the magazine of the accu

www.accu.org

{cvu}

Volume 26 • Issue 6 • January 2015 • £3

{cvu}

Volume 26 Issue 6
January 2015
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Ian Bruntlett, Pete Goodliffe, Vassili Kaplan, Ralph McArdell, Chris Oldwood, Roger Orr, Mark Radford, Emyr Williams

ACCU Chair
chair@accu.org

ACCU Secretary
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

A Mutual Understanding

I was recently watching a television program about one of Britain's oldest cathedrals – Canterbury. Part of the program was dedicated to some serious repair work needed after some stone fell away from a high window. No-one was hurt (apparently), but investigations by the masonry experts revealed that this window had previously undergone refurbishment to strengthen it some centuries ago.

That historical repair had introduced a metal support into the stone which had the effect (at the time) of preventing further damage to the structure, and allowed repairs to be made. It also had the unintended side-effect of directly causing the damage hundreds of years later! As I understand it, the metal had flexed during a period of exceptionally warm weather, and split the stonework causing a serious fracture. I'm sure I don't need to draw a diagram to illustrate the parallels with modern software development: how often have you visited a piece of code and some part of it has made you say (perhaps only to yourself...) "What muppet wrote this rubbish?"

In the case of Canterbury Cathedral, the modern stone masons recognised that those historical refurbishers were operating on the best knowledge of the time, and probably had not considered that their work would even be standing centuries later, much less be the cause of new problems. The same is true of our own forebears in code, those plucky pioneers who dreamed it up, or the generations of intrepid adventurers who've fixed it, updated it, squeezed new performance out of it, bent it to new purpose over time. 'They' may even have been 'You' in those times (and yes, I've looked back at my own code and wondered at my own imbecility), and were themselves operating under their own best knowledge and intentions, and almost certainly oblivious of the potential longevity of their inventions.

We all – I think – believe in some way that the best of our code will live a long and fruitful existence and be easily updated and maintained in the future, but I don't think we generally put as much imagination into exactly *what* that might entail. And besides, we can't know what future techniques and methods will be commonplace to our successors, given how difficult predicting the future is.

And so, we are left only with the hope that those successors think kindly of us – as the modern stone masons at Canterbury were most impressed by the ingenuity of their own antecedents – and understand the constraints and limitations of our endeavours. Of course, we should be similarly forgiving of our own programming ancestors, rather than just referring to them as 'muppets'!

Steve

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 27.1:	1st February 2015
C Vu 27.2:	1st April 2015
Overload 126:	1st March 2015
Overload 127:	1st May 2015

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

Simplicity Through Immutability

Chris Oldwood considers the benefits of unchangeable data.

Some languages, mostly notably C++, provide support for marking instances of objects as immutable, i.e. with the **const** keyword. Whilst C# supports a notion of **const**, it only applies to primitive values and is essentially an alias for a literal value. The best you can do with objects is to apply the SINGLE ASSIGNMENT PATTERN [1] to member variables using the **readonly** keyword. Java's final keyword has a wider scope than C#'s **readonly** for enforcing single assignment, but it still falls short of enforcing mutability of the referenced object.

Consequently in many languages immutability must be implemented through the design of the type, and as such every instance of that type will be immutable (use of reflection and other Machiavellian techniques notwithstanding). Generally speaking to create immutable objects you only pass any state at creation time through the constructor. Any properties that are exposed must only be readable and immutability must be deep, meaning that any types used in any exposed collections must themselves be immutable too. As it is with turtles [2], it should be immutability all the way down.

The most common examples of immutable types are the primitive types, such as integers and strings (at least in C#). But more complex types can easily be made immutable too with just a little thought. The benefits, as you shall hopefully see, are a significant reduction in accidental complexity [3]. Whilst the implementation of the type itself may not contain significantly less lines of code, far fewer tests will be required and significantly less grey matter exercised to drive out the behaviours that matter.

The basis for this article is a real class I recently decided to refactor to make it simpler by making it immutable. Hence what follows below was my rationale for doing the refactoring in the first place.

The mutable example

Imagine you are writing a service and you have a need to provide a simple lookup to map one value to another. The data changes pretty infrequently, but not so infrequently that you're happy to hardcode it. You decide to store the mapping data in a simple .CSV file and write a class to load the data and provide the mapping at runtime. Listing 1 is one possible implementation.

Listing 1

```
public class ThingToWotsitMap
{
  public ThingToWotsitMap()
  {
    _map = new Dictionary<string, string>();
  }
  public void LoadMap(string filename)
  {
    _map.Clear();
    using (var stream =
       new StreamReader(filename))
    {
      // Parse .CSV file data and build map
      . . .
    }
  }
  public string LookupWotsit(string thing)
  {
    // Map thing to wotsit
  }
  private Dictionary<string, string> _map;
}
```

If you are already screaming at the page because you spotted that the code touches the file-system and is therefore difficult to unit test, then have a gold star. But you'll have to put that to the back of your mind as that is not what this article is about.

So many questions

The class is very simple, surely there's not much to say. Is there? Looking over the unit tests (which were written after) one pattern immediately leaps out – the use of two-phase construction:

```
[Test]
public void one_of_the_tests_for_the_map()
{
  var filename = . . .;

  var map = new ThingToWotsitMap();
  map.LoadMap(filename);

  Assert.That(. . .);
}
```

The solution is simple. All we need to do is to add a constructor that takes a filename and then delegates to **LoadMap()** for the heavy lifting, right? That's not the aspect of two-phase construction that bothered me; it was the fact that there was two-phase construction even to begin with. Adding another constructor does not take away the ability to mutate the class and it's that mutability that starts to raise a number of questions about the behaviour of the class.

The first question I have is: what happens if the client doesn't even call **LoadMap()**? Does the class behave correctly if no data is loaded? This immediately leads to the question about **LoadMap()** throwing an exception. Is the internal state consistent if that occurs, and what happens if the client discards the error? We are back to the first question again.

So off we go and merrily find out the answers to these questions and write a bunch more tests to make sure that the class is exception-safe and also behaves 'safely' should something happen during its (two-phase) construction.

Thread safety

Time passes and our semi-static data becomes not quite so static. We decide that restarting the service every time we need to change the data is untenable and would like to detect when the file changes and load it again automatically at some convenient moment.

So we add a bit of code externally to the class to watch for when the file changes and then just call **LoadMap()** to load the new data file.

But wait. Have we got any tests for calling **LoadMap()** on the class a second time? What happens if this load fails – is the internal structure still consistent, can we limp along with the old data or should it behave as if freshly constructed and therefore no data was loaded?

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

And all before we even get to questions about thread safety. This is a multi-threaded service, is this class thread safe? For example, what happens if one thread tries to lookup data whilst another is calling **`LoadMap()`** to refresh the data? Should existing callers be blocked and wait for the new data or continue with the old data until the new set is fully loaded?

Assuming you can answer all those questions, how confident are you in your ability to write thread-safe, exception-safe code? How confident are you that whatever techniques you've chosen to use will continue to be thread-safe as you move to new platforms in the future? Do you have adequate test coverage and/or documentation to show that you've even considered all these scenarios?

Refactoring to immutability

Let's wind the clock back to the beginning where we noticed the two-phase construction and see if we can take a different path that doesn't involve us writing so much production and test code, and lead to so many tricky questions too.

My personal preference is to create immutable types where possible. Essentially a type should start immutable by default and prove that adding mutability will provide some significant advantage – performance, perhaps – that could not be obtained via immutability. There are some very obvious cases, such as the BUILDER pattern [5], where mutability is required up front, but that can often be as a stepping stone to an immutable form of the same type.

The changes I made to the example code above were fairly minor. Naturally there are many other changes we could make and so the result below is not intended to be the final word on the matter, it is only intended to show the minor transformations that were needed to achieve immutability.

The first change I made was to make the constructor private and to take the underlying container as a constructor argument:

```
private ThingToWotsitMap(Dictionary<string,
   string> map)
{
  _map = map;
}
```

The second step was to make the **`LoadMap()`** method **`static`** and change it to return an instance of the class; essentially turning it into a FACTORY METHOD [6] (see Listing 2).

Listing 2

```
public void LoadMap(string filename)
{
  Dictionary<string, string> map =
     new Dictionary<string, string>();
  using (var stream = new StreamReader(filename))
  {
      // Parse .CSV file data and build map
      . . .
  }
  return new ThingToWotsitMap(map);
}
```

And that's it really. One final step was to fix up the tests and production code to use this new factory method instead of the previous two-phase construction approach (I could lean heavily on the compiler here due to the nature of the refactoring):

```
[Test]
public void one_of_the_tests_for_the_map()
{
  var filename = . . .;
  var map = ThingToWotsitMap.LoadMap(filename);
  Assert.That(. . .);
}
```

Revisiting those questions

With our new design in place let's revisit those earlier concerns and see how it stacks up. The first question around the behaviour of a default initialised object is moot because you cannot create one – you have to provide a filename. Likewise the second question around the exception safety of the **`LoadMap()`** method is also moot because if an exception is thrown you will not have a fully constructed object to worry about. Essentially the whole issue of state corruption is moot because the point of immutable types is that you can't change them.

So far so good, but what about our latter change in requirements where we need to reload the data on-the-fly when it changes whilst multiple threads might be accessing it? This is still largely a moot point because the type is immutable – you cannot change it, you can only *create a different object* with the new data in it.

From the perspective of the type itself there are no thread-safety issues, but that does not mean there are no thread-safety issues at all. Instead of putting all the effort into making the type internally thread-safe we have pushed the problem up to the owner, but their problem is almost trivial by comparison; the owner just needs to switch ownership of the object in a thread safe manner, e.g.

```
var newMap = ThingToWotsitMap.LoadMap(filename);
// This assignment needs to be thread-safe
_currentMap = newMap;
```

In certain environments a write of a reference-sized value is already an atomic operation and so out-of-the-box this could already be enough, depending on how it is used. It is more likely that you'll mark the member as 'volatile' to introduce the relevant memory barrier and to ensure that the value is not aggressively cached. There shouldn't be a need to use a heavyweight synchronization object like a mutex in this example as it's just a single reference, but if you have to switch multiple references atomically it might be required.

Thinking about the performance of these two approaches they should be fairly similar, with the potential for the immutable version to win on the basis of needing less synchronization. Memory-wise reloading the data should be similar in both cases too, i.e. having two copies in memory at the point just before the switchover. You could choose to empty the internal container first in the mutable case before loading the file, but then you'd have to sacrifice exception safety which feels like a decision that needs serious consideration. On the face of it we don't appear to have lost anything with our move to immutability.

