doing the same job in the same way at the end of your life, as you were when you started.

This is no longer the case. Sometime during the last century, the rate of innovation became so high that individuals should expect that whatever job they do at the start of their working lives would either disappear or change very radically before the end of their working lives.

It has proved very difficult for individuals to adjust to this new pattern to work, however it has proved even harder for companies to respond. Most companies have a good deal invested in the status quo; they will resist change.

Most companies will have a few individuals with vision and understanding of the implications of new technology. Unfortunately, most companies will do their utmost to avoid taking notice of such employees.

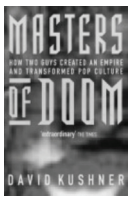
The purpose of this book is to introduce readers to various aspects of pursuing change. The authors are attempting to help the innovators work effectively. Somehow, companies need to find the balance between the stagnation of zero change and the chaos of constant change.

The authors have borrowed the idea of patterns from software development and applied it to working for healthy change to organisations of which you are a part. Such organisations are not just companies, but include clubs, associations, families and churches.

This book describes 48 patterns for achieving change. Like almost all pattern catalogues the result may seem obvious, but that is only after you have read them.

I think that, whoever you are, this is a book that is worth reading.

MASTERS OF DOOM – How two guys created an empire and transformed pop culture



by David Kushner, Piatkus Books, ISBN 0749924896, 335pp

reviewed by Ian Bruntlett.

This book chronicles the working lives of two key industry figures, John Carmack (monkish core engine developer) and John Romero (rock-star-like games designer).

Romero had a tough childhood and frequently escaped from reality through computer games (arcade games – Asteroids or text based adventure games – Colossal Cave) and roleplaying games (TSR's AD&D). He started playing on the arcades. He then started programming a mainframe running Hewlett Packard BASIC, then moved to an Apple II , programmed in assembly language. He was a brash, confident and highly successful developer, getting titles published while still at school. Eventually he arrived at a company called Softdisk.

John Carmack was brought up by academic parents and he enjoyed the usual hacker stuff – sci-fi, fantasy, D&D and computer games. He, too, came from a broken family. Like John Romero, he wrote games as a freelance, eventually ending up working for Softdisk.

After various difficulties, the two Johns ended up working at Softdisk, writing PC games at a rate of one every two months. John Carmack was the ultimate coder and John Romero the ultimate games designer. Although the two Johns were competent, they knew their limitations and demanded a manager and an artist so they could concentrate on what they did best – producing state of the art games with excellent playability.

Some of the struggles to get the (then sluggish) PCs have fast graphics are explained throughout the book. However, I would have liked to see more detail about this, with references for readers who wanted to go further and experiment with the Doom source code that can be downloaded from id software's website.

This book also chronicles a series of ground-breaking PC projects – ranging all the way from the primitive stuff all the way to Quake Arena. It is a fascinating book, full of gems of information.

At Softdisk, Carmack pioneered scrolling. When Romero saw this he persuaded the others that this was too big for Softdisk – so they started moonlighting. A shareware entrepreneur, Scott Miller, approached Romero very indirectly with a view to publishing their games at his startup company, Apogee. They used their ground-breaking graphics engine to develop a game called Commander Keen, listening to rock music, playing arcade games and taking the occasional break to play D&D. All this activity took its toll on family life.

So id software was born. With shareware money paying the bills, Carmack was free to develop the technology required to implement a decent first person arcade game on a PC, Hovertank. Successive releases got better as Carmack refined the engine and everybody else became better at exploiting it. Eventually they flew from the nest (Shreveport) and moved to Madison.

Romero worked but he also researched (playing other company's games) so he kept Carmack fed with new ideas – in this case, texture mapping. Combined with ray-casting, this resulted in Wolfentstein 3D.

They tried selling out to Sierra On Line but the deal fell through. So they remained independent. As time progressed, the engine was fine-tuned and more levels and games were being generated by third parties, leading the way to the development of a new game – Doom. Doom was "yet another levels game" which, not for the last time, resulted in creative differences of opinion within id software. The increasingly realistic and explicit material began to attract unwelcome political interest.

