the magazine of the accu

www.accu.org

{cvu}

Volume 29 • Issue 6 • January 2018 • £3

Features

Regulars

Martin-Baker

SOFTWARE ENGINEER - DENHAM

Martin-Baker is the world's leading manufacturer of aircrew escape and safety systems. It is the only company that can offer a fully integrated escape system that satisfies the very latest in pilot operational capability and safety standards. We are looking for talented and dedicated people that are interested in making a real difference and, like us, are proud to work for a Company whose products have saved over 7550 lives.

We have a fantastic opportunity for Software Engineers working in the Systems Engineering department at the Denham site. As a software engineer you will be joining the team responsible for development and maintenance of critical software assets integral to the growth of Martin-Baker.

This includes:

- Embedded software for safety-critical applications
- Embedded and PC-based test software, used to for verification and in-service usage
- PC-based simulation software, used to model the behaviour of ejection seats during use

Typical work for the Software Engineering Group includes:

- Working with Systems Engineers to specify system and software requirements
- Implementing innovative software solutions
- Delivering a technology refresh to the software toolchain
- Improving Software Group department productivity through targeted infrastructure enhancement projects

We are looking for good engineers who focus on producing high integrity software solutions, work with a wide range of toolsets, and reason about software requirements and design independently of the technology used for implementation.

Martin-Baker offers:

- A competitive salary
- A Non-Contributory Pension Scheme and Life Assurance Scheme
- A Healthcare Scheme
- The equivalent of 25 days annual leave plus statutory holidays
- The opportunity to develop your skills and progress your career
- A great working environment where you will work alongside experts who can help you to obtain a wide understanding of our business

The ideal candidate will be degree-qualified in an appropriate engineering discipline and have previous experience using at least 2 of the following: C (including MISRA an advantage), C++, C#, Java and Ada (including Ada 2012 and SPARK 2014).

For all successful candidates, Martin-Baker will undertake background security checks. As part of this, we will need to confirm your identity, employment history and address history to cover the past five years as well as your nationality, immigration status and criminal record. For positions that require Security Clearance, the successful candidate must hold or be willing to obtain security clearance up to the relevant level for the role.

To apply for this position please email send your covering letter, CV and current salary expectations by email to the Human Resources department at recruitment@martin-baker.co.uk.

Engineering For Life

EJECTION SEAT

Martin-Baker

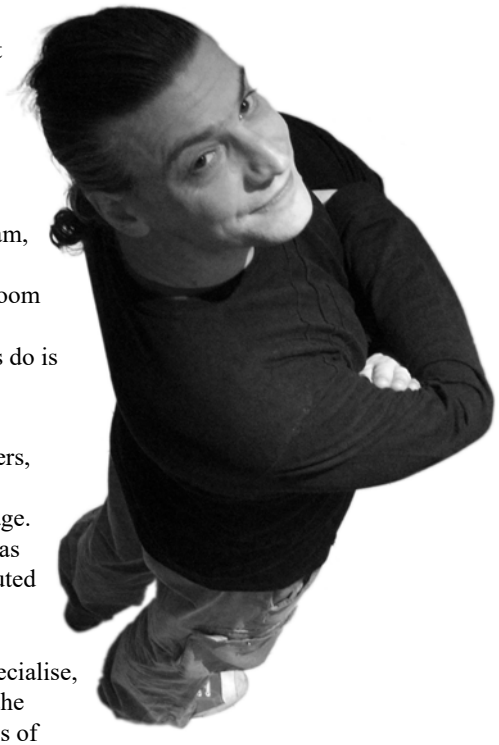
7553* Aircrew Lives Saved

*October 2017

www.martin-baker.com

{cvu}
Volume 29 Issue 6
January 2018
ISSN 1354-3164
www.accu.org

Know It All



There's a perception of the computer programmer as a kind of hermit wizard: alone with a keyboard, late at night, face lit by the light from an array of monitors, working magic spells with dread incantations. Notwithstanding the reinforcement of gender stereotypes going on here, it's probably not a completely unjustified image, but today's programmer is much more likely to work in a team, and the spell books are widely available in mainstream bookstores. The bad-tempered bedroom hacker image persists, mostly in terms of cyber-criminality, but by and large, what programmers do is no longer shrouded in mystique.

It's probably true that programmers are more comfortable in the company of other programmers, overall, because it's still easier to converse with someone who understands your (spoken) language. Nevertheless, the idea of 'hacker culture' that was so prized during the last century has become diluted to some extent. The sheer number of different programming languages and practices makes it impossible to know everything, and so as we specialise, new sub-sub-cultures arise. We mix and match the various sub-genres, of course, along various axes of programming language, platform, DB, favourite editor, etc.

I once had a kind of parody Tarot card, called 'The Developer'. I'd now need a whole deck: The Functional, The Embedded, The High-Performance, The Security, The Service, The Scientific, The UX...

Which brings me neatly to the term 'Full Stack Developer'. It began, fairly simply, as meaning a developer who could put together a full application with one of the popular technology 'stacks': LAMP, MEAN, LEAP, WINS. It now seems that 'Full Stack' cannot be expressed as a pronounceable word, never mind one with a single syllable. ANTJPMCDKMRK would be needed for one job I saw advertised – and that was just *some* of the technologies being used. Any takers on what that might stand for?

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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.

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 programmers, for programmers – and have been contributed free of charge.

To find out more about 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

SUBMISSION DATES

C Vu 30.1 1st February 2018
C Vu 30.2: 1st April 2018

Overload 143: 1st March 2018
Overload 144: 1st May 2018

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

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

Visualisation of Multidimensional Data

Francis Buontempo considers how to represent large data sets.

We are familiar with scatter plots for two dimensional data. Simply use the a-axis for one dimension and the y-axis for the other and plot your points. Job done. How do you visualise data with more than two dimensions? You can manage a three dimensional plot, though either need this to be interactive so you can look at your graph from different angles or print plots from a variety of projections to draw out salient features. For more than three dimensions you are in trouble.

Let's look at two ways to display high-dimensional data. The UCI holds a repository of a variety of data sets that are commonly used to show case machine learning algorithms. One frequently used set is the iris data [1].

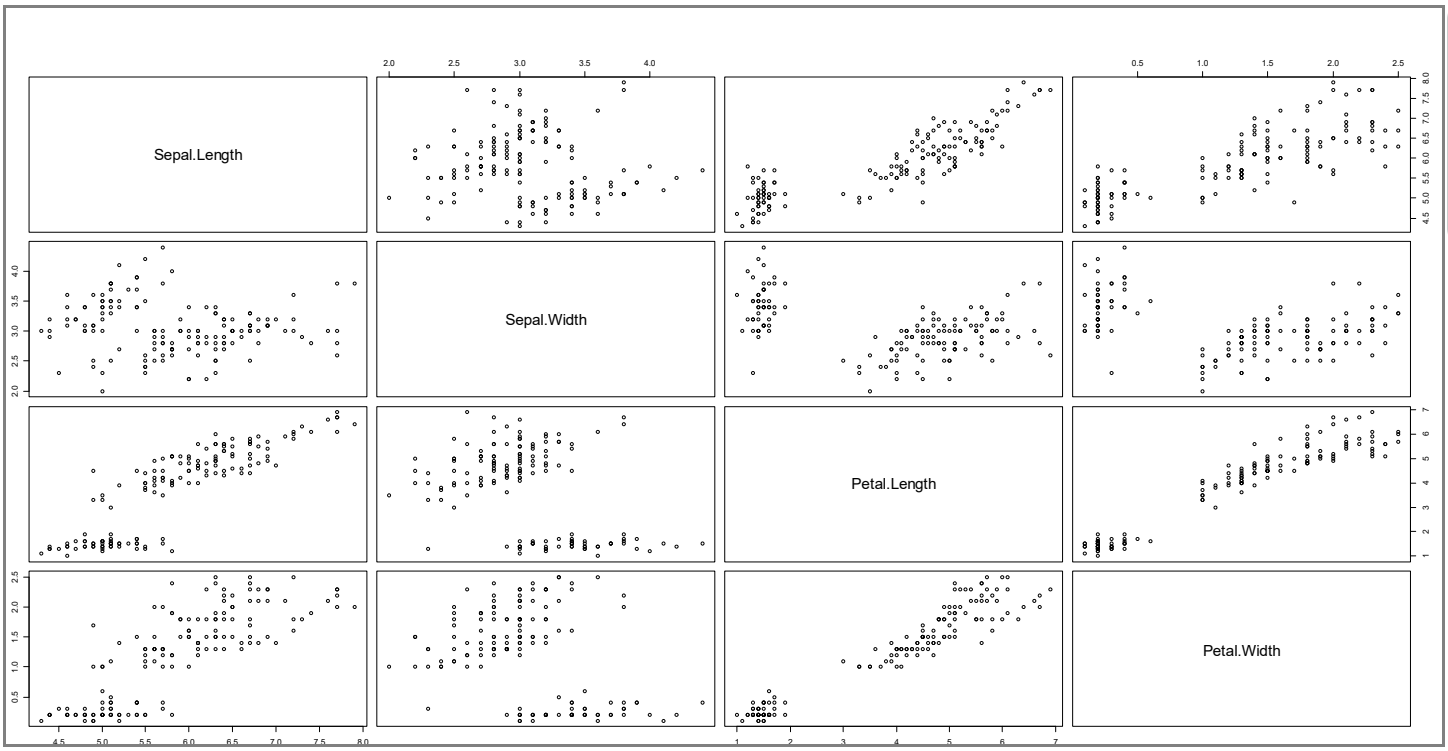


Figure 1

This contains 150 instances of measurements of iris flowers along with the category or class of iris each belongs to:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
   - Iris Setosa
   - Iris Versicolour
   - Iris Virginica

There are 50 examples or instances in each class, in blocks, so you know which is which. A machine learning algorithm will try to find ways to group the data correctly. We will ignore the class and just concentrate on the over-whelming (!) four dimensions of the data.

Scatter plots

This data set is extremely common. You can just load it in R:

```
data(iris)
```

You can then ask for pairs of scatter plots. Exclude the final class column to see how the attributes correlate:

```
pairs(iris[, 1:4])
```

This plots a matrix of scatter plots showing each pair of attributes in turn (Figure 1). (See the Quick-R website [2] for further details.)

You can't immediately see the three different types of iris in these plots. You can seem some apparent correlations between the attributes though. This will get out of hand for more than a few dimensions. Let's see an alternative approach to plotting lots of x values.

Parallel coordinates

A common way to plot such data is one line per data point on parallel y axes. Wikipedia [3] has an example point of the fabled iris data set. I said it was common! This approach, unlike our pairs of scatter plots, is scalable. You just need an extra y-axes in parallel for each new attribute.