Interlude: C# Initializer Syntax

Sadly there are some programming constructs that make the design of immutable types less alluring; one is the Initializer Syntax for objects and collections in C#. Both of these rely on the type being mutable, with state being mutated through properties for the former case and an **`Add()`** method for the latter.

Here is an example of the object initializer syntax:

```
var point = new Point { X = 1, Y = 2 };
```

Under the covers this is the same as writing:

```
var _point = new Point();
_point.X = 1;
_point.Y = 2;
var point = _point;
```

And this is the collection initializer syntax:

```
var points = new List<Point> { new Point(1, 2) };
```

This is the same as writing:

```
var _points = new List<Point>();
_points.Add(new Point(1, 2));
var points = _points;
```

Whilst we should not blame the language authors for providing us with a construct that allows more readable construction of objects, its over use possibly means mutability has become the *de facto* choice by accident. In

Advice for the Young at Heart

Pete Goodliffe offers sage advice, and asks you to do the same.

Not long after my latest book, *Becoming a Better Programmer*, was published I was contacted by a student programmer from the Philippines. He wanted to improve his software design and programming skills, and tracked down my contact details to ask for personal advice. His simple question was: how best to learn the art of programming without being on a Computer Science course, and without access to mentors within the industry.

It's a great question.

It's a great question, not just for the answer being sought, but because of the motivation for asking:

- If you're investigating how to get better, if you *care* about learning and improving, then the battle is won. Whilst apathy breeds sloppy careless coders, those who make a *conscious decision* to improve their skills will inevitably grow. Success stems from motivation.
- If you are prepared to actually *do something*, to reach out and contact others, to make some effort to improve, then the battle is won. Information won't come to you. It takes deliberate effort and investment to improve.

Are you putting in this deliberate effort, and committing yourself to learning and improving your craft?

This was my answer to the budding coder:

> If you're not on a Computer Science course, and don't have access to a mentor, then your best bet is to read widely, and practise, practise, practise. The more coding you do, the more mistakes you make, the more you'll learn. And the more fun you'll have coding, too. (The best graduate programmers have real experience under their belt, not just academic theory.)
>
> Grab some good general programming practice books (like mine, for example!), *Clean Code* (Bob Martin), plus some of the classics like *Design Patterns*, *Refactoring*, and so on. Devour them, and apply what they say to your code. There are many 'classics' in the field that are not hard to find. Also new classics like *97 Things Every Programmer Should Know*.
>
> Dive into some decent sized open-source projects where you can immerse yourself in Real World code-in-the-large, and can make public, demonstrable changes. Set up a GitHub account, and publish the things you work on there.
>
> Seek out code katas, and invest time in deliberate practice.
>
> Start a blog, and document your journey of learning.
>
> Are there any developer user groups near you that you could join? They often run evening events that are worth going to, to learn and to meet other coders.
>
> Don't worry about the academic course you're currently on; some of the best professional programmers I've had the pleasure to work alongside have not been Computer Science graduates, but people who qualified in other subjects (like Physics and Engineering disciplines).
>
> Good luck. And enjoy the journey!

That was an answer from the top of my head. I wonder what you'd add, or suggest differently? ■

Pete's new book – *Becoming a Better Programmer* – has just been released. Carefully inscribed on dead trees, and in arrangements of electrons, it's published by O'Reilly. Find out more from http://oreil.ly/1xVp8rw

Questions

1. What would *you* say to someone fresh entering the field? What advice would you give? What do you think have I missed?
2. Think back: what things helped you improve most as a programmer?
3. Is motivation to improve genuinely more important than technical skill? Can one be be learnt or 'caught' more easily than the other?
4. In 2015, how are you going to improve as a programmer?
5. Is there anyone you can give advice to, to mentor them, and help them improve? Is your excitement for programming and your motivation to seek to improve clearly seen by your peers?

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

Simplicity Through Immutability (continued)

a later version of C# the introduction of named arguments has meant that the invocation of a constructor was now also succinct and so mutability no longer had to be sacrificed for readability:

```
  var point = new Point( x: 1, y: 2 );
```

On the collection front the additional of interfaces such as `IReadOnlyList` and the new Immutable Collections library [4] means that the runtime madness of `ReadOnlyCollection` can be laid to rest.

Summary

Immutability is clearly not a panacea – one size never fits all – but hopefully following me through this thought exercise shows that it can vastly simplify the amount of time you spend thinking up scenarios to test, and therefore the amount of code you write to verify them. Given how tricky writing multi-threaded code already is, being able to reduce the amount you have to write will fill you with greater confidence that you haven't missed some subtle race condition. ■

Acknowledgements

Thanks to Jez Higgins and The Lazy Web (aka Twitter) for the brief discussion of Java's final keyword and its effects.

References

[1] http://www.bigbeeconsultants.co.uk/blog/single-assignment-pattern
[2] http://en.wikipedia.org/wiki/Turtles_all_the_way_down
[3] http://en.wikipedia.org/wiki/No_Silver_Bullet
[4] https://www.nuget.org/packages/Microsoft.Bcl.Immutable
[5] http://en.wikipedia.org/wiki/Builder_pattern
[6] http://c2.com/cgi/wiki?FactoryMethod

Delayed Copy Pattern

Vassili Kaplan presents some techniques for making efficient use of the STL containers in C++.

In time-critical financial applications it is very important to process market data messages as fast as possible. In this article we are going to take a look at different ways of adding big messages to a message queue implemented as an STL container. We will discuss advantages and drawbacks of each method and measure an average time it takes using each method. Finally, we will discuss why the new STL **`emplace_back`** method is so important for the time-critical applications.

Background and introduction

Consider an application that subscribes to the market data for a few financial instruments and then processes market data updates. These applications often implement the PUBLISH-SUBSCRIBE design pattern [1].

The application subscribes to the market data events by registering a callback with a market data publisher (e.g. Bloomberg, Reuters, etc.). On each market data event this callback is called by the publisher. In real life applications there can be thousands of subscriptions to different instruments and some liquid instruments can trigger up to a hundred of data events per second (new bids, asks, trades on different markets, etc.).

One such application is a 'Market maker' [2] application: based on some underlying instrument price, interest rates, time to expiry, and some other parameters, it calculates the price of a derivative instrument and quotes (i.e. buys and sells) this derivative instrument on an exchange. An example instrument might be Apple stock and its derivative an option, e.g. the right to buy Apple stock for $100 in 6 months. Obviously, the price of the option depends heavily on the current Apple stock price (among other things); this price can change a many times per second.

It is very important to react to the changes in market as quickly as possible, recalculating the option price when the underlying price changes (otherwise there may be an 'arbitrageur' who can buy the derivative too cheaply from the Market maker one millisecond before the price is updated and then sell it back to him once the Market maker finally updates the price – note that Market makers are obliged to buy as well as to sell – they usually make money from the spread between a bid and an ask price).

Very often the Black-Scholes equation [3] is used to calculate option prices. Since recalculation of the option price might take some time, usually market data events are only collected on the callback thread and added to a message queue to be processed from another thread. It is done in order to release the market data event thread so the next event can be added to the queue (the market data publishers also expect you to release the callback thread as soon as possible).

As a message queue an STL container is often used, e.g. a vector [4].

In this article we are going to look at different ways to efficiently add a market data object to the message queue, implemented as a vector. The use of locks and synchronization is beyond the scope of this article.

VASSILI KAPLAN

Vassili Kaplan has been a Software Developer for almost 15 years, working in different countries and different languages (including C++, C#, and Python). He currently resides in Switzerland and can be contacted at vassilik@gmail.com.

Listing 1

```
int currentId = 0;
vector<Normal> normal;

void normalTest(const Huge& huge)
{
  normal.push_back(Normal(++currentId, huge));
}
```

The classical way of adding an object to an STL container

Consider the following callback method where we receive a huge object with a lot of fields. Suppose that we do not own this object and want to add it to a container (see Listing 1).

Suppose that the **`Normal`** structure is a wrapper over the **`Huge`** object and is defined as in Listing 2.

When running the **`normalTest`** above the output will be the following:

```
<-- Normal Constructor, id: 1
--> Normal Destructor, id: 1
--> Normal Destructor, id: 1
```

Why was the constructor called only once, whereas the destructor was called twice?

The explanation is that the first time the **`Normal`** structure constructor was called was when passing the object to the vector's **`push_back()`** method, and second time the default copy constructor of the **`Normal`** structure was called when actually adding the **`Normal`** object to the vector's container. So all of the **`Huge`** object fields were copied in that copy constructor.

How can we avoid copying the fields twice?

Emplace_back method

Starting from the C++ 11 release a new **`emplace_back()`** [5] method is available. Instead of creating/copying an object twice, it is created just once:

```
void normalTest(const Huge& huge)
{
  normal.emplace_back(++currentId, huge);
}
```

Listing 2

```
struct Normal
{
  Normal(int id, const Huge& huge) : m_id(id),
                                     m_huge(huge)
  {
    cout << "<-- Normal Constructor,
            id: " << m_id << endl;
  }
  ~Normal()
  {
    cout << "--> Normal Destructor,
            id: " << m_id << endl;
  }
  private:
    int         m_id;
    Huge        m_huge;
};
```

When comparing the speed of adding a really big object with `emplace_back()` it is considerably faster than the classical `push_back` we discussed earlier (see time measurements below).

The only drawback is that `emplace_back` is available only starting from C++ 11 on. If you have an older compiler you have to use something else.

Using a vector of pointers

Another optimization is to use a vector of pointers instead of the vector of objects on the stack. Then even though there is still a double copying when using the `push_back()` method, one of the extra copies is copying pointers, which is much cheaper than copying our `Huge` objects. Instead of using pointers and then explicitly deleting them, we can use a `std::shared_ptr` instead:

```
vector<shared_ptr<Normal>> normalPtr;
void normalPtrTest(const Huge& huge)
{
  normalPtr.push_back(shared_ptr<Normal>
    (new Normal(++m_currentId, huge)));
}
```

When measuring the performance of creating objects on the heap, it is already much better than having a vector of the objects on the stack that we saw first. But it is still not as good as the `emplace_back()`.

How can we improve it for the cases when a C++ 11 compiler is not available so we cannot use the `emplace_back()`?