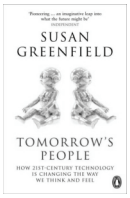
Ultimately the creative differences between the two Johns led to a parking of ways – Romero leaving for pastures new. Others followed suit. Romero's departure allowed him to pursue his dream of starting a design focussed company, Ion Storm – in stark contrast to id's technology based approach.

Politics intervened again with the Columbine shootings.

This book took 6 years and hundreds of interviews. This is a high quality work and it shows.

Highly recommended.

Tomorrow's People : How 21st Century technology is changing the way we think and feel



by Professor Susan Greenfield, Penguin, ISBN 0-141-00888-1

reviewed by Ian Bruntlett

Professor Susan Greenfield (Director of the Royal Institution, neuroscientist) is one smart cookie. In this book she makes predictions on the future of today's population.

She covers changes to lifestyle, robots, work, reproduction, education, science, terrorism and human nature. The thing I find scariest is how pervasive IT is going to become in the future.

Highly Recommended

View From the Chair

Jez Higgins
chair@accu.org

Where to begin? This is, I'm happy and proud to say, my first View From The Chair. As described last C Vu, Ewan has stood down as Chair, and the AGM was very kind in electing me to replace him. At the AGM, I gave a rather nervous and stumbling little speech about why I was there and why I wanted to be Chair. I'm still rather nervous writing this, but at least I have the luxury of a bit of time and editing.

Put simply, the ACCU has made me a better programmer. I've learned, and continue to learn, a huge amount from the journals, on accu-general, and at the conference. I've learned a great deal from the many people I've met through the ACCU, both in person and on line. Because I'm a better programmer, I've had more fun. I like fun! So thanks! I thank you, my family thanks you.

By taking on the job of Chair, I hope to carry on having fun and to ensure the ACCU helps all its members become better programmers and have more fun. I'd like to try and extend the reach of the ACCU too - we can't be the only "programmers who care". There must be more of us out there, and they deserve to be having more fun too.

I'd like to thank Ewan for his hard work as Chair over the past three years. He is, I'm delighted to say, staying on as Conference Chair, a job which this year's conference again demonstrated he's terrific at. I got a tiny glimpse, this time, of the work involved in organising the conference. Ewan, my family thanks you.

Membership Report

David Hodge
membership@accu.org

In 2005 we had 109 new members.

In 2006 (4.5 months) we have so far had 102 new members.

My next job is to create the spec for the web changes to do the membership mainly online (any comments on list below are welcome).

Summary of automatic website operations:

- Join as a new member (Members from certain countries require manual validation)
- Renew membership
- Change address details
- Automatic generation of journal label file for new members (daily)
- Automatic reminders 4 weeks and 2 weeks before expiry date
- Automatic removal from member group on expiry
- Automatic generation of journal numbers for production person
- Automatic generation of journal label file for journal distribution
- Automatic email to all members on an ad hoc basis (manually initiated)
- Automatic generation of statistics on manual request

Summary of duties of new membership secretary:

- Create labels from daily new member file and post latest journals
- Create and enclose welcome letter (This could maybe done by Able Types at a cost, if they held the spare copies)
- Manually validate at the request of the website new members from certain countries (we have had a few fraudulent tries from Nigeria in the past year)
- Answer any queries coming to membership@accu.org

Website Report

Allan Kelly
allan@allankelly.net

The conference allowed me (and Tony and Tim) to talk to a lot of people about the website and talk ourselves to better understand what we do next.

Our priorities for the next few months are:

1. Move the mailing lists to the new server
2. Move the journals to the new server
3. Create a new membership system

There is some debate about what happens when we get the journals on the new server. Do we keep the current policy of members-only access? Perhaps with the odd 'teaser' piece to entice in new members.

Or, should we open up our archives to the public: we'll have more content, perhaps generating advertising revenue and bringing in more members. Going further, we could publish the latest journals online - this would keep the site fresh and pull in more advertising but would we still have members?

Since the conference there has been a lot of talk on the committee list about creating local groups - inspired by Reg Charney. One of the things we've started thinking about is creating an online calendar to track all these events.

Hopefully by the time I write the next report we'll have some new stuff to talk about. ■