If you download the data, and put quotes around the text in the final category column you can load it easily in Python. If you use numpy, you will make your life easier. Import pylab, or a graph package of your choice and load the data (see Listing 1).

This code plots 150 lines, on the same graph. The magic `T` *transposes* the data, swapping the rows and columns. Without the magic `T`,you are

Frances Buontempo Frances has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Listing 1

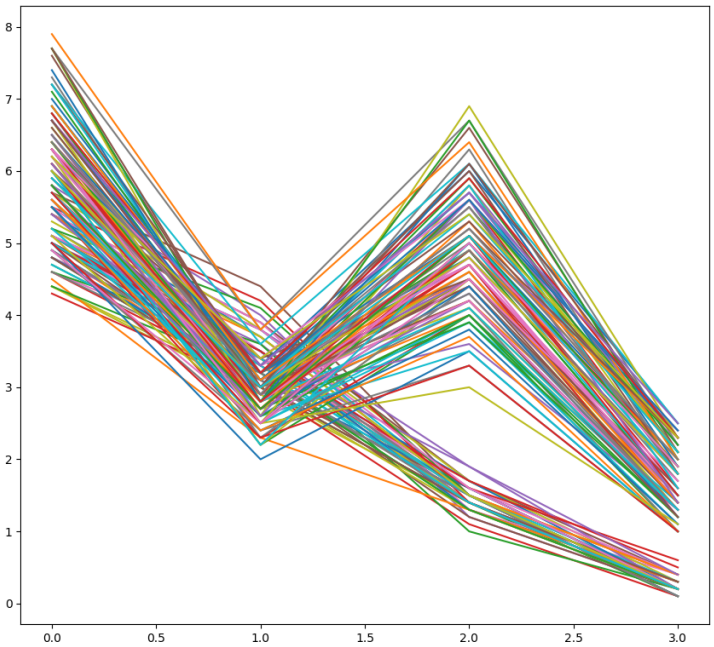
```
import numpy as np
from pylab import *

def display():
  csv = np.genfromtxt ('iris.data',
delimiter=",")
  fig = figure(figsize=(11,11))
  plot(csv [:, 0:4].T)  #Magic!
  show()
```

sending the four attributes to pylab, so get four lines; one per attribute, with 150 data points on each. Once transposed, you have 150 lines with four points on each; one point per attribute. This gives you the four parallel coordinates you are after.

You can see (in Figure 2) where the four numeric attribute columns are; obviously at the left and right and then at the trough and spike in between. We end up with a colour for each, which is a bit multi-coloured. For those viewing in black and white, that's a relief I assure you.

We just get a y-axis labelled for the first attribute, which we could add to.

Figure 2



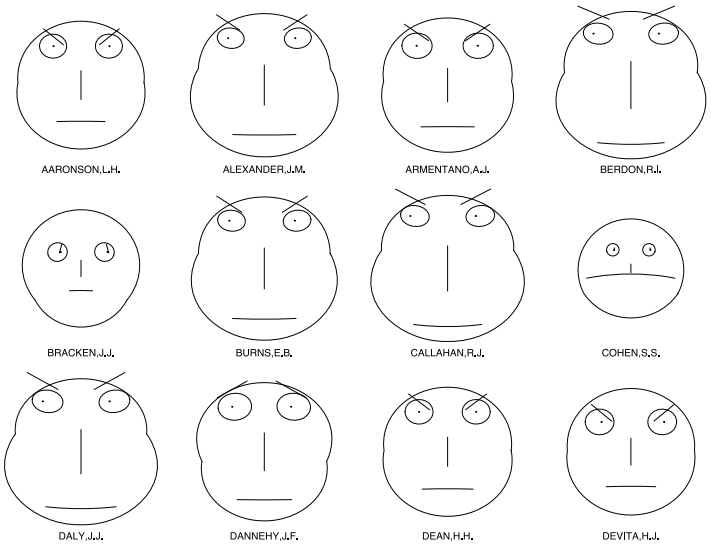
The a-axis is a bit pointless. We want four y axes, in parallel. I'll leave that as an exercise for the reader. It's easy enough to use **axvline** to draw vertical lines where you need them. You could try to plot each line in a colour dependant on the iris type too. In general, you should normalise the data. In the iris case, the numbers are all between 0 and 8 or so. However, you can see the last column has smaller values, so to do this properly we should scale everything between 0 and 1. If you had data with, say height in centimetres and shoe size you are likely to get the shoe sizes squashed up compared to the heights.

As your data sets increase in size, you can easily add extra coordinates. One potential downside of parallel coordinates is the patterns you see can depend on the order you place the axes in. Many visualization are interactive, allowing you to drag the columns around to see what's happening. Let's look at one last way to visualise multi-dimensional data that circumvents this problem.

Chernoff faces

Herman Chernoff deigned a way to show multi-dimensional data using faces in 1973 [4]. The idea is to map attributes of a dataset to salient features of faces, for example face shape, nose size, slant of mouth, eye shape and so on. I fell across this on Twitter and tried it for the iris data set. His motivation was an assumption that you are wired up to recognise faces, so will spot similar and different patterns easily. Wikipedia [5]

Figure 3



shows a plot of rating of judges, in which you can see some very similar faces and a few outliers (see Figure 3).

Wikipedia has a link to some Python code [6], providing a function called **cface**, which I will leave you to experiment with. This uses 18 attributes:

1. height of upper face
2. overlap of lower face
3. half of vertical size of face
4. width of upper face
5. width of lower face
6. length of nose
7. vertical position of mouth
8. curvature of mouth
9. width of mouth
10. vertical position of eyes
11. separation of eyes
12. slant of eyes
13. eccentricity of eyes
14. size of eyes
15. position of pupils
16. vertical position of eyebrows
17. slant of eyebrows
18. size of eyebrows

Using this for the iris data is not very sensible, since this data set only has four attributes. This didn't stop me, though, and after a bit of

Listing 2

```
with open('iris.data', 'r') as csvfile:
  reader = csv.reader(csvfile)
  fig = figure(figsize=(11,11))
  i = 0
  for row in reader:
    ax = fig.add_subplot(15,10,i+1,aspect='equal')
    data = [0.5]*17
    data[0] = float(row[1]) #overlap of lower face
    data[1] = float(row[0]) # half of vertical
size of face
    data[2] = float(row[2]) # width of upper face
    data[12] = float(row[3]) # size of eyes
    cface(ax, .9, *data)
    ax.axis([-1.2,1.2,-1.2,1.2])
    ax.set_xticks([])
    ax.set_yticks([])
    i += 1
```

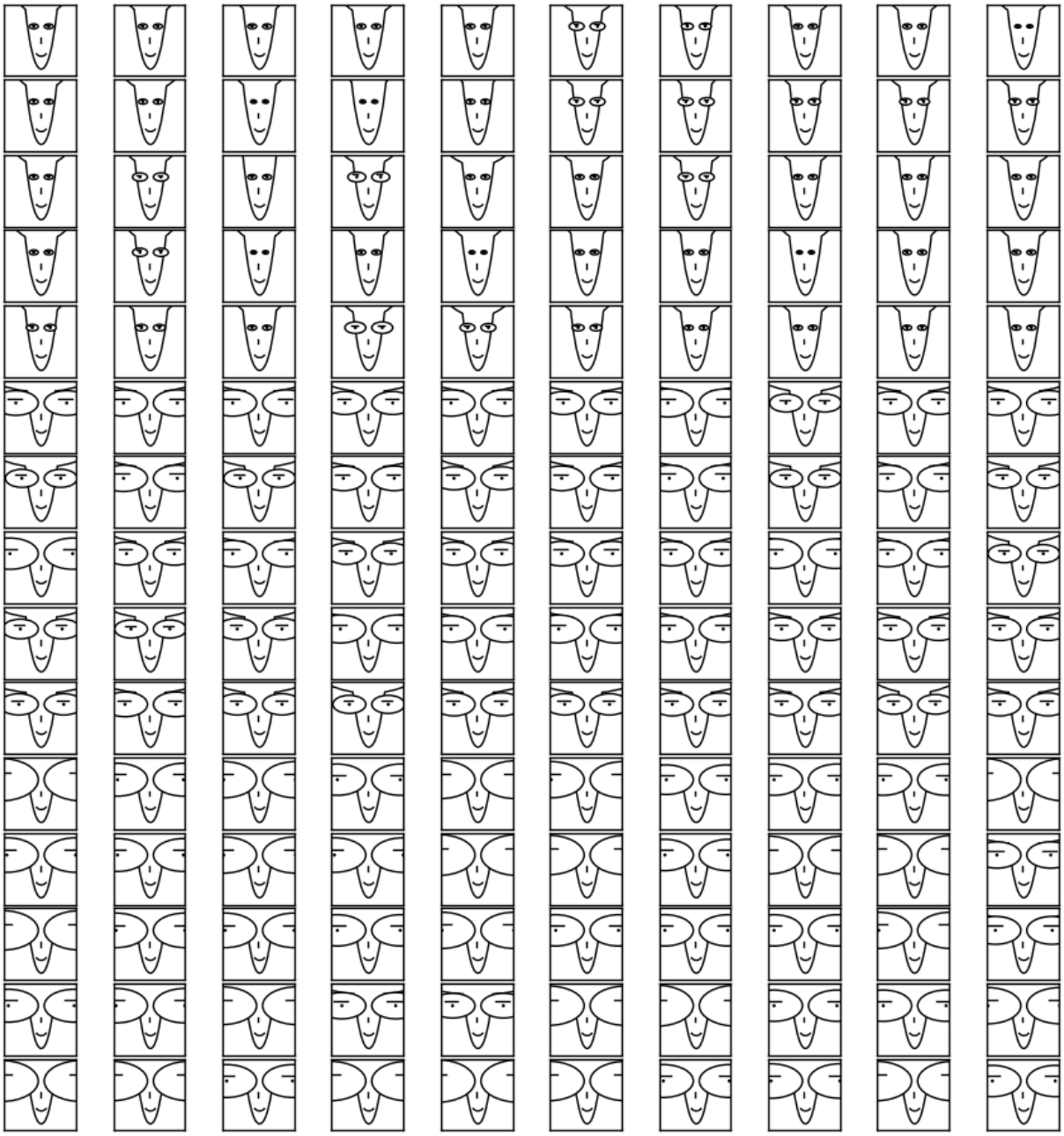


Figure 4

experimentation I chose four of the eighteen facial features. The sample code fixed the first feature, height of upper face to 0.9. so I followed suit. I set the others to 0.5, apart from the four, related to face size and eye size, to produce 150 faces for the 150 data points. (See Listing 2 and Figure 4.)