Adding a temporary field to the wrapper structure

Since the copy constructor is always called when the object is added to an STL container (from the `push_back()` method), we should do the actual copy of the big and expensive objects in the copy constructor, making the normal constructor very light-weight. But for this we need to temporarily keep a reference or a pointer to the object to copy. Let’s call this temporal reference `m_hugePtr`, which will point to the object to copy and will be later used in the copy constructor (see Listing 3).

Now, when running the same code with the `Tricky` objects:

```
vector<Tricky> tricky;
void trickyTest(const Huge& huge)
{
  tricky.push_back(Tricky(++currentId, huge));
}
```

Listing 3

```
struct Tricky
{
  Tricky(int id, const Huge& huge) : m_id(id),
     m_hugePtr(&huge)
  {
    cout << "<-- Tricky Constructor,
       id: " << m_id << endl;
  }
  Tricky(const Tricky& t) : m_id(t.m_id),
     m_huge(*t.m_hugePtr), m_hugePtr(&m_huge)
  {
    cout << "<-- Tricky COPY Constructor,
       id: " << m_id << endl;
  }
  ~Tricky()
  {
    cout << "--> Tricky Destructor,
       id: " << m_id << endl ;
  }
  private:
    Tricky& operator = (const Normal& other) {}
    void *operator new(size_t s) {}
    int         m_id;
    Huge        m_huge;
    const Huge* m_hugePtr;
};
```

As output we get:

```
<-- Tricky Constructor, id: 2
<-- Tricky COPY Constructor, id: 2
--> Tricky Destructor, id: 2
--> Tricky Destructor, id: 2
```

Note that it is easy to get into a real problem when this pattern is not used as shown above (e.g. copying a `Tricky` object when the underlying `Huge` object does not exist anymore, i.e. has been destructed). This is why we disabled creating the `Tricky` object on the heap by making the operator `new` private [6]. We also made the assignment operator private since it is not supposed to be used with this object.

Testing different ways of adding objects to the vector

For simplicity, suppose that the `Huge` object is implemented as in Listing 4 (in the real financial world it would have a lot of different double, integer, and string fields).

Table 1 contains the results of running four tests discussed earlier on a 4-core Windows 8.1 machine with an Intel i5 1.8 GHz and 4 GB in RAM. Each test was run 100 times and the data has the CPU times in milliseconds when adding 250 `Huge` objects to the message queue, implemented as a vector.

Table 1

Test name	Mean	Min	Max	Standard deviation
Normal Copy	58.59	46	79	8.95
Emplace C++ 11	45.94	31	63	8.71
Shared pointers	69.65	46	94	10.99
Delayed Copy	46.36	31	71	8.71

We can see that the new `emplace_back()` functionality should be used whenever a C++ 11 or a later compiler is available. When it is not available, for time critical applications the DELAYED COPY pattern could be used with care.

Conclusions

For time-critical financial applications every millisecond counts so it is important to develop very fast running software, otherwise there can be some faster guys who might punish you for being too slow.

Adding a big structure containing a big object to an STL container can be more expensive than it might appear: the objects belonging to the structure are copied twice – first time when we create and initialize the structure and a second time when it is added to an STL container.

C++ 11 has an excellent solution with the new `emplace_back` method which creates an object in-place, so no double copy is made.

But what can we do if we have an earlier compiler?

Listing 4

```
class Huge
{
public:
  Huge(const string& data = "",
     size_t numberElements = 0)
  {
    if (numberElements > 0)
    {
      m_data.reserve(numberElements);
      for (size_t i = 0; i < numberElements; i++)
      {
        m_data.push_back(data);
      }
    }
  }
  virtual ~Huge(){}
private:
  vector<string> m_data;
};
```

Const and Concurrency (part 2)

Ralph McArdell continues musing on comments to Herb Sutter's updated GotW #6b solution.

Previously [1] I wondered, from musings when reading Herb Sutter's updated Guru of the Week 6b [2] article, how one might – in C++11 – enforce a concurrent usage pattern in which an object can only be modified after creation by the creating thread until all modifications are done when the object becomes immutable and concurrently accessible. Concurrent access before an object becomes immutable is considered an error as are attempts to modify an object that is immutable.

Part 1 ended with a scheme in which erroneous updates in the mutable phase are detected by operations verifying they are called in the context of the creator thread by comparing the creator thread id – an instance member – with the caller's thread's id. I speculated that entering the immutable state could be indicated by setting the thread id member to the 'no executing thread' value of **std::thread::id** after which all non-mutating operations (spelt **const** in C++) may be called concurrently by any thread, and all calls to mutating operations would fail.

Before continuing I shall mention that I realise there are obvious, simpler, ways to arrange code to support this type of usage. That is not the point; the point is to see if such a usage pattern can be implemented in such a way as to report misuse and be convenient and efficient to boot while taking some of the new C++11 features out for a spin and seeing where we end up!

Let's continue by taking a detailed look at changing objects from mutable to immutable. This transition has two consequences:

1. The object cannot be modified at all other than to be destroyed. This could be termed *freezing* the object or similar.
2. All threads may access the state of the object, thus all modifications need to be made visible to any reader threads. We might say this is *publishing* the object.

On some hardware platforms we may be able to reliably achieve the first effect in the manner previously described – that is by just setting the updating thread id member to 'no executing thread', but this ought to be an atomic update. If C++11's **std::atomic** type template fully supported class-types, which it does not, we could just freeze an object something like so:

```
void the_type::freeze()
{
  validate_call_context();
  update_id.store(std::thread::id{},
     std::memory_order_relaxed);
  }
```

The reasoning is thus: the only thread that can access the object initially is the creator thread – all other threads will fail call context validation. After calling **freeze** even the creator thread would fail call context validation. Other threads will either see the original creator thread's id during call context validation or the updated 'no executing thread' value, neither of which will allow them to update the object.

Unfortunately no thread can even get read access to a frozen object as they will also in general be call context validated (the exception by the way would be for data that is initialised in the object's constructor and never modified thereafter). But after freezing non-mutating – or **const** – operations should be allowed. Providing an overloaded **const** qualified implementation of **validate_call_context** that allows access if the **update_id** is 'no executing thread' would achieve this.

This scheme will not fully publish an object's state to other threads. To do this there needs to be inter-thread memory access synchronisation. Specifically, the creator-thread has to release all memory writes it has made and all other threads will have to ensure they acquire these released writes before reading their values. Of course, each reading thread should do this as efficiently as possible – preferably only once.

Because in this case we have sets of pairwise synchronisation requirements between the creator thread and each reader thread acquire-release ordering can be used.

RALPH MCARDELL
Ralph McArdell has been programming for more than 30 years with around 20 spent as a freelance developer predominantly in C++. He does not ever want or expect to stop learning or improving his skills.

Delayed Copy Pattern (continued)

One solution is to have a container of pointers to the objects on the heap instead of a container of objects on the stack. But it turns out that on Windows this approach is suboptimal for really big objects.

Another solution is to delay the copy of the big object until the copy constructor is called by from the vector's **push_back()** method. But we need to keep a pointer to the big object. This pattern should be used with care – it is easy to get into a real problem when this pattern is not used as shown above. Also note that this pattern makes sense only for objects having a lot of underlying data, where an extra copy does take some considerable time.

Finally, the new **emplace_back()** feature available starting from C++ 11, permits not to use any potentially dangerous code and deliver a very efficient functionality that can be used in time-critical applications, so it should be used whenever possible. ■

References

[1] Publish-Subscribe Pattern
http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

[2] Market makers
http://en.wikipedia.org/wiki/Market_maker

[3] Black-Scholes Equation
http://en.wikipedia.org/wiki/Black%E2%80%93Scholes_equation

[4] Standard Template Library Programmer's Guide
http://www.sgi.com/tech/stl/

[5] STL Vector's **emplace_back**
http://en.cppreference.com/w/cpp/container/vector/emplace_back

[6] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Professional; 1996

The obvious choice here would be to use a **`std::atomic<bool>`** flag that the creator thread store-releases to in the publishing operation and each reader thread load-acquires from:

```
void the_type::publish()
{
  validate_call_context();
  published.store(true,
                  std:: memory_order_release);
}
```

In which the published instance member is of type **`std::atomic<bool>`** initialised to **`false`**. This allows the two **`validate_call_context`** overloads to simply check published to see if the object has been published (see Listing 1).

Note the difference in the checks for the **`const`** and non-**`const`** overloads.

Placing all the required scaffolding into a single class would allow 'client' classes to provide the required support more conveniently. As it stands only **`publish`** needs to be accessed outside the type's implementation so instances of such a support class could be included by composition as an instance member, with **`publish`** implemented as a forwarding operation to this member. Another possibility would be as a mix-in base class included by (private) inheritance.

There are still questions to resolve. Most prevalent is how the reader-threads know when they can access an object. With the scheme as discussed so far each reader thread would have to try a non-mutating operation repeatedly until it did not throw an exception – which underlines that it would definitely be a Good Idea™ to define a specific exception type in any real implementation.

Next, the scheme is intrusive – each operation has to remember to do something to validate the call context such as calling **`validate_call_context`**. Not only that but each operation has to atomically fetch data with potential memory synchronisation overheads on *each* call.

In theory at least the memory synchronisation overheads could be reduced for those processors where such overheads are high – namely those with a weakly ordered memory model – by using **`std::memory_order_consume`**, which is intended to rely on data dependency ordering, in place of **`std::memory_order_acquire`** [3]. As in this case all updates occur in the context of a single object, they would be dependent on the object's **`this`** pointer. Thus we can replace the published flag with a **`std::atomic<T*>`**, where **`T`** is the type of our object, initialised to **`nullptr`** and store-released to a value of the object's **`this`** pointer in **`publish`**. The use of **`std::memory_order_acquire`** would be replaced by **`std::memory_order_consume`** in both **`validate_call_context`** overloads. Additionally, all references to the object would also have to be initially loaded via a call to **`published.load(std:: memory_order_consume)`** – indicating that some refactoring of the code might be in order.

Note that I said 'in theory' in the preceding paragraph. This is because the current specification of C++11 makes it difficult for compiler writers to create an efficient, data-dependency ordering implement of **`std::memory_order_consume`** for weakly ordered CPUs and all implementations to date it appears take the lazy option of implementing **`std::memory_order_consume`** as **`std::memory_order_acquire`** [3][4].

The final problem that springs to mind is the question of knowing when it is safe to delete an object.

Then there is the question of what effect relaxing some of the constraints would have: allowing the transfer of update-status to another thread as mentioned towards the end of part 1 for example. So it seems the scheme is workable but there is definitely much room for improvement. I feel I may have to write up some further instalments at some point…■

Listing 1

```
void the_type::validate_call_context()
{
  if ( published.load(std:: memory_order_acquire)
      || std::this_thread::get_id()!=update_id
     )
  {
    throw std::runtime_error
    { "Illegal usage : Concurrent access or"
      "attempt at mutating operation on a"
      "published immutable object."
    };
  }
}

void the_type::validate_call_context() const
{
  if (!published.load(std:: memory_order_acquire)
      && std::this_thread::get_id()!=update_id
     )
  {
    throw std::runtime_error
    { "Illegal usage : Concurrent access to "
      " an unpublished object."
    };
  }
}
```

References

[1] Const and Concurrency (part 1), *CVu* Volume 26 Issue 5, November 2014

[2] http://herbsutter.com/2013/05/28/gotw-6b-solution-const-correctness-part-2/

[3] The Purpose of memory_order_consume in C++11, http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

[4] N4215: Towards Implementation and Use of memory_order_consume, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4215.pdf.

Standards Report

Mark Radford brings the latest news from C++ Standardisation.

Hello and welcome to my latest standards report.

In my last report I mentioned that the next ISO C++ meeting would be held in Urbana-Champaign, IL, USA, 3rd–8th November. Unfortunately my deadline for that report was a couple of weeks before the meeting, which meant I couldn't give it any coverage. Therefore I'll be covering it in this report. Also there are two new mailings since my last report i.e. the pre- [1] and post-Urbana [2] mailings (the timing was such that when I actually wrote my previous report, even the pre Urbana mailing wasn't yet published).

Getting back to the Urbana meeting, too much goes on at standards meetings for me to cover everything. Therefore, in this report I'll pick out the things that particularly caught my eye, and that I think will be of particular interest to readers. With this in mind, I'll cover the developments in the concurrency TS with regard to Executors, I'll catch up with the Concepts TS, and I'll give a brief update on the networking TS. First, however, something in the Evolution Working Group (EWG) came to my attention, because we discussed it in the BSI C++ Panel meeting at the end of last October, and because it has come up more than once over the years in the discussions on ACCU General. I'm referring to a proposal to make it possible to overload operator dot.

Operator dot

The proposal to make operator dot overloadable, by Bjarne Stroustrup and Gabriel Dos Reis, is entitled simply, *Operator Dot* (N4173). The idea is to be able to write 'smart reference' classes, similarly to how we currently have smart pointers. The ARM [3] explains that not allowing this in the first place was a conscious decision, so it is ironic that one of the authors of the ARM (Bjarne Stroustrup) is one of the authors of this proposal. This paper has met with a mixed reception. At the BSI C++ Panel meeting, there was concern about the possible effects, for example the possibility that some templates coud be made unsafe (a statement I'm not going to attempt to justify here, but which I hope to come back to in the future). Also, there was concern about the potential for causing general confusion. However, there was also some agreement that the idea is well motivated i.e. smart references are potentially useful, and it would be easier to write proxy classes. There was a consensus (in the BSI Panel meeting, and when this proposal was discussed by the Evolution group in Urbana) that the authors should be encouraged to continue this work.

Concepts

I first covered the topic of Concepts Lite in my May 2013 report i.e. the one that followed the Bristol ISO meeting where Andrew Sutton (the author of the proposal) gave his presentation on the Wednesday evening. The proposal was well received and generated a lot of interest, but things have been quiet on this topic of late. That's partly because recent work has been focused on C++14 (now an international standard), and partly because the Concepts work that has been carried out, has been in the background for a while. Note that there was never any ambition to get Concepts Lite ready in time for the C++14 standard. Instead, the goal was to produce a TS. I wondered if the committee might aim for C++17, but the goal is still a TS i.e. the original plan hasn't changed.

MARK RADFORD
Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

As the Urbana meeting began, the Core Working Group (CWG) began work on reviewing the current Concepts Lite draft, with the hope that a PDTS (Preliminary Draft Technical Specification) might be voted out by the end of the meeting. Unfortunately this was not achieved. By the end of the Tuesday session the author had a number of edits to make and work went on until the end of the Thursday session. However, at the end of that session a straw poll revealed that there was no clear consensus for the paper being ready to move to the PDTS stage. In the absence of a positive consensus, clearly the PDTS [4] will have to wait for more work to be done.

Executors and the Concurrency TS

I have already commented (in previous reports) on the dropping of Executors from the concurrency TS. Also, last time, I reported on the discussion of the two 'competing' Executers proposals that took place at the SG1 two day meeting last September. The two proposals are *Executors and schedulers, revision 3* (N3785) by Chris Mysen et al, and *Executors and Asynchronous Operations* (N4046) by Christopher Kohlhoff. Both these papers now have updates (following the SG1 meeting) in the pre Urbana mailing: the former is N4143, and the latter is N4242.

Here, there is some good news, at least in my opinion: Executors have still not been put back into the concurrency TS. Further, in Urbana, SG1 concluded that the concurrency TS should go to the PDTS stage without Executors. I think this is good news because of the lack of certainty over which Executors design C++ will eventually use. I should expand on that comment a little.

Previously the Executors design included in the concurrency TS (before the removal) was taken from Chris Mysen's proposal. Following the SG1 face to face meeting it was looking like that was going to be the case once again (with Chris Mysen updating his proposal to take ideas from Christopher Kohlhoff's into account). I have previously commented that Christopher Kohlhoff's proposal has been well received, and is preferred by some members of the BSI C++ Panel (me included). Therefore, with the concurrency TS not committed to either proposal, there is still time for it to make more progress and possibly become the Executors design adopted by C++.

In passing, note that the current version of *Working Draft, Technical Specification for C++ Extensions for Concurrency* (N4107) hasn't changed since the post Rapperswil mailing. However, the pre Urbana mailing contains a paper entitled *Improvements to the Concurrency Technical Specification* (N4123). I'm not going to comment on the latter, but I thought it would be worth drawing readers' attention to it.

Networking

The BSI C++ Panel meeting of December 2014 brought to my attention another paper by Christopher Kohlhoff: *Networking Library Proposal (Revision 3)* (N4332), which appears in the post Urbana mailing (revision 2 is in the pre Urbana mailing). This paper is currently passing through the Library Evolution group. This proposal now joins the several other works in progress that are heading towards becoming TSs. The interesting thing about this paper is that there was already a networking TS in development but, in Urbana, Library Evolution took the decision to replace it with Christopher Kohlhoff's proposal.

Finally

That's nearly all for this report. Just one more thing before I finish: I need to mention the BSI C++ Panel meeting dates for 2015. These are: 9th February, 20th April, 8th June, 3rd August, 5th October and

From the Coal Face

Ian Bruntlett shares his experiences: not salaried because of mental ill-health, but still working and learning.

In 2001 I was diagnosed with schizophrenia. I spent a fair amount of time between 2001 and 2004 in St George's (Psychiatric) Hospital. Whilst on East Loan (the hospital's rehab unit) I started a blog [1]. To cut a long story short, I found it hard to keep a job down because of both the effects of schizophrenia and the side-effects of my medication. In particular, I experienced Cognitive Impairment. So I set about fixing my brain using my brain and the help of NHS staff. I got to grips with Cognitive Impairment and went on to run a 45 minute talk about it at ACCU 2014 (I also wrote up my experiences [2]).

Working is a goldilocks problem for people with schizophrenia. Salaried work is nice but in my case the stress involved triggered ill-health – psychotic episodes. Sitting around in a hospital lounge drinking tea and chatting was nice but not productive. So I discovered that voluntary work at Contact [3] was the best compromise.

Volunteering

I am a Polymorphic Volunteer for Contact – I do I.T. support and pretty much anything else (running a Hearing Voices Group, answering phones, fielding queries, taking messages, making cups of tea). Because that does not take up too much of my time in Contact, I teach myself things whilst there. Thanks to Chrissie O'Dell, I have a laptop which I keep in Contact. I read Linux books and try things out on the laptop. I am currently working my way through *Learning PHP, MySQL, JavaScript and HTML5 (3e)* [4]. When I've done that I will submit a book review to Astrid Byro.

We had a wiki with a Software Toolkit page, full of links to useful pieces of free software. Because wikispaces.com has gone commercial, we have lost access to those pages. So I'm waiting for Contact's website to be handed over to Contact and I'll rebuild my Software Toolkit page.

I.T. is a bit like a Traveller (tabletop) RPG. You are faced with problem(s) and a less than perfect skill-set to fix the problems.

A recent problem happened with Contact's networked laser printer/photocopier. It stopped talking to the office's PCs. So I did some research. We had hundreds of pages of documentation on the printer and router as PDFs. Some of the printer's manuals had been printed. So I started reading and googling and taking notes for two days. Last time I was a netorking expert was for the Sinclair QL and things have changed since then. I discovered that the printer's I.P. address was now 0.0.0.0 which even I knew was a problem. There were two things that had to be done to fix the problem.

1. Configure the router to allocate a static I.P. address to a particular MAC/physical address.
2. Configure the printer to have the static I.P. address mentioned in step 1.

And, once things were working, I wrote up the notes. The printer has a drawer for manuals so I put it there.

I am also a volunteer for Ubuntu and lubuntu Quality Assurance. I have access to some old Dell computers and I install Ubuntu/lubuntu onto them and report successes and failures to particular e-mail mailing lists [5]. I also help out friends with Ubuntu/lubuntu PCs. Because I use Ubuntu/lubuntu Linux so much, every so often I buy stuff from the Ubuntu shop [6].This time round I bought a variety of things including a 14.10 t-shirt – Utopic Unicorn and an Ubuntu badge.

Platforms

I've changed software platforms a number of times – HP calculator, ZX Spectrum, Sinclair QL, PC with DOS, PC with Windows. And now I'm using Linux exclusively for personal use and for learning new things. I decided that, taking my experience into account, when I read the *Linux Pocket Guide* [7] from cover to cover, understanding and knowing all of it, I would be a Linux person. That stage has been passed.So I am a Linux person these days.

Direction

I've discovered that as long as I pace myself, I can stay well and stay focussed. I don't know what kind of voluntary stuff I will do in the future. I do have a few ideas, though :)

References

[1] http://schizopanic.blogspot.co.uk/
[2] https://sites.google.com/site/ianbruntlett/home/health
[3] http://contactmorpeth.org.uk/
[4] Nixon, Robin (2014) *Learning PHP, MySQL, Javascript, CSS & HTML5* (3rd Edition), O'Reilly Media, ISBN 978-1-49194-946-7
[5] http://community.ubuntu.com/
[6] http://shop.ubuntu.com/
[7] Barrett, Daniel J. (2012) *Linux Pocket Guide*, O'Reilley Media, ISBN 978-1449316693

IAN BRUNTLETT

On and off, Ian has been programming for some years. He is a volunteer system administrator for a mental health charity called Contact (www.contactmorpeth.org.uk).