You can clearly see the first 50, setosa iris flowers look very different to the next 100. Since the data is in groups of 50; setosa, versicolor then virginica, we know the first five rows are setosa, and so on. Had I labelled the earlier plots more clearly, you would have seen the petal width and lengths are much smaller for setosa flowers. The scatter plots have a clump of separated points for the petal attributes. The parallel coordinates also have a bunch of lines separated from the rest on the last two columns. You can also see a slight difference between the next two groups of flowers. The eyes, mapped to petal length does provide some disambiguation between versicolor and virginica flowers.

Conclusion

There are many ways to display multi-variate data. None are ideal, so it's worth trying a few approaches if you have some data you want to explore. Everyone has seen scatter plots before. They are common because they can be very informative. Parallel coordinates are not so widely known - you probably didn't study them at school. They do crop up quick frequently in serious data analysis studies so are worth knowing about. Chernoff faces seem to be relatively obscure. I can see a few academic articles and critiques of the technique on Google. I think the idea of a projection onto features is worth considering for data analysis. However, I suggest you have the same number of features as data columns, rather than spending far too much time trying to find the best four of eighteen to use if you want to spot differences in the iris data! ■

References

[1] Iris database: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/

[2] Quick-R: https://www.statmethods.net/graphs/scatterplot.html

[3] Wikipedia: https://en.wikipedia.org/wiki/Parallel_coordinates

[4] Chernoff faces: https://en.wikipedia.org/wiki/Chernoff_face

[5] Wikipedia (Chernoff faces): https://en.wikipedia.org/wiki/Chernoff_face

[6] Python example: https://gist.github.com/aflaxman/4043086

Testing Times (Part 1)

Pete Goodliffe explores how to test code to ensure it works as expected.

Quality is free, but only to those who are willing to pay heavily for it.
~ Tom DeMarco and Timothy Lister,
Peopleware: Productive Projects and Teams

Test-driven development (TDD): to some it's a religion. To some, it's the only sane way to develop code. To some, it's a nice idea that they can't quite make work. And to others, it's a pure waste of effort.

What is it, really?

TDD is an important technique for building better software, although there is still confusion over what it means to be *test driven*, and over what a *unit test* really is. Let's break through this and discover a healthy approach to developer testing, so we can write better code.

Why test?

It's a no-brainer: we *have* to test our code.

Of course you run your new program to see whether it works. Few programmers are confident enough, or arrogant enough, to write code and release it without trying it out *somehow*. When you do see corners cut, the code rarely works the first time: problems are found, either by QA, or – worse – when a customer uses it.

Shortening the feedback loop

To develop great software, and develop it well, programmers need *feedback*. We need to receive feedback as frequently and as quickly as possible. Good testing strategies shorten the feedback loop, so we can work most effectively:

- We know that our code works when it's used in the field and returns accurate results to users. If it doesn't, they complain. If that was our only feedback loop, software development would be very slow and very expensive. We can do better.
- To ensure correctness *before* we ship, the QA team tests candidate releases. This pulls in the feedback loop; the answers come back more quickly, and we avoid making expensive (and embarrassing) mistakes in the field. But we can still do better.
- We want to check that our new subsystems work before integrating them into the project. Typically, a developer will spin up the application and execute their new code as best they can. Some code can be rather inconvenient to test like this, so it's possible to create a small separate test harness application that exercises the code. These *development tests* again reduce the feedback loop; now we find out whether our code is functioning correctly *as we work on it*, not later on. But we can still do better.
- The subsystems are comprised of smaller units: classes and functions. If we can easily get feedback on correctness and quality of code at this level, then we reduce the feedback loop again. Tests at the smallest level give the fastest feedback.

The shorter the feedback loop, the faster we can iterate over design changes, and the more confident we can feel about our code. The sooner there's a problem, the easier and less expensive the fix is, because our brain is still engaged with the problem and we recall the shape of the code.

> To improve our software development we need rapid feedback, to learn of problems as soon as they appear. Good testing strategies provide short feedback loops.

Manual tests (either performed by a QA team, or by the programmers inspecting their own handiwork) are laborious and slow. To be at all comprehensive, it requires many individual steps that need repeating each time you make a minor adjustment to the code.

But hang on, isn't repeated laborious work something that computers are good at? Surely we can use the computer to run the tests for us automatically. That speeds up the running of the tests, and helps to close the feedback loop further.

Automated tests with a short feedback loop don't just help you to develop the code. Once you have a selection of tests, you needn't throw them away. Stash them in a test pool, and keep running them. In this way your test code works like a canary in a mine – signalling any problem before it becomes fatal. If in the future someone (even you on a bad day) modifies the code to introduce errant behaviour (a functional *regression*), the test will point this out immediately.

Code that tests code

So the ideal is to automate our development testing as much as possible: *work smarter, not harder*. Your IDE can highlight syntax errors as you type – wouldn't it be great if it could show you test breakages at the same speed?

Computers can run tests rapidly and repeatedly, reducing the feedback loop. Although you can automate desktop applications with UI testing tools, or use browser-based technology, most often development tests see the coder writing a programmatic test scaffold that invokes their production code (the SUT: *System Under Test*), prodding it in particular ways to check that it responds as expected.

We write code to test code. Very meta.

Yes, writing these tests takes up the programmer's precious time. And yes, your confidence in the code is only as good as the quality of the tests that you write. But it's not hard to adopt a test strategy that improves the quality of your code and makes it safer to write. This helps *reduce* the time it takes you to develop code: *more haste, less speed*. Studies have shown that a sound testing strategy substantially reduces the incidence of defects. [1].

It is true that a test suite can slow you down if you write brittle, hard to understand tests, and if your code is so rigid that a change in one method forces a million tests to be re-written. That is an argument against *bad* test suites, not against testing in general (in the same way that bad code is not an argument against programming in general).

Who writes the tests?

In the past some have argued for the role of a dedicated 'unit-test engineer' who specialises in verifying the code of an upstream programmer. But the most effective approach is for the programmers themselves to write their own development tests.

After all, you'd be testing your code as you write it, anyway.

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@goodliffe.net or @petegoodliffe

We need tests at all levels of the software stack and development process. However, programmers particularly require tests at the smallest scope possible, to reduce the feedback loop and help develop high-quality software as quickly and easily as possible.

Types of tests

There are many kinds of tests, and often when you hear someone talk about a 'unit test' they may very likely mean some other kind of code test. We employ:

Unit tests

Unit tests specifically exercise the smallest 'units' of functionality *in isolation*, to ensure that they each function correctly. If it's not driving a single unit of code (which *could* be one class or one function) in isolation (i.e., without involving any other 'units' from the production code), then it's not a unit test.

This isolation specifically means that a unit test will not involve any external access: no database, network, or filesystem operations will be run.

Unit-test code is usually written using an off-the-shelf 'xUnit' style framework. Every language and environment has a selection of these, and some have a de facto standard. There's nothing magical about a testing framework, and you can get a long way writing unit tests with just the humble `assert`. We'll look at frameworks later.

Integration tests

These tests inspect how individual units integrate into larger cohesive sets of cooperating functionality. We check that the integrated components glue together and interoperate correctly.

Integration tests are often written in the same unit test frameworks; the difference is simply the scope of the system under test. Many people's 'unit tests' are really integration-level tests, dealing with more than one object in the SUT. In truth, what we call this test is nowhere near as important as the fact that the test exists!

System tests

Otherwise known as *end-to-end* tests, these can be seen as a specification of the required functionality of the entire system. They run against the fully integrated software stack, and can be used as acceptance criteria for the project.

System tests can be implemented as code that exercises the public APIs and entry points to the system, or they may drive the system from outside using a tool like Selenium, a web browser automator. It can be hard to realistically test all of an application's functionality through its UI layer, in which case we employ *subcutaneous tests* that drive the code from the layer just below the interface logic.

Because of the larger scope of system tests, the full suite of tests can take considerable time to execute. There may be much network traffic involved or slow database access to account for. The set-up and tear-down costs can be huge to get the SUT ready to run each system test.

Each level of developer tests establishes a number of facts about the SUT, and constructs a series of *test cases* that prove that these facts hold.

There are different styles of test-driven development. A project can be driven by a unit-test mentality: where you would expect to see more unit tests than integration tests, and more integration tests than system tests. Or it may be driven by a system-test mentality: the reverse, with far fewer unit tests. Each kind of test is important in its own right, and all should be present in a mature software project.

When to write tests

The term TDD (that is, *test-driven development*) is conflated with *test-first* development, although there really are two separate themes here. You can 'drive' your design from the feedback given by tests without religiously writing those tests first.

However, the longer you leave it to write your tests, the less effective those tests will be: you'll forget how the code is supposed to work, fail to handle edge cases, or perhaps even forget to write tests at all. The longer you leave it to write your tests, the slower and less effective your feedback loop will be.

The test-first 'TDD' approach is commonly seen in XP circles. The mantra is: *don't write any production code unless you have a failing test*. The test-first TDD cycle is:

- Determine the next piece of functionality you need. Write a test for your new functionality. Of course, it will fail.
- Only then implement that functionality, in the simplest way possible. You know that your functionally is in place when the test passes. As you code, you may run the test suite many times. Because each step adds a small new part of functionality, and therefore a small test, these tests should run rapidly.
- This is the important part that's often overlooked: now tidy up the code. Refactor unpleasant commonality. Restructure the SUT to have a better internal structure. You can do all this with full confidence that you won't break anything, as you have a suite of tests to validate against.
- Go back to step 1 and repeat until you have written passing test cases for all of the required functionality.

This is a great example of a powerful, and gloriously short, feedback loop. It's often referred to as the *red-green-refactor* cycle in honour of unit-test tools that show failing tests as a red progress bar, and passing tests as a green bar.

Even if you don't honour the test-first mantra, keep your feedback loop short and write unit tests during, or very shortly after, a section of code. Unit tests really do help 'drive' our design: not only does it ensure that everything is functionally correct and prevent regressions, it's also a great way to explore how a class API will be used in production – how easy and neat it is. This is invaluable feedback. The tests also stand as useful documentation of how to use a class once it's complete.

Write tests *as you write* the code under test. Do not postpone test writing, or your tests will not be as effective.

This test-early, test-often approach can be applied at the unit, integration, and system level. Even if your project has no infrastructure for automated system tests, you can still take responsibility and verify the lines of code you write with unit tests. It's cheap and, given good code structure, it's easy. (Without good code structure, an attempt to write a test will help drive you towards better code structure.)

Another essential time to write a test is when you have to fix a bug in the production code. Rather than rush out a code fix, first write a failing unit test that illustrates the cause of the bug. Sometimes the act of writing this test serves to show other related flaws in the code. Then apply your bugfix, and make the test pass. The test enters your test pool, and will serve to ensure that the bug doesn't reappear in the future.

When to run tests

You can see a lot by just looking.
~ Yogi Berra

Clearly, if you develop using TDD, you will be running your tests *as* you develop each feature to prove that your implementation is correct and sufficient.

But that is not the only life of your test code.