Standards Report (continued)

16th November. All dates are Mondays. Anyone who wants to get involved can contact me in the first instance.

Acknowledgement

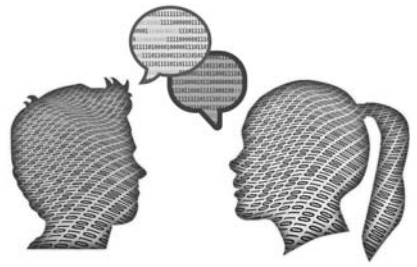
Thanks to Roger Orr for his email reports from the Urbana meeting, which he posted to the BSI C++ Panel reflector. They were very helpful to me in writing this report.

References

[1] Pre-Urbana mailing: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/#mailing2014-10
[2] Post-Urbana mailing: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/#mailing2014-11
[3] *The Annotated C++ Reference Manual* by Bjarne Stroustrup and Margaret A. Ellis.
[4] For information on the PDTS stage, and other stages that TSs go through during their development, see: https://isocpp.org/std/iso-iec-jtc1-procedures.

Code Critique Competition 91

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



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: we are investigating putting code critique articles online: if you would rather not have your critique visible please inform me. (We will remove email addresses!)

Last issue's code

I'm trying to instrument some code to find out how many iterator operations are done by some algorithms I use that operate on vectors. I've got some code that worked with C++03 and I'm trying to get it working with the new C++11 features. Mostly the C++11 code is just nicer but I can't get my wrapper vector to work with the new style for loop. I've stripped out the instrumentation for other methods in this example code so it only counts the increment operations and when I run the two simple examples below I get this output:

```
C: >example03.exe
Increments: 3

C: >example11.exe
Increments: 0
```

I expected the same output from both examples as all I've done is re-write the `for` loop using the new style `for` syntax.'

Can you find out why it doesn't work as expected and suggest some ways to improve (or replace) the mechanism being used?

The code is in Listing 1.

Listing 1

```
// -- wrapped_vector.hxx --
#include <vector>

template<typename T>
struct wrapped_vector : std::vector<T>
{
  typedef typename std::vector<T> vector;
  typedef typename vector::size_type
    size_type;
  // Sort the constructors
  #if __cplusplus >= 201103
    using vector::vector; // Nice :-)

  #else
    // legacy: need to spell them out ...
    wrapped_vector() {}
    explicit wrapped_vector(size_type n,
      const T& value = T())
    : vector(n, value) {}
    template <class InputIterator>
    wrapped_vector(InputIterator first,
      InputIterator last)
    : vector(first, last) {}
    // ...
  #endif // C++11
  // print the stats
  static void dump();
  // instrumented iterator
  struct iterator : vector::iterator
  {
    typedef typename vector::iterator base;
    iterator() {}
    iterator(base it) : base(it) {}
    iterator& operator ++();
    // (other instrumented methods removed)
    static int increments;
  };
  // const_iterator (unused so removed)
};
#include "wrapped_vector.inl"
// -- wrapped_vector.inl --
#include <iostream>
template <typename T>
void wrapped_vector<T>::dump()
{
  std::cout
    << "Increments: "
    << iterator::increments
    << std::endl;
}
template <typename T>
typename wrapped_vector<T>::iterator&
wrapped_vector<T>::iterator::operator ++()
{
  base::operator ++();
  ++increments;
  return *this;
}
template <typename T>
int wrapped_vector<T>::iterator::increments;
// -- example03.cpp -
// (also works with C++11)
#include <cassert>
#include "wrapped_vector.hxx"
int main()
{
  wrapped_vector<int> iVec;
  iVec.push_back(1);
  iVec.push_back(0);
  iVec.push_back(2);
  int total(0);
  for (wrapped_vector<int>::iterator
    iter(iVec.begin()), past(iVec.end());
    iter != past; ++iter)
  {
    total += *iter;
  }
  assert(total == 3);
```

Listing 1 (cont'd)

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

Listing 1 (cont'd)

```
  wrapped_vector<int>::dump();
}
// -- example11.cpp -- (C++11 example)
#include <cassert>
#include "wrapped_vector.hxx"
int main()
{
  wrapped_vector<int> iVec;
  iVec.push_back(1);
  iVec.push_back(0);
  iVec.push_back(2);
  int total(0);
  for (auto & p : iVec)
  {
     total += p;
  }
  assert(total == 3);
  wrapped_vector<int>::dump();
}
```

Critiques

Alex Paterson <alex@tolon.co.uk>

The problem area is around analysis of algorithm performance, with this case specifically looking at iterator usage. Analysis of code in this manner often proves tricky, but can provide vital insights into how efficient our code actually is rather than just how efficient we think it is.

The C++03 and C++11 code specimens have one major difference that is the root of the problem; the type of iterator used to iterate over the collection is different – one of them using the instrumented iterator and the other is not. This problem is introduced by the different method of iterating over the collection in the C++11 code. The C++03 method of declaring and incrementing an iterator is replaced with the range-based **for** loop.

The range-based **for** statement (RBF) is type of statement that can be referred to as 'syntactic sugar'; it is a construct that performs some task that is already possible using other language features, but does so with simpler and/or a more succinct syntax. In other words, the RBF does not add any new functionality to C++, but it reduces the amount or code we have to write (and thus read and maintain). The C++ standard specifies how RBF usage translates into the more long-winded syntax, encouraging compiler writers to add support for the new syntax by implementing it in terms of existing functionality. If I ignore usage for array types, the standard states that the compiler will use argument-dependent lookup to try and find suitable **begin()** and **end()** functions for the given type; if all goes well, the type of iterator used in the RBF loop will be the type returned from **begin()**.

```
class X { /* … */ };
vector<X> container_of_x;

// C++11 Range Based For
for (auto& x : container_of_x)
{ /* do something with x */ }

//Long Winded Equivalent
for (vector<X>::iterator it =
  begin(container_of_x);
  it != end(container_of_x); ++it)
{
  X& x = *it;
  /* do something with x */
}
```

The C++03 code compiles and produces the expected output when compiled against either the C++03 or C++11 standard, which reinforces the suspicion that the two problem code specimens are not equivalent. Making a small change in the C++03 code specimen to use **auto** as the iterator type reveals the problem behaviour.

```
//C++03 Code
for (wrapped_vector<int>::iterator
  iter(iVec.begin()), past(iVec.end());
  iter != past; ++iter)
{
  total += *iter;
}
// Output: Increments: 3

//C++03 Code Modified
for (auto iter(iVec.begin()),
    past(iVec.end());
    iter != past; ++iter)
  {
    total += *iter;
  }
// Output: Increments: 0
```

The conclusion from the output is that the loop using **auto** is not using the instrumented iterator. In case this is not obvious, we can stream out the type of iterator using **typeid(iter).name()**, which shows different type names. [Editor: it *may* show different type names – but the usefulness of the output from **name()** varies widely.]

In both cases **iVec.begin()** is used to initialise the iterator (which returns **std::vector<T>::iterator**), but in the first loop this is wrapped in the instrumented iterator (of type **wrapped_vector<T>::iterator**). In the second loop, the wrapper is not present, so the instrumentation code never gets run. Additionally, a better name for the iterator type in **wrapper_vector** could have perhaps made it easier to spot this issue; as the code stands, there are two definitions of iterator in **wrapped_vector<T>**, one at the **wrapped_vector<T>** scope and another one at the **std::vector<T>** scope.

This explains the behaviour we are seeing, but the question now is how can we change things so that we can still use the pretty new syntax and be able to determine how many iterator increments occur?

Inheriting from **std::vector** in this manner is not recommended (see stackoverflow – http://stackoverflow.com/questions/ 6806173/subclass-inherit-standard-containers, http://stackoverflow.com/ questions/ 4353203/thou-shalt-not-inherit-from-stdvector). Using private inheritance in the original code improves the situation [slightly] and reveals that three functions are being used from the base class: **begin()**, **end()** and **push_back()**. **using push_back;** can be used to pull the private base class declaration into public visibility, however we need to write our own implementations of **begin()** and **end()** in order to provide an instrumented iterator using our wrapper class, as follows (remembering that at the **wrapped_vector<T>** scope, the type 'iterator' is our instrumented iterator);

```
template<typename T>
struct wrapped_vector : private std::vector<T>
{
//…
  using std::vector<T>::push_back;
  iterator begin()
  { return iterator(std::vector<T>::begin()); }

  iterator end()
  { return iterator(std::vector<T>::end()); }
  //…
};
```

This gives us the expected output for our C++11 loop, however it is still inheriting from **std::vector** so is far from ideal. Extending **std::vector** in this manner is not recommend, even if it does yield the right results. The standard library collection classes are not designed to be extended. **std::vector<T>** does not have a virtual destructor and we therefore run the risk of the instrumented vector's destructor not getting run.

This would lead to a resource leak if our vector class had any non-static member variables. E.g.

```
std:vector<int>* my_naked_vector_ptr
  = new wrapped_vector<int>;
delete my_naked_vector_ptr; // <-- risk of
                            // resource leak!
```

So if inheritance is not an option, then how can we analyse how efficient our algorithms are? Here are two options that I have used: alter the behaviour of the container or alter the behaviour of the contained objects. To alter the behaviour of the container, we can provide a custom allocator to track allocations of the contained objects. To alter the behaviour of the contained objects, we can substitute a test object that can track events like construction, destruction, assignment and comparison.

I've found that using a test object is the preferred technique as it provides far more opportunities to examine different behaviours, but there are cases where it is not desirable/possible to change the type of the contained objects, so altering the container is still a useful technique to know.

Altering the container

The template parameters of the standard containers provide the opportunity to specify not only the type of object that is contained, but also how they are allocated.