Add both the production code *and* its tests to version control. Your test is not thrown away, but joins the suite of existent tests. It lives on to ensure that your software continues to work as you expect. If someone later

modifies the code badly, they'll be alerted to the fact before they get very far.

All tests should run on your build server as part of a *continuous integration* toolchain. Unit tests should be run by developers frequently on their development machines. Some development environments provide shortcuts to launch the unit tests easily; some systems scan your filesystem and run the unit tests when files change. However, I prefer to bake tests right into the build/compile/run process. If my unit-test suite fails, the code compilation is considered to have *failed* and the software cannot be run. This way, the tests are not ignorable. They run *every* time the code is built. When invoked manually, developers can forget to run tests, or will 'avoid the inconvenience' whilst working.

Injecting the tests directly into the build process also encourages tests to be kept small, and to run fast.

> Encourage tests to be run early and often. Bake them into your build process.

Integration and system tests may take too long to run on a developer's machine every compilation. In this case, they may justifiably run only on the CI build server.

Remember that code-level, automated testing doesn't remove the need for a human QA review before your software release. Exploratory testing by real testing experts is invaluable, no matter how many unit, integration, and system tests you have in place. An automated suite of tests avoids introducing those easily fixable, easily preventable mistakes that would waste QA's time. It means that the things the QA guys do find will be *really* nasty bugs, not just simple ones. Hurrah!

> Good development tests do not replace thorough QA testing.

Next time

In the next instalment, we'll look at *what* should be tested, what a (good) test looks like, and how we structure tests.

See you next time. ■

Questions

- How many styles of testing have you been exposed to?
- Which is the best development test technique: test-first, or test (very shortly) after coding? Why? How has your experience shaped this answer?
- Is it a good idea to employ a specialist unit-test writing engineer to help craft a high-quality test suite?
- Why do QA departments traditionally not write much test code, and generally focus on running through test scripts and performing exploratory testing?

Reference

[1] David Janzen and Hossein Saiedian, 'Test-Driven Development Concepts, Taxonomy, and Future Direction,' *Computer* 38:9 (2005).

Programmers' Puzzles

Francis Glassborow reviews his last challenge and presents a new one.

As any magazine editor (commercial or otherwise) knows, the number of responses to a competition is a small fraction of those who tried it. Readers often try competitions and even finish them but choose not to send in their solutions. I know how many of the *New Scientist*'s puzzles I solved – about 50% in the days when they ran a regular one – yet I never sent in one of my solutions. I also regularly tackle the Bridge competitions in the *English Bridge Union* magazine but never submit an answer. That behaviour makes me depressingly normal.

My first challenge required you to find some difference between C and C++ that could be exploited to switch the behaviour of a program. The difference could manifest at any stage from pre-processor through to run time behaviour.

I had in mind many little differences that might be exploited. In a way the more interesting ones are those that can trap programmers writing real code. A couple of quick examples:

The way in which the struct keyword introduces a name. C, for reasons that will seem strange to modern programmers, has a completely separate namespace (do not confuse with the C++ keyword **namespace**) for typenames created by the keywords struct and union. That is the reason that portable code (code that will necessarily behave the same way both as C and C++) couples a **typedef** with **struct** in the idiom:

```
typedef struct A {
  // declarations
} A;
```

A C compiler distinguishes between the plain name **A** and the elaborated name **struct A**.

Here is an example from James Holland.

> If I understand Francis's challenge correctly, what is needed is some source code that will produce two different outputs depending on whether the code was compiled by a C compiler or a C++ compiler. I assume using built-in compiler macros is not allowed! The solution must, therefore, rely on the code behaving differently depending on which compiler is used. My solution makes use of the fact that a C++ compiler enters the name of a struct in the scope in which it is declared. A C compiler does not do this. The following code [in Listing 1] makes use of this feature.
>
> When evaluating **sizeof(T)**, a C compiler will not see **T** as being the name of the **struct** as it is not within scope but will see the global **typedef** and conclude that **sizeof(T)** is 1. The body of the **if** statement will, therefore, not be executed. A C++ compiler, on the other hand, will see the locally declared **struct** and conclude that **sizeof(T)** is 2. The **if** statement can then be used to realise the different behaviour as required.

I will deal with James' assumption about the allowability of built-in compiler macros later. However, there is a flaw in James' code that means that it will often fail to work as expected because compilers are not prohibited from adding padding at the end of a **struct** so his test for equality with 2 will often fail because the compiler (usually for alignment purposes) may have added space at the end of **T**. Existing compilers will frequently return 4 or 8 for the **sizeof T** (the **struct** version).

Listing 1

```
#include<stdio.h>
typedef char T;
int main()
{
  struct T {char c[2];};
  printf("I was compiled by a C");
  if (sizeof(T) == 2)
    printf("++");
  printf(" compiler.\n");
}
```

```
if( sizeof (T) == 2)
```

needs to be replaced by

```
if (sizeof(T) > 1;
```

Apart from the flaw, the idea will work for any type, not just **char**.

Note that James avoided the flawed use of:

```
sizeof (char) == sizeof 'a';
```

as a test. This will usually work because character literals are of type **int** in C and **char** in C++. However, some compilers (largely for DSPs) use the same storage allocation for **int** and **char** types.

The other aspect is that James assumed that built-in compiler macros were not allowed. In these challenges, anything not explicitly excluded is allowed. Every C++ compiler is supposed to have **__cplusplus** as a built-in macro. This is absolutely essential so that code can test which version of C++ is in use. I leave it to the reader to surf the net to discover what values are required for conforming C++ compilers. Non-conforming C++ compilers will normally provide a value for **__cplusplus** but one that is not one of the standard values.

Hubert Matthews exploited this in the first of offering in his submission:

> Francis threw out a challenge for pieces of code that produce different results when compiled as C or as C++. Here are two: one cheaty [Listing 2] and one sneaky [Listing 3].

Listing 2

```
// prints "C" when compiled as C and "C++"
// when compiled as C++

include <stdio.h>
int main()
{
   puts("C"
#ifdef __cplusplus
   "++"
#endif
 );
}
```

I do not think that is a cheat. It shows a grasp of what is provided by the C++ Standard but I think he did not fully exploit the potential of the pre-processor to bizarrely alter the behaviour of code.

Using **__cplusplus** gives us all kinds of potential. Instead of conditionally adding a couple of characters at compile time, we can define complicated macros. We even have the ability to redefine a C++ keyword that is not also a C keyword for when our code is being compiled as C, without stepping into undefined territory. For example something such as:

FRANCIS GLASSBOROW

Since retiring from teaching, Francis has edited C Vu, founded the ACCU conference and represented BSI at the C and C++ ISO committees. He is the author of two books: *You Can Do It!* and *You Can Program in C++*.

```
#ifndef __cplusplus
#define try struct A
#endif
```

will work if we can design code that is a `try` block in C++ but a `struct` definition in C. That is just an idea for the reader to think about (replacing keywords, not specifically replacing `try`).

Hubert's second offering is interesting.

I have not been able to check that `auto x` without an explicit `int` specifier is allowed in current versions of C (post the abolition of 'implicit `int`'). GCC in C mode gives me a warning that `auto x` defaults to `auto int x`, which is sane even if it is not required that strictly conforming C makes the `int` explicit.

I leave it as an exercise for the reader to work out why this code produces different output. (Hint, you need C++11 or later compiler to reliably see the difference).

My choice of winner of this challenge is Hubert because his use of `auto` surprised me and highlighted another potential trap for the unwary C programmer accidentally using a C++ compiler. Of course, no reasonable C programmer uses `auto` and I suspect quite a number do not even know it is a C keyword.

Listing 3

```
// prints "12" when compiled as C
// and "13" when compiled as C++

#include <stdio.h>
int main()
{
  auto x = 5.6, y = 7.5;
  printf("%d\n", (int) (x+y));
}
```

Challenge 2

In the early days of computing, a frequent task was to write code with a missing instruction (such tests were even frequently set in Computer Science A level papers of the early 1980s).

I think it is time to revive this kind of mind exercise. As an example to get you on track, suppose that you have no `+` available, you could write

```
template <typename T>
T add (T a, T b){
  T result = -b);
  return  a -result;
};
```

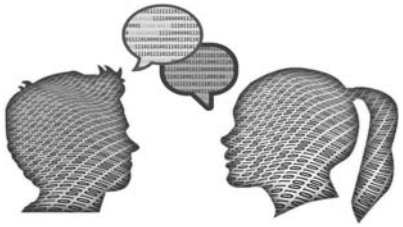
which will work for any type that supports minus as an inverse operation to plus. You would then need to specialise for other types such as `std::string`. That would be an interesting challenge in itself but it is not what I am going to set you.

The challenge is to write code that will assign the sum of two integer values, `a` and `b` and store the result in `c` without using the `=` symbol in your code. There are several simple solutions. Bonus points for multiple solutions.

You are restricted to C and C++ because those are the languages that I am familiar with (well you could try it in Forth, Prolog or Snobol?) ■

Code Critique Competition 109

Set and collated by Roger Orr. A book prize is awarded for the best entry.



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

Note: If you would rather not have your critique visible online, please inform me. (Email addresses are not publicly visible.)

Last issue's code

I've got a problem with an extra colon being produced in the output from a program. I've stripped down the files involved in the program a fair bit to this smaller example. Here is what the program produces:

```
test_program Example "With space"
1:: 1001:Example
2:: "1002:With space"
```

I can't see where the *two* colons after each initial number come from as I only ask for *one*.

Please can you help the programmer find the cause of their problem and suggest some other possible things to consider about their program.

- Listing 1 contains `record.h`
- Listing 2 contains `record.cpp`
- Listing 3 contains `escaped.h`
- Listing 4 contains `test_program.cpp`

Listing 1

```
namespace util
{
  class Record {
  public:
    Record(uint64_t id,
      std::string value = {});
    std::string to_string() const;
    // other methods elided
  private:
    uint64_t const id;
    std::string value;
  };
  inline
  std::string to_string(Record const &r)
  {
    return r.to_string();
  }
}
```

Listing 2

```
#include <cstdint>
#include <sstream>
#include <string>
#include "record.h"
util::Record::Record(uint64_t id,
  std::string value)
  : id(id), value(value) {}
std::string util::Record::to_string() const
{
  std::ostringstream oss;
  oss << id << ":" << value;
  return oss.str();
}
```

Listing 3

```
#pragma once
#include <string>

namespace util
{
  // provide 'escaped' textual representation
  // of value
  // - any double quotes need escaping with \
  // - wrap in double quotes if has any spaces
  template <typename T>
  std::string escaped_text(T t)
  {
    using namespace std;

    auto ret = to_string(t);
    for (std::size_t idx = 0;
      (idx = ret.find(idx, '"')) !=
        std::string::npos;
      idx += 2)
    {
      ret.insert(idx, "\\", 1);
    }
    if (ret.find(' ') != std::string::npos)
    {
      ret = '"' + ret + '"';
    }
    return ret;
  }
}
```

Listing 4

```
#include <cstdint>
#include <iostream>
#include "record.h"
#include "escaped.h"

using namespace util;
template <typename K, typename V>
void output(K key, V value)
{
  std::cout << escaped_text(key) << ": "
    << escaped_text(value) << '\n';
}
int main(int argc, char **argv)
{
  static uint64_t first_id{1000};
  for (int idx = 1; idx != argc; ++idx)
  {
    Record r{++first_id, argv[idx]};
    output(idx, r);
  }
}
```


ROGER ORR

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

Critiques

Paul Floyd <paulf@free.fr>

The issue presented in this Code Critique all centres around one line.

```
auto ret = to_string(t);
```

This is called from templatized **escaped_text**. This is itself called, with both of the templatized arguments, from the double templatized **output**.