We implement an instrumented allocator by placing our counter increment code before we delegate the **allocate** and **construct** calls to their default implementations.

```
template<typename T> struct wrapped_allocator
  : std::allocator<T>
{
  typedef T* pointer;
  pointer allocate(
    typename std::allocator<T>::size_type n,
    std::allocator<void>::const_pointer hint
       = 0)
  {
    allocations++;
    return std::allocator<T>::allocate(n,
      hint);
  }
  template< class U, class... Args >
  void construct( U* p, Args&&... args )
  {
    constructions++;
    return std::allocator<T>::construct(p,
      std::forward<Args>(args)...);
  }
  static void reset_counters()
  { allocations = constructions = 0; }
  static void dump_counters()
  {
    std::cout
      << "allocations: " << allocations
      << ", constructions: " << constructions
      << std::endl;
  }
  static int allocations;
  static int constructions;
};
template<typename T> int
  wrapped_allocator<T>::allocations;
template<typename T> int
  wrapped_allocator<T>::constructions;
int main()
{
  std::vector<int, wrapped_allocator<int>>
    v{5,4,3,2,1};
  wrapped_allocator<int>::reset_counters();
  std::sort(begin(v), end(v));
  wrapped_allocator<int>::dump_counters();
}
```

Output: allocations: 0, constructions: 0

Oh, so maybe not so useful in this case, but what about comparisons? For this we need to move to our own object where we can define our own comparison operator.

Using a test object

A test object provides a way to examine the behaviour of our algorithm by instrumenting various functions. I usually add instrumentation to the constructors, assignment operator and inequality operators, but of course what gets instrumented depends on what we are trying to find out.

Combining these typical traces together, we can get a pretty good picture of what the algorithm is doing in terms of object creation and comparison.

```
class CTestObject
{
public:
  CTestObject(int n) :
    m_payload(n)
  { constructions++; }
  CTestObject(const CTestObject& src) :
    m_payload(src.m_payload)
  { constructions++; }
  CTestObject& operator=(
    const CTestObject& rhs)
  {
    assignments++;
    m_payload = rhs.m_payload;
    return *this;
  }
friend bool operator<(const CTestObject& lhs,
  const CTestObject& rhs)
{
  CTestObject::comparisons++;
  return lhs.m_payload < rhs.m_payload;
}
friend bool operator==(const CTestObject& lhs,
  const CTestObject& rhs)
{
  CTestObject::comparisons++;
  return lhs.m_payload == rhs.m_payload;
}
static void reset_counters()
{
  assignments = constructions = comparisons=0;
}
static void dump_counters()
{
  std::cout << "CTestObject, assignments: "
    << assignments << ", constructions: "
    << constructions << ", comparisons: "
    << comparisons << std::endl;
}
private:
  int m_payload;
  static int assignments;
  static int constructions;
  static int comparisons;
};
int CTestObject::assignments;
int CTestObject::constructions;
int CTestObject::comparisons;
int main()
{
  std::vector<CTestObject> v{5,4,3,2,1};
  CTestObject::reset_counters();
  std::sort(begin(v), end(v));
  CTestObject::dump_counters();
}
```

Output: CTestObject, assignments: 16, constructions: 8, comparisons: 9

Of course, the number of assignments, constructions or comparisons is not necessarily the be-all and end-all of whether you have a good algorithm.

It may be more important in certain cases to ensure that your algorithm is cache-friendly, which may mean extra allocations or a significant increase in the number of comparisons, but it may ultimately be faster.

James Holland <james.holland@babcockinternational.com>

I may have the wrong end of the stick, but I do not think there is much wrong with the author's code. It does exactly what is required. Specifically, it counts and displays the number of **iterator** increment operations. The confusing thing, perhaps, is that the C++11 style range-based **for** loop shown in the code listing does not use **iterator**s. It works directly on the type the **vector** contains, namely, **int**s. This is why a count of zero **iterator** increment operations are displayed when using the C++11 loop.

If the author wants to monitor the operations performed on **int**s, one way of proceeding is to replace the plain **int**s contained in the vector with a class with the same behaviour as **int**s but with additional instrumentation. This will allow the number of operations performed on the instrumented class to be counted. One version of such a class, named **My_int** is listed below.

```
class My_int
{
  int i;
  static int operator_plus_plus;
  static int operator_plus_equals;
  public:
  My_int(int ii):i(ii){}
  My_int & operator+=(My_int my_int)
  {
    i += my_int.i;
    ++operator_plus_equals;
    return *this;
  }
  My_int & operator++()
  {
    ++i;
    ++operator_plus_plus;
    return *this;
  }
  static void dump_operator_plus_plus()
  {
    std::cout << "operator++(): "
      << operator_plus_plus << std::endl;
  }
  static void dump_operator_plus_equals()
  {
    std::cout << "operator+=(): "
      << operator_plus_equals << std::endl;
  }
};
int My_int::operator_plus_plus = 0;
int My_int::operator_plus_equals = 0;
```

My_class includes definitions of **operator+=()** and **operator++()** that when called increment corresponding static members **operator_plus_equals** and **operator_plus_plus**. Other operations could be defined if required.

All the author has to do is replace the **vector** of **int**s with a vector of **My_int**s, execute the algorithms that use the **vector** and then call **My_int** member functions **dump_operator_plus_plus()** and **dump_operator_plus_equals()** to display the total count of operations.

Incidentally, it is interesting to consider the construction of the C++03 style **for** loop. The author has declared a variable, named **past**, and assigned it the value of **iVec.end()**. The loop then used past, as opposed to **iVec.end()**, to determine when to terminate. I was wondering what the advantage of this method affords. I can only think it is marginally faster. It would be interesting to hear what others have to say about this construction.

Commentary

This critique shows how hard it can be to derive from a class that is not designed for derivation. In particular, if new methods are added to the base class the assumptions used to originally produce the derived class may fail completely. C++11 added methods to the many of the standard classes, such as vector and its iterators, as well as mechanisms to support the 'range-base for' syntax.

The fundamental problem is that none of the methods of vector and the iterator are virtual, and so it is all too easy to call one of the original methods being overloaded. In the presenting case that's what has happened to the iterator type: the fragment for **(auto & p : iVec)** deduces the type of the underlying iterator as a 'vanilla' vector iterator rather than the wrapped iterator type.

In this case the fix is simple: add member functions **begin()** and **end()** to the **wrapped_vector**:

```
iterator begin() { return vector::begin(); }
iterator end() { return vector::end(); }
```

to ensure – in this case, anyway – that the range-based **for** loop uses the desired iterator.

Alex's suggestion of using private inheritance is a good solution to the difficulty as there is then no implicit conversion (outside the wrapped type) to the underlying class, and so missing methods in the wrapped class will normally cause compilation errors rather than silently using methods from the base – as happened here. (This also prevents the dangerous conversion to a base class without a virtual destructor.)

The presenting problem, "I'm trying to instrument some code to find out how many iterator operations are done by some algorithms I use that operate on vectors" begs the question of *why* – I strongly suspect there's a performance or scalability problem somewhere.

It is often better to instrument the whole program, or a small test program, using standard tools (you may recall articles about the valgrind set of tools in recent issues of *Overload*) as otherwise the danger is, as Alex points out, that the measure you are collecting is not the right one.

The winner of CC90

Alex and James both suggest instrumenting the payload rather than the iterator; although this mechanism may not permit exactly the same measurement it does provide a clean way to instrument other operations.

Alex also gave clear explanation of why the code was broken so I am happy to award him the prize for this critique.

CC89 reprise...

Silas Brown wrote in, following his entry in *CVu* 26.5, to say:

> In Code Critique 89 I incorrectly stated that all cards with odd value will also have colour set to red, whereas all cards with even value will also have colour set to black. This is not the case: as pointed out by Paul Floyd in his critique on the same page, the single loop works because 4 and 13 are relatively prime. The value is taken modulus 13, so the iterations are out of phase and eventually all combinations will be assigned. We can confirm that simply enough by typing the following into a Python prompt:
>
> ```
> sorted((i%4,i%13) for i in range(52)) == \
> [(x,y) for x in range(4) for y in range(13)]
> ```
>
> and Python will reply:
>
> ```
> True
> ```
>
> Silly me didn't see it: that's what sometimes happens when you think about code without a compiler on hand to actually try it. However, the rest of what I said still stands, including the part about doing it with multiple loops being much clearer. If I can misunderstand the code in this way after more than 20 years of programming, then your colleagues might misunderstand it as well. The only disadvantage of having multiple loops that I can see is that more CPU registers would be required, but that's only an issue nowadays if you're programming a microcontroller. And if that's what you're doing, taking modulus 13 on every step might

Scott Meyers: An Interview

Emyr Williams continues the series of interviews with people from the world of programming.

Unless you're very new to C++ programming, or have been living in a cave, then you will have heard of Scott Meyers. He is the author of the best selling book *Effective C++*, and his latest book *Effective Modern C++* has just hit the bookshelves. Scott has been a C++ consultant and trainer for at least twenty-five years. And he trained me on the new stuff in C++ in September, and so I got to carry out this interview face to face.

How did you get in to computer programming? Was it a sudden interest? Or was it a slow process?

> I started programming in grade school, and they had a time sharing system, and there was a math teacher who recruited me and a couple of other people. And she thought that we might enjoy programming, which started out by playing games. This was on a teletype system with a piece of paper in it, and it was a 110 baud teletype thing. You could play still some remarkably good games on that system.
>
> After we played some games, we thought it would be fun to learn how to program. Which was probably the math teacher's idea, to learn how to program. And we started to do that after school, as did my friend. And one thing led to another, and before you know it you're spending way too much time in a former cloakroom closet with a teletype until they make you to go home at six o'clock at night.

It was probably a small room as well? Probably about the size of a large wardrobe?

> It was a former 'cloak' room. So it was like a walk in closet that had a teletype, which made a lot of noise. It was a mechanical device, so they stapled eggshells as noise dampeners all the way around. So it was this narrow room, with two young men, and a machine that made a lot of noise. It must have smelled horrendous!

What was the first program you ever wrote? And in what language was it written in? Also is it possible to provide a code sample of that language?

> I don't remember what the first program I ever wrote was. The earliest program off hand I can remember writing was a couple of years after that, but it was a horse racing program. Which had all the sophistication you would expect from somebody who was fourteen years old.
>
> It basically involved as I recall, several horses, that had to go a certain distance and each iteration you chose a random number that would determine how much further they went. That's about as sophisticated as it got. But it was kind of cool, because it's on a teletype system. You can't show things in real time, so what I'd do was have it type out x's – like a histogram. But it's a 110 baud teletype, so you'd see the all the x's, and then you had to wait before

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

Code Critique Competition 91 (continued)

> turn out to cost more CPU resources than the nested loops would anyway.