This means that **escaped_text** gets instantiated with **t** having type **int** and also with **t** having type **Record**. What the author of the code probably intended was for the **int** specialization to call **std::to_string** and for the **Record** specialization to call **util::to_string**. The arguments match so it should work, right? No, wrong. This is not the full picture of how function call overload resolution works.

I won't go into the whole gory details (though if you are interested, almost everything I ever learnt about overload resolution I got from a set of fantastic videos by Stephan T. Lavajev on the Microsoft Channel 9 site [1], and there are the usual cppreferences [2] [3]). In short, overload resolution performs the following steps:

- Name lookup to get the candidate functions. This can involve argument dependent lookup and template argument deduction.
- Check for validity (correct number of arguments)
- Select the best match
- Member access check (e.g, public or private)

As you can see, name lookup occurs before selecting the best match. In this case it is 'unqualified lookup' since there is no scope operator indicating which scope to search for **to_string**. The lookup searches scopes from the current to the global scope until it finds one or more names. Once it has found a name in a scope it stops.

Obviously the lookup finds **util::to_string** which is in the same namespace as the call. But what about **std::to_string**? From the cppreference description:

> For the purpose of unqualified name lookup, all declarations from a namespace nominated by a using directive appear as if declared in the nearest enclosing namespace which contains, directly or indirectly, both the using-directive and the nominated namespace.

The nearest enclosing namespace that contains both **std::** and **util::** is the global namespace, i.e., **std::to_string** is considered to be in the global namespace. This means that the lookup stops in the **util** namespace and only considers **util::to_string**.

Unfortunately, **Record** has a non-explicit constructor that can convert from **uint64_t**

```
Record(uint64_t id,
  std::string value = {});
```

This means that for the second call to **escaped_text**, a **Record** is passed and it produces a string of the format **"id:name"**. For the first call to **escaped_text**, instead of **std::to_string** generating the string representation of the index, the index is implicitly converted to a **Record** with a default value for the name, the empty string. This produces the output **"index:[empty_string]"** or just simply **"index:"**. And here we have the extra colon.

To fix the code, it would be possible to remove the **util** namespace so that the overload resolution really does take place between **std::to_string** and **util::to_string**. I don't like that, and I'd rather suggest just simplifying output to call **std::to_string** and also to make the **Record** constructor **explicit**.

References

[1] https://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Core-C-

[2] http://en.cppreference.com/w/cpp/language/lookup

[3] http://en.cppreference.com/w/cpp/language/overload_resolution

James Holland <James.Holland@babcockinternational.com>

The student should not be surprised that two colons appear in the output. The first colon comes from **Record**'s member function **to_string()**. The second colon comes from the **output()** function called directly from **main()**. If only one colon is required, I suggest removing the one in **output()**.

Although the code now works correctly with the supplied examples, there are some features that need attention. From reading the code it is clear that should a supplied argument contain a quotation mark, a backslash is to be inserted just before it. It is a pity that this feature has not been tested, as far as is known, because there is a flaw in the mechanism that is responsible for doing this. It transpires that the arguments of **find()** (within **escaped_text()**) are swapped. The first parameter of **find()** is of type **char**, the second is of type **size_t**. The function call should, therefore, be **ret.find('"', idx)**. The student's code is attempting to find a **NUL** character starting at character position 34 (the ASCII code for the quotation mark character) as opposed to a quotation mark starting at position 0. It is, perhaps, unfortunate that the compiler is willing to cast a **size_t** to a **char** and a **char** to a **size_t** without comment. Incidentally, it is my understanding that starting a search beyond the end of the string is permissible, **find()** will simply return **std::string::npos** to indicate that the character was not found. Testing code to show that it meets its requirements is always important.

Some other features of the student's code that are immediately apparent include the following.

- It should be noted that if **using namespace std** is used, as in **escaped_text()**, there is no need to make explicit references to that namespace.
- The include guard **#pragma once** is not standard C++. **#ifndef** should be used, as shown below, to be standard compliant.

```
#ifndef <file name>_H
#define <file name>_H
// ...
#endif
```

- The header file, `record.h` is missing any form of include guard.
- The function **void output(K key, V value)** does not need to be a template as the types are always **int** and **Record** respectively.
- There is no need to make the variable **first_id** static.
- The constructor of **Record** need not have a default value for its second parameter.
- **Record**'s non-const member function **to_string()** is not used.

I am sure there are many other improvements to be made but I think this is enough to be going on with.

Jason Spencer <contact+pih@jasonspencer.org>

The reason for the two colons being printed is to do with function name lookup of **to_string** in **escaped_text**.

The output on each line is **"<okey>: <ovalue>"** (as per **output<K,V>** in `test_program.cc`), where **ovalue** is what is returned by **Record::to_string(value)** (via **util::to_string** via **escaped_text**) (after some formatting changes to make it 'escaped' if necessary). What is printed for **okey**, on the other hand, is expected by the student to be the result of **std::to_string(unsigned long long int)** (via **escaped_text**), which is expected to be chosen since **T = uint64_t** in **escaped_text<T>(T t)**, and probably the reason why **using namespace std** was included.

However, due to the function lookup rules **util::to_string** is first checked as a suitable function, and is found to be suitable as **uint64_t** can be implicitly converted to a **Record** thanks to the **Record ctor**'s default second argument. The output is therefore:

```
<id_of_okey>:<value_of_okey>:
<id_of_ovalue>:<value_of_ovalue>
```

value_of_okey is an empty string as per the default second argument in the **ctor**, and the printed output is now

```
<id_of_okey>:: <id_of_ovalue>:<key_of_ovalue>
```

The solution is discussed below.

There are also other issues with the program:

- in record.h
  - **#include <cstdint>** and **<string>** are missing. They're in record.cc, and record.h is included after they are, but they're still be needed in **record.h** for when **Record** is used elsewhere and later linked with the object file generated from Record.cpp.
  - The file is missing an include guard or #pragma once
  - Unless you have a *really* good reason to keep it, the default value for second arg in the **Record** class constructor should be dropped or the constructor should be marked explicit, otherwise there's a high risk of implicit conversion from **int**. I don't quite see a use case where a key:value pair is created and the value is assumed to be some default (ie blank).
- in record.cc
  - looks fine, but perhaps consider making **id** and **value** arguments const in the constructor definition – they're copied anyway to the member variables.
- in escaped.h
  - consider renaming **escaped_text** to **as_escaped_string**? **escaped_text** sounds like a variable, and text can take many forms. The return type is string.
  - input argument **t** should be **const** reference because **T** may not be copyable, or expensive to copy, or a temporary.
  - I'm strongly against doing the conversion to string and escaping in a single function – this function should only do the escaping and the conversion can be done in **output(...)** in test_program.cpp
  - To fix the headline problem of the double colon, **using namespace std** should be **using std::to_string**. This will add **std::to_string** to the list of candidates when doing the unqualified function name lookup of **to_string**. I'll come back to the lookup later.
  - **ret.find(idx, '"')** has the arguments the wrong way around, but because a **char** is implicitly convertible to **int**, and **int** to **char**, there is no error. The easy way to remember the order of the arguments is to remember that the thing you are searching for is not optional and has no obvious default value, but the start position is and does (the default could be the start of the string), so it must come as the latter argument.
  - Maybe I'm nitpicking but consider using **std::string::size_type** over **std::size_t** for the type of **idx** as that is the exact type returned by **find**, and required by **std::string::insert**. While it's almost definitely a **typedef** for **size_t**, **std::string::size_type** is there for a reason.
  - The use of **+= 2** in the **for** loop is brittle – it's pretty much a magic number (a hard coded pre-calculated value without context or adaptability to other changes). If someone changes the escaping char to a multiple char sequence and changes the third argument of the insert (the number of chars to insert from the second arg) to a value other than 1, then the **+=2** is wrong. Consider putting **const std::string escape_prefix { "\\" };** before the **for** loop and make the **for** step **1+escape_prefix.length()**, and the **insert** statement **ret.insert(idx, escape_prefix);** (ie the version that takes a string and no length argument).
  - The use of **ret.find(' ')** works for space, but not for all whitespace – there are other forms of whitespace including \t, \n, \r, in ASCII, as well as other characters in unicode. If the student would like to search for any of a list of chars in a string then **std::string::find_first_of** takes a null terminated string which is a list of chars to search for. Alternatively, use a regular expression and a character group, as described later.
- in test_program.cc
  - **output(K key, V value): key** and **value** should be passed by **const** reference, as we don't know whether the types are copyable.

    Additionally, we might like to avoid a potentially expensive copy by using a reference. The **const** is to alert us if we accidentally mutate the object and to allow temporaries to be passed. Consider passing an output stream as an argument so output could be sent elsewhere rather than the hard coded **std::cout**. As later described the conversion of types **K** and **V** should not be done in **escaped_text**, so should be done here instead. That also gives us control over formatting.
  - **main**: There isn't much point in declaring **first_id** static as **main** cannot be called by the program itself, so its value cannot be required to be maintained across calls. I can see two side-effects of declaring a local variable **static** in **main** – firstly, it is initialised before **main** is called, and secondly its storage is in the data segment rather than the stack. However, I cannot see any reason why any of these may be required, so I'd suggest the **static** modifier be removed. Since C++11, local static variables are also known as 'magic statics' and their initialisation is guaranteed to be thread-safe. But again, there's no good reason to do this in **main**.
  - **main**: perhaps rename **first_id** to **next_id**? Once you've incremented the value, it's no longer the first id.

Aside from the corrections above I'd recommend bigger changes to the code – and they are all to do with **escaped_text**.

Firstly, move the conversion of type **T** to **std::string** out of **escaped_text**. Let **escaped_text** have one responsibility **escaped_text** escaping the text. The conversion can then be much more flexible. Consider using a **std::stringstream** and the stream operators for the conversion, rather than **to_string**, as they are supported by more STL types (e.g. **std::bitset**), and more likely to be supported by user defined types.

The output of **std::to_string** can also often be unexpected – usually in terms of precision. Streams also have I/O manipulators, so there's a lot of flexibility in formatting (the base of output integers, capitalisation, number of decimal points, padding). On the subject of I/O manipulators, there's actually one in C++14/17 that will do the escaping for us:

```
template <typename K, typename V> void
output(const K & key, const V & value) {
  using std::to_string;
  std::cout << std::quoted(to_string(key))
    << ": " << std::quoted(to_string(value))
    << '\n';
}
```

Note however, that **std::quoted** doesn't return an **std::string**, so it isn't exactly equivalent, but it is designed to be very efficient, and it can also be used on input streams to unquote an incoming string.

Secondly, the original **escaped_text** has a worst-case time complexity of O(N^2). The loop iterates through the entire input string, and potentially, if the string were composed entirely of double quotes, could call **std::string::insert(...)** that many times, which itself is linear in time complexity wrt string length. This gets worse when you consider **insert** might trigger a dynamic memory allocation in the string. This probably isn't too bad if the string is small (and fits within the SSO [1] buffer), but if it is expected to be long then a two-pass algorithm could be considered:

1) First count the number of double quotes appearing in the input string (as `unsigned number_of_quotes`), and test for the presence of spaces (as `bool has_whitespace`)

2a) Create a new empty string that reserves `input_string_length + number_of_quotes + (has_whitespace ? 2 : 0)` chars.

2b) Iterate over the input string again, copying chars to the new string and inserting quotes and slashes as required.

This in now linear in time and should have at most a single allocation.

A sample implementation might look like:

```
std::string escaped_text(
  const std::string & in) {
  unsigned number_of_quotes = 0;
  bool has_whitespace = false;
  const std::string escape_prefix { "\\" };
  for ( const char c : in ) {
    if(c=='"') ++number_of_quotes;
    if(c==' ') has_whitespace = true;
  }
  std::string out;
  out.reserve( in.length()
    + escape_prefix.length()*number_of_quotes
    + (has_whitespace?2:0) );
  if ( has_whitespace ) out.push_back('"');
  for ( const char c : in ) {
    if(c=='"') out.append ( escape_prefix );
    out.push_back(c);
  }
  if ( has_whitespace ) out.push_back('"');
    return out;
}
```

There are, of course other, more generic ways to do this (see regular expressions later), but this is an O(N) solution that has no C++11 requirements (aside from the range for loop which can be factored out).

There are a number of points of further investigation when considering the student's program:

1. Name lookup in C++

   Name lookup in C++ has a number of nuances depending on the exact calling conditions. The call to `to_string` here is an unqualified lookup – that is there is no scope specifier preceding it (ie class name or namespace). See basic.lookup.unqual in [2] for the details behind the selection criteria and namespace.udecl in [2] describes the use of `using` to bring `std::to_string` into the lookup space.

   Bear in mind also that you cannot just add your own `to_string` implementation of your UDT to the `std` namespace. The C++ specification allows adding specialisations of templates to the `std` namespace (e.g. `std::swap`, `std::hash`), but adding overloads of existing functions is considered undefined behaviour (see namespace.std in [2]). `std::to_string` is not a template but a series of overloads for arithmetic plain-old datatypes (`int`, `float` and variations).

2. Whitespace, escape sequences and regular expressions. In ASCII, there are more whitespace characters [3] than just space (' '), tab ('\t') and newline ('\n') – there are also the less commonly seen carriage return ('\r'), vertical tab ('\v') and formfeed ('\f'). UTF-8 and UTF-16 add a whole lot more. You could check for all known whitespace directly, or you could rely on a regex (regular expression)[4] class to match the whitespace (single byte char example below, use `std::wregex` instead to match `std::wstring` contents):

```
  std::string escaped_text(
    const std::string & in) {
    std::regex doublequote_regex(R(")");
    std::regex whitespace_regex("[[:space:]]");
    const char *
      prefix_pattern_with_backslash_fmt =
        "\\$&";
    std::string ret = std::regex_replace(in,
      doublequote_regex,
      prefix_pattern_with_backslash_fmt);
    if ( std::regex_search(ret,
      whitespace_regex) )
      ret = '"' + ret + '"';
    return ret;
  }
```

   This example uses ECMAScript[5] type regexes (the default type for C++ regexes), but there are other types[6].

   There are many ways to escape a string – which method is used typically depends on which elements of the char set are reserved or otherwise have a special meaning. You wouldn't want to see a newline in a URL, for example, or a colon, or a bell, null or otherwise unprintable character – but they could appear as part of the data in a GET HTTP request URL (whether you should include such data in a GET URL is a discussion for another time). The escaping being done by the student here is reminiscent of CSV escaping, but there is no formal spec for CSV escaping, and some may argue that this isn't proper CSV escaping; for example, a comma should also be treated as special. I'd argue all non-printable and whitespace characters should also be escaped, as well as the backslash.

   Irrespective of the type of escaping, regular expressions are a great way to match patterns in strings and potentially perform some operation on them.

   Have a look also at Boost.Xpressive for an even more sophisticated library of string manipulation. It allows lambdas to be called where a regex matches so you can do hex conversion, for example, and spaces can be URL encoded to %20.

3. Testing

   If I may be blunt: there are many gaping holes in the testing here. There's no test for the escaping of quotes in the input, for example, which would have caught the swapped args in `ret.find(idx, '"')` in the `for loop` in `escaped_text`. The functionality in the student's program has a relatively good separation, apart from the conversion being in `escaped_text`. This makes it prime for unit testing [7] [11], and the related Test Driven Development [8] [11]. Using the header-only Catch2 [9] testing framework it's trivial to check expected output against a given input:

```
#define CATCH_CONFIG_MAIN
#include <catch.hpp>
#include <string>

const char * no_changes_tests [] = {
  "",
  "abcdef", "a_b_c_d_e_f", "_a_b_c_d_e_f_",
  "123456", "1_2_3_4_5_6", "_1_2_3_4_5_6_",
  "a1b1c", "_a_1_b_1_c_",
  "_", "1_", "_1", "_1_", "__", "11"
  };
const char * quotes_added_tests [] = {
  " ", "  ", "   ",
  "a b c d e f", " a b c d e f ",
  "1 2 3 4 5 6", " 1 2 3 4 5 6 ",
  "a 1 b 1 c", " a 1 b 1 c", "a 1 b 1 c ",
  " a 1 b 1 c "
  };
TEST_CASE( "correctness", "[escaped_text]" ) {
  SECTION( "No change to string" ) {
    for ( const char * test_string :
      no_changes_tests )
        REQUIRE ( escaped_text(test_string) == \
          test_string );
}
```

```
  SECTION( "Quotes should be added" ) {
    for ( const char * test_string : \
      quotes_added_tests )
      REQUIRE ( escaped_text(test_string) == \
      std::string("\"") + test_string + '"' );
  }
  SECTION("Existing quotes should be escaped")
  {
    REQUIRE( escaped_text( R")" ) == \
      R"(\")");
    REQUIRE( escaped_text( R"")" ) == \
      R"(\"\")");
    REQUIRE( escaped_text( R(" ")" ) == \
      R"("\" \"")");
    REQUIRE( escaped_text( R"( ")" ) == \
      R"(" \"")");
    REQUIRE( escaped_text( R"(" )" ) == \
      R"("\" ")");
    REQUIRE( escaped_text( R"( " )" ) == \
      R"(" \" ")");
  }
}
```

For increased coverage, tests for non space whitespace escaping should also be added. Long string testing also, etc. Writing good tests can be somewhat of an art [11].

I think that's it for now. I hope I don't sound too nit-picky, but the compiler can mis-understand intent, and the CPU is unforgiving, and even if that's all fine, the user (API user or end user) *will* find a way of breaking or abusing things. There's also the whole broken windows argument [10].

References

[1] https://stackoverflow.com/questions/10315041/meaning-of-acronym-sso-in-the-context-of-stdstring/10319672#10319672

[2] The C++11 Programming Language standard – ISO/IEC 14882: 2011(E)

[3] https://en.wikipedia.org/wiki/Whitespace_character

[4] https://www.regular-expressions.info/

[5] http://ecma-international.org/ecma-262/5.1/#sec-15.10

[6] http://www.cplusplus.com/reference/regex/regex_constants/#syntax_option_type

[7] https://en.wikipedia.org/wiki/Unit_testing and https://martinfowler.com/bliki/UnitTest.html

[8] https://www.agilealliance.org/glossary/tdd/

[9] http://catch-lib.net/

[10] https://pragprog.com/the-pragmatic-programmer/extracts/software-entropy

[11] *The Clean Coder: A Code of Conduct for Professional Programmers* by Robert C. Martin ISBN 978-0137081073

Commentary

The presenting problem is caused by an interaction between two different things: C++ name lookup rules and implicit constructors.

The idiomatic way to ensure the correct version of swap, for instance, is used in a template is write code like this:

```
{
  using std::swap;
  swap(a, b);
}
```

As the critiques pointed out, this has the right characteristics when there is a **`swap`** visible in another namespace whereas **`using namespace std`** does not.

There is good reason for this: bringing in the whole namespace allows access to a potentially large number of identifiers and if the added symbosl were all added into the immediate scope there is a strong likelihood of accidentally hijacking a call to a function in the current namespace in favour of one in the referenced namespace that happens to be a better match. So the rule for using namespaces adds the names to an enclosing scope, where they will be found if necessary when no closer symbol matches.

The second problem that went into the issue experienced by the user was that a constructor with defaulted arguments can provide an implicit conversion from one type to another. This can easily be prevented by making this constructor **`explicit`** so preventing most of the places where an implicit conversion occurs – notably for function arguments. It is worth considering the habit of making at least single argument constructors **`explicit`** by default – some static analysis tools will recommend this automatically. (It can of course be argued that making the second argument to the **`Record`** object optional might not make sense from a design perspective, but single argument constructors are common.)

Header files generally ought to have some kind of include guard to prevent errors caused by duplicate definitions. While **`#pragma once`** is a non-standard pragma, as James pointed out, it is in practice supported by the vast majority of modern compilers and does have two main advantages over 'traditional' include guards when the environment allows its use. Firstly it avoids adding additional macro identifiers into the program's scope and secondly the macro identifiers used for include guards sometimes end up duplicating each other (or are used incorrectly), which can cause some very confusing problems.

One problem in the example that no-one commented on was that the eighth line in the file `escaped.h` contains a comment ending with a backslash, hence turning it and the following line into a multi-line comment. While benign in the current code, this does occasionally have the consequence of accidentally commenting out a line of code, for example:

```
// check for \
if (s.find('\\') != std::string::npos)
{
  // code
}
```

where the **`if`** statement is commented out by the trailing backslash and so the following code is executed unconditionally. (Fortunately, many IDEs will show the affected line in comment markup, giving a visual cue.)