Code Critique 91

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

> I'm trying to migrate my skill set from C to C++ so thought I'd get started with a simple program to fill in a set of strings and print them. But I'm getting a compilation problem on the call to **std::copy** that makes no sense to me, although I thought I'd copied it from some working code on the Internet. Can you help me get it to compile?

Can you help this programmer to get past this presenting problem and help them to identify any other issues with their code? The code is in Listing 2.

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

Listing 2

```
#include <iostream>
#include <iterator>
#include <set>
// compare *contents* not raw pointers
bool string_less(char const *, char const *);
template <class T, class U>
void test(std::set<T, U> s, T p)
{
  if (s.insert(p).second)
    std::cout << "Added " << p << std::endl;
  else
    std::cout << p << " already present";
}
```

Listing 2 (cont'd)

```
int main()
{
  std::set<char const *,
    decltype(&string_less)> s(string_less);
  test(s, "A");
  test(s, "B");
  test(s, "AB");
  std::copy(s.begin, s.end,
    std::ostream_iterator<
      char const *>(std::cout , " "));
  return 0;
}
bool string_less(const char *s, const char *t)
{
  while (*s == *t)
  {
    s++;
    t++;
  }
  return(*s < *t);
}
```

it printed out the next set of results. So there was a certain degree of suspense in trying to figure out which horse was going to win. Nothing fancy, I'm sorry.

What about a modern language? Such as C or C++?

In C, because I learned from Kernighan and Richie, I wrote Hello World. In C++ I also wrote Hello World. Because there's a lot of merit in making sure you have all the pieces together to write computer programs.

What would you say is the best piece of advice you've ever been given as a programmer?

I don't know what the best piece of advice I've been given is, but I'm going to turn the question around a little bit. In my youth, there came a point where I thought I knew everything. And I was working as a software developer at the time. And I made a comment about blah blah blah being impossible because of blah! And another guy looked at me and went, "you know..." and laid it out for me, that it was not impossible, and how it was not impossible and it turned out I didn't know everything. And that was an important learning experience for me, to recognise that there's a lot of stuff I don't understand.

And what I've since learned over the years is that there's a reason for everything. And if you look at something and it makes no sense or it seems crazy or it seems stupid, there's a reason for it. And whoever came up with whatever you're looking at, or whatever group came up with what you're looking at, they had a reason for doing it. And it was probably a reasonable reason.

I've found that to be extremely helpful over the years, just to say "Why were these things done?" Someone had the goal of achieving something good. So if you're looking at something that's overly complicated or doesn't look correct, then it's important to find out why it's done the way it is.

You are very well known for the *Effective* series, how did that come about? Did anything make you decide "there needs to be one of these"?

I was working as a trainer at the time, and I was training C programmers using five day hands on courses to learn C++. This was in the early 1990s. So C++ was a simpler language back then, there were no exceptions, no templates, so it was as simpler thing. But when you teach C programmers, C++ programmers forget how much how much has to be learned. For the poor C programmers, they don't know what a class is, they don't know what a virtual function is, they don't know what inheritance is, they don't know what overloading is, they don't know what constructors are, what destructors are, they don't know what references are and how they're different from pointers. The list goes on and on.

And so I would teach these poor C programmers over the course of five days how to use C++, and by Friday afternoon, their heads were swimming. They were thinking "I'm never going to be able remember all this." And five days is not a lot of time. So on Friday afternoon, I'd write on the whiteboard, and tell them "It's not that hard, let's list the stuff you really need to remember." So for example, if you have a base class, you need a virtual destructor, or if you have a class with pointers then you need a copy constructor and an assignment operator, or you should never redefine a default parameter value, these sorts of things.

And I found that this made them feel better. So that's how the list of items began. I taught somewhere, and I don't recall where, and the group said "you should write a book." And I said I'm not going to write a book. Then another place said, "you should write a book," and at that time I was working on my PhD, and I thought "well, I could work on my PhD, which is hard, or I could write a book, which can't be as hard as working on a PhD." So I decided to write the book instead. It was a spontaneous decision, and the book was well received, and everything was derived from that.

If you were to go back in time and meet yourself at 14 years old, when you were in the teletype room, what would you tell yourself?

You don't know everything. The world is more complicated than you think it is. If I learned that lesson a lot sooner, that would have been useful.

If you started your career as a programmer now, what would you focus on? Would it still be C++? And which field would you be interested in working in?

I think that for me personally, so much is a matter of happenstance. So the first language I learned was BASIC. I learned BASIC because the teacher said "Why don't you learn BASIC?" And for that matter, the reason I learned C++, was because I in graduate school, I was required to TA [be a teaching assistant for] a course on software engineering, and the professor of that course decided "we're going to use C++, so you must learn C++". I didn't choose either one of those things, they just sort of happened to me.

So I'm going to turn your question around a little bit, and say if it was my goal to introduce new people to programming, then what would I do? And what I would do is something mobile. I think that mobile devices are really interesting to people, and you can do all sorts of cool stuff. And I would do something using some technology which would allow people to get a lot done really quickly, with really fast turnaround. Because I think that's what really hooks people: I can get this stuff to work. So the Hello World for me would probably be a way to very quickly write a little application that would send a message using some technology from one mobile phone to another using giant libraries that the programmers would have no idea about how they worked.

For a lot of new developers, it's important that they see something happening, so when I started on Visual Basic for example, you'd draw a button, write some code behind it, click on it, and something would happen, so the fact they can see what they've written actually does something is quite important.

I think the immediate feedback combined with high likelihood for success is what draws people in. At some point you have to get in to the nitty gritty stuff. But I wouldn't start with C++. For one thing it's not fun, it has no graphics, it has no networking. A lot of stuff is missing.

What would you describe as the biggest "ah ha" moment or surprise you've come across when you're chasing down a bug?

The one that comes to mind is when I found out that the cause of the problem I was running in to was an instruction in my program that was comparing two floating point numbers for equality. And it hadn't occurred to me that because two things are mathematically equivalent, that does not mean you'll get the same result on the computer. So I spent a lot of time tracking that down, and I've never forgotten that.

Have you got any tips for any new programmers that are chasing down bugs at the moment.

I would say that like most things, the way you get better at it is by doing more of it. But you have to do more of it, but learn at the same time. So seek out other people who are better at it than you are, and try to continue to get new sources of information. If you're tracking down a bug, and it's difficult, get someone to help you. Ask other people what their ideas are, and that will help you develop an intuition of what to look for.

Do you have any regrets as a programmer? For example wishing you'd followed a certain technology more closely or something like that?

You know, I really don't. I'd like to know more about some areas, but at the same time, I'm happy where I am. As a specific example, in the mid 1990s when Java became very popular, a lot of people in the C++ community went "this is a cool thing!" and they moved across to Java, and I decided not to move to Java. Primarily because I thought "I've already learned one programming language, and the last thing I need to do is to prove that I can learn another programming language." Which allowed me to completely and utterly miss the boat on Java. But as a result, I've stayed constant to

C++, and I've learned a lot from it, so I don't really have regrets as far as that goes.

Do you code in other languages? Such as Python or other scripting languages?

C++ is my language. It's the only thing I do. If I were an actual programmer, I'd have to do a lot more. There are some other languages I can do a little bit, but C++ is really what I do.

From what I've read of your website, a lot of your work today is training and consulting, so I was wondering how do you make your code samples relevant to what the programmer may face in their everyday work?

What I try to do is to have enough contact with real programmers on a regular enough basis that I get feedback from them as to whether what I'm advocating or what I'm saying makes sense. And because I'm not a practicing programmer right now, and because I work by myself and I'm not surrounded by a bunch of other people in a company, I try really hard to stay in touch with real programmers developing real code.

My experience is if you give people advice that's not practical, they will tell you right away. If you give them advice that's too simple, they'll tell you right away. For example we just spent the last two days talking about this sort of stuff, and people ask questions, people make comments, they have funny looks on their faces. And you get feedback as to whether what you're telling them seems relevant to their job. And that's my primary goal.

I feel I should apologise, that I typed in some of your code during the course, and it didn't do what you said it would do.

I love that! It reminds me I was in China at one time, and I was teaching a course and it happened to be on the internals of how certain things work. And I was talking about how virtual function tables are implemented under multiple inheritance.

So I do my presentation and I go "that's how virtual functions are implemented under multiple inheritance." And this one guy looks up from his computer and goes "no it's not!"

So we looked at his compiler, and did some disassembly, and it turned out that the information I had was completely accurate for one compiler, and he was using a different compiler, which did things in a different way. So this is an example of how people keep me honest. So he learned that there were other ways to do it than that one compiler he was using, and I learned that there are other ways to do it other than the compiler I was using.

So I love it when people point out stuff that I wasn't aware of.

Are you currently a mentor? And if so, what do you do with your mentees?

No

Did you ever have a mentor yourself?

Certainly not by that name, I certainly learned from other people, but I never had a formal relationship with someone who was supposed to help me improve what I do. And in retrospect I learned mostly from peers with more or less the amount of experience as I did, but I wasn't a practicing programmer for all that long. I was a programmer for three years.

As a trainer, I suppose you read quite a lot? What would you say is the best book you've read?

Nothing comes to mind. What I will say is that I don't read many books. I read a giant amount of blog posts, I read Stack Overflow when I get a chance, or when I'm researching, I read a lot of e-mail that I exchange with other people, I read a lot of papers, I read a lot of online shorter stuff. I don't read a lot of books. For example, we talked earlier (in the course) about Anthony Williams' book, *C++ Concurrency in Action*, I didn't read that whole book. But for example, the chapter he has in there on the memory model is just killer! So I was very pleased with that. I don't tend to sit down and read entire books, because in the C++ area, it's uncommon to find a book that is filled with stuff I don't already know.

So how do you go about learning new techniques then? For example, when the move paradigm came in C++ 11/14, how did you go about learning that? I suppose I'm really asking what's your learning style? Do you read then dive in or do you dive in or is it a mix?

It depends on what I'm trying to learn. Let's take something like C++ 11/14, something that's technically defined by the standard. What I normally do in those conditions is start with blog entries, where people have written overview blog entries or something like that, so I can try to build a mental model. And if I have a questions, I get the questions answered in various ways, sometimes I turn to Stack Overflow, sometimes I turn to people that wrote standardisation proposals. I read a lot of standardisation proposals. I play around with compilers too, but I have to say that when it comes to standardisation-related stuff, compilers can be helpful, but if you want to know what the standard says, then you need to know what the standard says. So playing around with code is less useful, especially with technology that is newer, because compilers may not have implemented it yet.