The Winner of CC 108

I liked Paul's links to information about name lookup: both the introduction and the reference. There are a lot of useful resources available to help with C++ programming.

James made several simplification suggestions – such as not requiring the **`output`** function to be a template. It is often a good thing to pass through code, once it is believed to be functionally complete, and remove some of the 'cruft' that tends to creep in. One of the advantages of code reviews is that additional pairs of eyes, seeing the code for the first time, often notice little details like these which those familiar with code are no longer surprised by.

Jason also discusses a number of improvements to the program – including a small design change to separate the escaping of text and the conversion of the types to a string. This sort of low-level refactoring enables each piece of code to focus on a single task and, when used well, can result in code that is much simpler to understand and test.

There is a bit of an open question over the best type for argument passing. Is it best to pass by value or by **`const`** reference? As with many things in C++, 'it depends'. What are some of the issues we must consider? On the one hand, passing by value requires an accessible copy/move constructor and, depending on whether the argument is a temporary or not, may require an actual copy. On the other hand passing by value is simpler for scalar types and, when passing temporary objects, can end up being more efficient than passing the temporary by reference.

Overall the entrants found a lot of issues to discuss in a relatively small critique, but I think that Jason provided the best set of answer to this issue's problem.

Code Critique 109

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

I'm trying to write a very simple dice game where the computer simulates two players each throwing dice. The higher score wins and after a (selectable) number of turns the player who's won most times wins the game. (I'm going to make the game cleverer once it's working.) But the games always seem to be drawn and I can't see why. Here is what the program produces:

```
dice_game
Let's play dice
How many turns? 10
Drawn!
How many turns? 8
Drawn!
How many turns? ^D
```

Listing 5

```
// Class to 'zip' together a pair of iterators
template <typename T>
class zipit : public std::pair<T, T>
{
  zipit &operator+=(std::pair<int,int> const &rhs)
  {
    this->first += rhs.first;
    this->second += rhs.second;
    return *this;
  }
public:
  using std::pair<T, T>::pair;
  zipit &operator+=(int n)
  {
    return *this += std::make_pair(n, n);
  }
  zipit &operator-=(int n)
  {
    return *this += std::make_pair(-n, -n);
  }
  zipit &operator++()
  {
    return *this += 1;
  }
  zipit &operator--()
  {
    return *this += -1;
  }
  auto operator*()
  {
    return std::make_pair(
      *this->first, *this->second);
  }
  auto operator*() const
  {
    return std::make_pair(
      *this->first, *this->second);
  }
  // Hmm, operator-> ??
};
template <typename T>
auto begin(T one, T two)
  -> zipit<typename T::iterator>
{
  return {one.begin(), two.begin()};
}
template <typename T>
auto end(T one, T two)
  -> zipit<typename T::iterator>
{
  return {one.end(), two.end()};
}
```

What's going wrong, and how might you help the programmer find the problem? As usual, there may be other suggestions you might make of some other possible things to (re-)consider about their program.

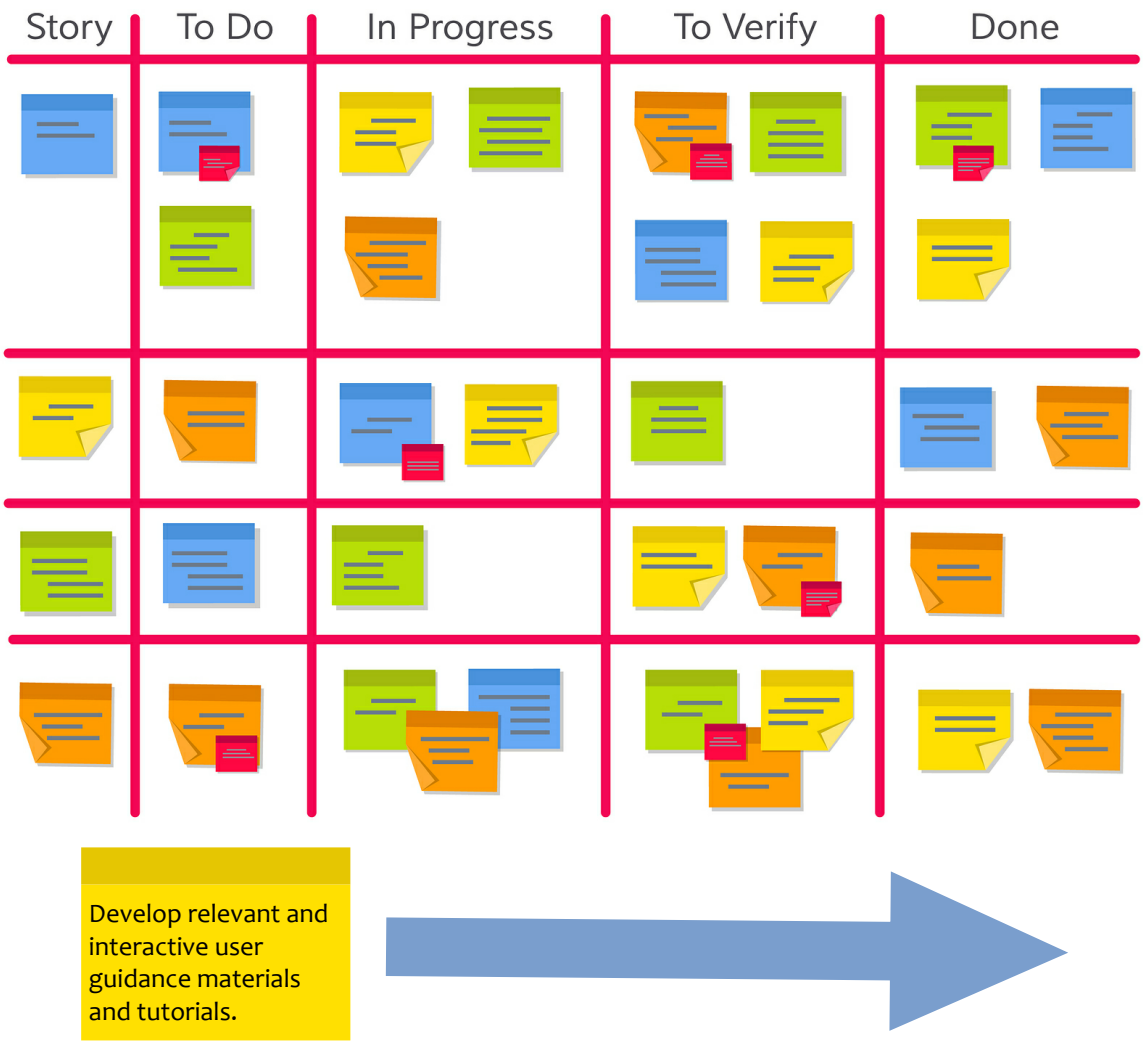
- Listing 5 contains `zipit.h`
- Listing 6 contains `dice game.cpp`

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

Listing 6

```
#include <algorithm>
#include <iostream>
#include <random>
#include "zipit.h"

class randomize
{
  std::mt19937 mt;
public:
  int operator()() { return mt() % 6 + 1; }
};
void play(int turns, randomize &generator)
{
  std::vector<int> player1(turns);
  std::vector<int> player2(turns);

  std::generate(player1.begin(),
    player1.end(), generator);
  std::generate(player2.begin(),
    player2.end(), generator);

  int total{};

  for (auto it = begin(player1, player2);
       it != end(player1, player2); ++it)
  {
    if ((*it).first != (*it).second)
    {
      auto diff = *it.first - *it.second;
      total += copysign(1.0, diff);
    }
  }
  if (total > 0)
  {
    std::cout << "Player 1 wins\n";
  }
  else if (total < 0)
  {
    std::cout << "Player2 wins\n";
  }
  else
  {
    std::cout << "Drawn!\n";
  }
}
int main()
{
  randomize generator;

  int turns;
  std::cout << "Let's play dice\n";
  while (std::cout << "How many turns? ",
    std::cin >> turns)
  {
    play(turns, generator);
  }
}
```

Wishing you a productive and prosperous 2018!



Why not make 2018 the year that you move developing a modern approach to user assistance from your 'To Do' list to 'Done'

Clearly stated...

If you need some help in developing a sustainable strategy, get in touch.

www.clearly-stated.co.uk

Standards Report

Emyr Williams updates us on the latest in C++ standardisation.

He's probably best known as 'the guy who interviews people for *CVu*', but he branched out by responding to a call for volunteers to be the new ACCU Standards Officer. Emyr is interested in C++, and as an attendee of the BSi C++ Panel in London, felt volunteering seemed the logical choice. He volunteered so he could learn more about C++, and how a language is put together; the role will also force him to be a better programmer, and gain a deeper knowledge of the language.

The last ISO C++ Committee meeting was held in Albuquerque, New Mexico for six days in November, hosted by the folks at Sandia National Laboratories, and by all accounts was quite a busy week with around 140 people in attendance representing 10 national bodies. [1] [2]

Allow me to start with a caveat: I found it's quite difficult to write a report for a meeting you didn't attend, but thanks to numerous blog posts and articles online, I'm able to provide something of a digest of the meeting. While I cannot comment on the atmosphere at the meetings, I can comment on what the outcomes were and where C++ is headed. Additional contributions were made to the report by Guy Davidson and Roger Orr, and are accredited accordingly.

As Roger mentioned in his previous standards report, the ISO voting was still ongoing for the draft International Standard for C++ 17; however, I'm happy to report that the draft was accepted, and that we do now in fact have C++ 17. Which is great news. In terms of compilers, the latest versions of both GCC and Clang have complete support for C++17, and MSVC expects to be feature complete by March 2018.

One of the primary goals of the meeting was to address the comments that the national bodies had sent in, in regards to the Modules TS comment ballot. These were addressed in a single meeting. The main area for additional work was when entities were implicitly exported from a module – for example return types of functions. The discussions on modules are still on going but it's hoped that it will make it in to C++ 20.

It was also the second time that changes to the current C++ 20 draft could be voted on. And some of the highlights include (in no particular order…):

Range-based for statements with an initializer (p0614r1)

This allows you to initialise an object within the parenthesis of the **`for`** loop [3]. The benefit is that it allows the developer to use locally scoped variables, so for example, you could do something like this:

```
for( T widgets = getWidgets() ;
auto& w :   widgets.items())
{
  // do stuff here…
}
```

Whereas before you'd have to do something like this:

```
{
  T widgets = getWidgets();
  for(auto& w : widgets.item()) {
    // it's worth noting that
    // for(auto & w : w().items())" is wrong
  }
}
```

This change makes the range-based **`for`** loop consistent with other control flow statements such as **`if`** and **`while`**, which gained the facility to contain variable initialization before their condition in C++ 17. A simple example would be if you had a vector that you wanted to populate then loop over, you could do it all in the parenthesis of the **`for`** loop.

Consistent comparison (spaceship operator) (p0515r3)

One of the more significant features voted in was Herb Sutter, Jens Maurer and Walter E. Brown's proposal for consistent comparison [4], which is also known by colloquially as the spaceship operator, or **`operator<=>`**.

There had been previous efforts and proposals to create a Consistent Comparison proposal which all served as the basis of the paper as proposed. The proposal sought to pursue three-way comparison, by allowing default copying to guide default comparison, and it would allow developers to write a memberwise comparison function body far easier than it is at present, and to enable more powerful and precise comparisons, with less code.

The paper gave two cases, the common case, where if you wanted to write all comparisons for your type X with type Y with memberwise semantics, all you needed to write would be :

```
auto X::operator<=>(const Y&) = default;
```

And that's it! Whereas previously you'd need to write:

```
class point{
  int x;
  int y;

public:
  friend bool operator==(const Point& a,
    const Point& b)
    { return a.x == b.x && a.y == b.y; }
  friend bool operator< (const Point& a,
    const Point& b)
    { return a.x <b.x ||
      (a,x == b.x && a.y < b.y);
    }
  // you'd still need to write another 4
  // of these operator overloads!
}
```

Herb Sutter has an excellent example on his trip write up comparing two case insensitive strings. [5]

Modules TS

At the last meeting (Toronto), the Modules TS was published and circulated for balloting, which is where national bodies could vote, and submit comments on the TS. The ballot did pass, but there were numerous technical comments that were worked through during the meeting. Progress was made, but not enough to publish a final TS at the end of the meeting. I believe there are teleconference meetings taking place over the coming few months to work on this, and an update will follow in due course.

There were a couple of planned TS's proposed as well, these do not have an official project or a working draft at the time of writing.

EMYR WILLIAMS

Emyr Williams is a C++ developer who is on a mission to become a better programmer. His blog can be found at www.becomingbetter.co.uk

Bookcase

The latest roundup of book reviews.

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

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

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.

Astrid Byro (astrid.byro@gmail.com)

Make good art

By Neil Gaiman, published by Headline in May 2013, ISBN-13: 978-1472207937

Reviewed by Ian Bruntlett

This book is brief, visually rich and full of wisdom, based on a speech successful author Neil Gaiman made to the Philadelphia University of the Arts in May 2012. That speech (just under 20 minutes) can be viewed online at this page – https://vimeo.com/42372767. Chip Kidd did the graphic design – in an attempt to look edgy graphically (e.g. white text on a pale blue background), it can be very difficult to read in dim light. Despite the speech being aimed at artists and writers, it still has good advice that can be of benefit to software developers, both at the start of a career and during it. It is a hardbacked book and looks like it could also have been entitled *The Ladybird Guide to being a Creative*. Kevlin Henney suggested I provide a favourite quote from the book. I went through the book again and I came up with seven. My favourite quote is at the start – "This book is for anybody who is looking around and thinking Now What?" – and it expands on that throughout the book. Overall I enjoyed this book and intend to have it on-hand for future reference.

Standards Report (continued)

Library

The Library Working Group (LWG) discussed how Concepts should be used in the C++ library. The consensus was that a full proposal would need to be seen at a future meeting before Concepts are to be used in future proposals.

The LEWG approved proposals which are mainly aimed at C++ 20, and were sent to the LWG for a wording review. These proposals included:

- `std::polymorphic_value<T>` (p0201r2)
- `<version>` (p0754r1)
- Calendars and Timezones (p0355r4)
- `std::hash_combine` (p0814r0)
- Bit operations (`rotr`, `popcount` etc) (p0553r2)
- Integral power-of-2 operations (p0556r2)
- Efficient access to basic `std::stringbuf`'s buffer (p0408r3)
- `std::bind_front` (p0356r3)
- `std::spanstream` (p0448r1)
- `.contains()` for `std::map` (p0458r0)

The following proposals were discussed and given design feedback and guidance from the group:

- `std::transform_if` (p0838r0): Brings the implementation of `transform_if` from boost (boost/compute/algorithm)
- `std::flat_map` (p0429r3)

  A more space/runtime efficient representation of a map structure. Commonly used in gaming, embedded or system software development. Its intention is that the `flat_map` is a drop-in replacement for `std::map` but with different time and space efficiency properties. It's primarily based on Boost's `flatMap`, and its API is nearly identical to `std::map`.
- `std::function_ref` (p0792r0)

  The idea behind `function_ref` is allow further functional programming idioms to be added to the language. 'Higher-order' functions are one of the key areas of this paradigm; essentially, they are functions that take functions as arguments, and can return functions as results. At present, the language doesn't support referring to an existing Callable object, or at least not flexibly at any rate. The proposal hopes to change that.
- `std::wide_int` (p0539R2)

  There's no cross-platform solution to have bigger numbers than `int64_t`. While there's the non-standard type `__int128` which is provided by GCC and clang, there is no other way to do this. The paper proposes a templated class where you can specify the size of integer you want: e.g.

  ```
  std::wide_uint<128> veryBigNumber;
  ```
- Endian support (p0803r0)

  At the moment, there's no standardised way to handle endianness in C or C++. Some platforms provide a mechanism for this in C, it varies between platforms. The paper was written to determine whether or not there was interest in adding this to the C++ STL or a library TS.

Details are available online of all the papers being discussed by the committee. [6]

References

[1] https://botondballo.wordpress.com/2017/11/20/trip-report-c-standards-meeting-in-albuquerque-november-2017/

[2] https://www.reddit.com/r/cpp/comments/7ca2sh/2017_albuquerque_iso_c_committee_reddit_trip/

[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0614r1.html

[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r3.pdf

[5] https://herbsutter.com/2017/11/11/trip-report-fall-iso-c-standards-meeting-albuquerque/

[6] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/#mailing2017-11

View from the Chair

Bob Schmidt
chair@accu.org

Committee spotlight

We have some upcoming changes to the makeup of the ACCU committee.

- Malcolm Noyes, ACCU's Secretary, has informed the committee that he will not be seeking re-election in 2018, after four years of service. If I recall correctly, Malcolm tried to step away from the role in 2016, but volunteered to continue as secretary when no one else stood for election that year, and then stood for re-election in 2017. Malcolm will finish his out his term ending at the AGM in April, 2018.
- Rob Pauer has announced that he would like to retire from his role as Treasurer. Rob has been treasurer since 2011, and has been retired from his career in insurance and pensions for several years. Rob would like to have a new treasurer shadow his activities for a few months in preparation for taking over full-time. (Rob has been a member of ACCU since 1987, and is believed to be the longest current member of ACCU.)
- Jonathan Wakely is stepping down from his role as Standards Officer. Jonathan recently became a father (congratulations Jon!), with all of the time constraints associated with parenthood.

Please join me in thanking Malcolm, Rob, and Jon for their contributions to ACCU.

Emyr Williams and Guy Davidson have volunteered to work together to be Standards officer. There are details remaining to be worked out, since Guy currently is serving his second year as Auditor, a position that is independent of the committee. Thank you to both of them for volunteering.

2018 Annual General Meeting

ACCU's 2018 Annual General Meeting (AGM) will be held on Saturday, April 14th, 2018, at the Marriott City Centre in Bristol, UK, in conjunction with the 2018 ACCU conference.

The important dates associated with the AGM are in the table below.

The most important part of our pre-AGM activities is election of officers. From ACCU's constitution [1]:

> 5.3.1 Members of the Committee shall hold office from the date of appointment until the next Annual General Meeting, and shall be eligible for re-election.
>
> 5.3.3 Any member of the Association can stand as a candidate for election to any role on the committee. Any such member shall notify the Secretary in writing (letter or email), including names of a nominating member and a seconder, on or before the Proposal Deadline (described in 'Section 7 – General Meetings'). The same person cannot stand for more than one role in the same election.

As of now, ACCU will have at least three critical committee positions to fill in April: secretary, treasurer, and auditor (and possibly a fourth, as I have not yet decided if I will stand for re-election as chair). Secretary and treasurer are elected each year; auditors volunteer for staggered two year terms.

ACCU has had trouble getting people to volunteer for positions in the organization. (In 2016, two executive committee roles had no one standing for election; in 2017, all incumbents ran unopposed. The position of Web Editor has been vacant for six months.) In this I suspect we are not much different from most volunteer organizations.

I get it. We all have other stuff to do. I myself have never been much of a joiner, let alone a volunteer. (You may recall, from my very first View, my wife's reaction to the news that I volunteered to serve on the committee – "she just laughed".) Nevertheless, ACCU needs people to fill these roles. Ideally we need multiple people standing for each role, to give substance to our elections.

If you would like to nominate someone for a role, or would like to volunteer to stand for election, please send an email to accu-committee@accu.org. Nominations are due by 13 February 2018.

AGM Deadlines

14 January 2018	Official Announcement Date	90 days prior to AGM
13 February 2018	Proposal and Nomination Deadline	60 days prior to AGM
3 March 2018	Draft Agenda	42 days prior to AGM
17 March 2018	Agenda Freeze	28 days prior to AGM
24 March 2018	Voting Opens	21 days prior to AGM

WG21 Albuquerque

Carter Edwards (sponsor of the meeting) and Herb Sutter (convener of WG21) graciously allowed ACCU to distribute copies of the most recent Overload and advertise the upcoming conference during the week. Thanks to both of them for their support.

International standards development fund

You may recall from my last column that the committee awarded an ISDF grant to Mr Walter Brown. I'm pleased to report that the grant allowed Mr Brown to travel to Albuquerque for the fall WG21 meeting, where I was honoured to make his acquaintance.

ACCU 2018

As mentioned above, the next ACCU conference will be held in Bristol, U.K., from the 11th through the 14th of April, 2018, with pre-conference workshops on April 10th. The conference again will be held at the Marriott City Centre, our home for the past several years.

Web site redesign

As announced last issue, we are soliciting ideas for a new platform for ACCU's website. As this issue goes to press we have not gotten any feedback. If you have experience with a content management platform and would like to express your opinion on it, please send your comments to accu-committee@accu.com.

Overload Reviewers

Fran Buontempo, editor of *Overload*, has announced additions to the makeup of the magazine's peer review team. Please join me in welcoming these new reviewers to the *Overload* team:

Kaartic Sivaramm	Craig Inches
Yubin Ruan	Tor Arve Strangeland
Balog Pal †	Araray Velho
Philipp Schwaha †	Gennaro Prota
Christopher Gilbert	Ben Curry †
Paul Johnson †	

† Have already commented on Overload 142 articles!

Finally, Fran has announced that Phil Bass has stepped down as a reviewer. On behalf of the committee I'd like to thank Phil for his service to *Overload* and ACCU.

Reference

[1] ACCU Constitution. https://accu.org/index.php/constitution