On the other hand, for example we had that question in the course about what happens if you have an infinite loop in a **`constexpr`** function. To me, that's a question compilers need to answer.

So we have an update to *Effective C++*, will you be doing an update to *More Effective C++* and *Effective STL*? Or are all the updates incorporated in the new book?

To be clear, I didn't update *Effective C++*. The new book is called *Effective Modern C++*. It has completely new information, so it's not an update of *Effective C++*.

Sorry

No that's fine; actually I'm eager to get the word out about that. It's a completely new book. It would have been a lot less work had it been a new edition.

As to the question of whether I'll be updating any of my other books, that remains to be seen. I will say that it's unlikely that *More Effective C++* will get updated. That book is now almost 20 years old. A lot of the information is still useful, but I'm not sure at this point it's really worth updating.

Where do you think the next big shift in programming is going to come in? I've noticed you've done some stuff with D. Do you see that as the next big thing? And for the uninitiated, what is D?

I'm not doing anything with D, actually. There are people in the D community that would like me to do something with D, people who I respect. But that's not my plan. I was asked to give a keynote talk at the most recent D conference, that's what I did. I went and gave a talk to the D people, and I basically encouraged them to avoid creating a language as complicated as C++, that was basically my message for them.

As for the next big thing, I'm really bad at predicting that sort of thing, so I'm not going to even try.

Finally, what advice would you offer to kids or adults that are looking to start a career as a programmer?

The people who I know who are good at programming, who are happy with their lives – that kind of stuff – they do it because they love it. And so I would say that if you are looking at it as though to say "this would make a nice career" but your heart isn't in it, maybe you should be looking elsewhere. On the other hand, if you play around with it, if it seems fun, if you like the idea of controlling machines or if you want to accomplish something, then you should just do it. Because it can be fantastically rewarding. It's a "Follow your heart" kind of thing. Because if you don't like it, it's hard!

Thank you very much for your time.

You're very welcome.

Bookcase

The latest roundup of book reviews.

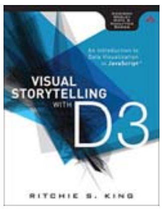
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous '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)

Visual Storytelling with D3



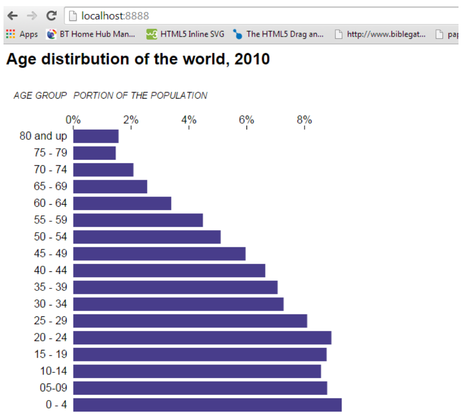
By Ritchie S. King, published by Addison-Wesley 2015 , 264 pages, £24.99

Reviewed by Frances Buontempo

D3 is a JavaScript visualisation library for HTML and SVG, which you can either download or link directly to in a script element on your html. There are several online tutorials, but reading a real dead-tree book from time to time is enjoyable.

This book developed a narrative, starting by drawing a barchart in html, then moving on to using JavaScript, and after introducing the use of svg on a web-page, showed how to use D3 selections and data joins to do the same thing. It ends with a dynamic example that 'plays' through bar charts for various years of data.

It is aimed at a beginner, explaining that you cannot use Word as an editor and how to inspect elements and use the console in Chrome. Some of the earlier chapters were devoted to the 'storytelling' part of the book, considering when bar chart and lines charts are better, and reminding us that pie charts with more than a few sections can be hard to garner information from. Simpler is usually better. I suspect there are books on these specifics with far more detail about possible graphics for various types of data and things to avoid. I personally didn't gain much from this short section, but if you've never heard of a bar-chart before or can't remember how they work, it could be useful. It also reminded me how to calculate percentages, so again this is aim at a newbie. Other parts were devoted to explaining what a variable is, a function is, not to be afraid of anonymous functions as well as explaining the difference between a domain and a range. If you have no background in such things, this might be helpful.



One thing that is not immediately obvious from the online docs (after reading them for a total of about 5 minutes) is how to use a data file – lots of the simple examples hard-code the data in an array. The books showed how to run a simple web service using python, which allows the script to read the file, when placed in the same directory as a script. That was useful, since you will want to move beyond hard coding small data sets if you wish to move beyond the basics.

By the end of the book I did actually have code showing a bar chart in a web browser, which I failed to get working first time when I just went to the D3 manual pages, so I gained something from reading this book. It didn't move beyond bar charts, though it mentioned circles, eclipses and other shapes besides rectangles in the svg section. There are many other things you can do with D3, including trees, Bezier curves, zoomable graphs and so on. I would have preferred a few hints on some other things too and less background things I already know. However, the focus on one data set and how to draw a bar chart of it kept the book focussed and it was very quick to read. It was also well written, even if in a somewhat chatty style in places. I enjoyed reading it, and am glad I did. I feel more confident about exploring some of the other examples on the d3 gallery now. I am not sure how often I would need to refer back to this book though. Certainly a good starting point if you've never used D3 before.

Joomla! 3 explained: Your step-by-step guide

By Stephen Burge, published by Addison-Wesley (2012), ISBN: 978-0-321-94322-4, 401 pages, £18.95

Reviewed by Andrew Marlow

The approach this book takes is a step-by-step guide to setting up a joomla site and exploring the functionality of joomla thereby. Explanations tend to be brief and in the context of the aspect of the evolving website. The book needs to be read and followed in conjunction with an environment that will allow joomla to be set up and used for a real website. I did it using the hosting facilities provided by my website provider. This approach makes it different from many books in that it is entirely practical with almost no theory. One has to follow the steps in order since each chapter is a stepping stone for the next. In the preface tha author compares learning joomla via a book to learning to ride a bike or drive a car. He says "A book will help and give some advice, but without actually riding a bike or driving a car, you'll never really learn these skills".

I rate this book as highly recommended.

The lack of theory and lack of precision in some areas has caused some to give this book a negative review but I disagree. In the preface the author says that he wrote the book for his Dad and people like him, i.e. non-technical people. The idea is that everyone, even and indeed especially non-technical people, will be able to install and set up joomla and use it to create a website in such a way as to explore and use all the major facilities of joomla. I think this mission has been a total success.

The book covers the following areas: planning your joomla site, installing and setting up joomla, navigation, the CASh (Categorise, Add, SHow) workflow, content, modules, components, modules, plugins, extensions, templates, users, site management. That is a lot to cover and the distinction between some of these areas is not always made very clear. However, the author is trying to keep the book concise, so some finer technical details are glossed over.

The author covers several different approaches for installing joomla, which will depend on your web hosting environment. This material is very clear and well laid out, with helpful screen snapshots. Indeed, the whole book has very helpful screen snapshots. They really do help, there are not too many of them and they are most certainly not padding, as screenshots in other books so often are.

View from the Chair

Alan Lenton
chair @accu.org

As you're probably aware, if you read my piece in the last *CVu*, (You did read my piece in the last *CVu*, didn't you? It was hidden away on the back page...) I'm preparing a paper on our options as an organization for the future.

As I've been working on this magnum opus, it's starting to become clear to me that that it is not an accident we are having these discussion at this period in time. It seems to me that we are on the cusp of a wide reaching societal change. That change is being driven by a whole range of factors, mainly digital.

That is, I realize a somewhat sweeping claim, so let's look at what is involved. First, industrialized society has now evolved to the stage where it can no longer function without computers.

Second, the computing power available for research – especially in nano-chemistry and biology – is driving new discoveries in the science of materials and our understanding of how our bodies work. Over the next 20 years this will change the way we live out of all recognition.

Third is the rise of digital communications technology – the internet. But the internet as we know it is under threat from politicians, business and malware. I've no idea what the final outcome of this is going to be, but judging from the material that crosses my desk every day, it could be very different from what it is now. Perhaps Balkanization along national boundaries, perhaps the distribution method for big media companies with just limited input available for licensed amateurs similar to the situation with radio media. Perhaps state control of what can go on the net under the creeping (and creepy) rubric of the cry, "Think of the children".

Fourth, in the last ten years the nature of the digital society has significantly changed. Until the turn of the century having a digital device meant that you had a general purpose computer. The significance of a general purpose computer is that you can create programs as well as use programs created by other people.

The start of the 21st century has, however, seen the mass market rise of the digital consumer device. SatNavs, tablets and smart phones to name but a few. What do these have in common? They aren't general purpose computing devices – you can consume digital material using them, but you can't create it. And this is the real digital divide – between those who produce digital material, and those who consume it. The writing was on the wall with the introduction of gaming consoles, but it really didn't become obvious until the advent of tablets. Etch-a-Sketch comes of age!

And where does this leave us as software architects, engineers, and programmers? I'd suggest it puts us in the position of a surfer riding a wave with no idea when and where it's going to go. And that is our problem. ACCU was originally built in the late 80s and the 90s as a forum for enthusiasts who were digitally radicalized by the potential of the fast developing general purpose computers becoming available as commodities.

With the massive expansion of requirements for digital creatives, most of us entered the software business – hobbies became work. Anyone with talent could get into the business regardless of whether they had any formal qualifications in designing and writing software.

In the mid to late 90s there was a similar flowering around the massive expansion of the internet as people learned to create web sites for themselves and for small businesses. Since then there have been mini-booms around apps for smart phones and tablets, but nothing like the same as the earlier stuff. Not surprising really. When you acquire a tablet, it doesn't exactly inspire you to write applications for it, and even if you want to you can't develop them on the tablet – for that you need a general purpose computer. It's not an accident, therefore, that the apogee of ACCU membership was around the early to mid-2000s.

The question facing us now is simply how do we ride the wave? What sort of structures will allow us to recruit new members and move forward in a fashion that will permit flexibility as things change? Because, mark my words, things will change and we have to be prepared for it, or we will fade away like the Cheshire cat, leaving only a grin as a reminder of what fun it used to be.

Bookcase (continued)

Something that is crucial that I missed initially was how important it is to set the sample joomla site up as 'Brochure English'. This is what sets the site up with the initial content and structure and the particular option used has to be chosen in the install because the initial content and structure has done some of the heavy lifting for you.

The book is very easy to follow throughout. The step by step guide with carefully selected screen snapshots and meticulous descriptions of exactly what to do, result in the website building up in a very steady way as the book progresses.

By the time we get to the components chapter things are getting more involved. However, the book still has the step by step approach. At this stage the book would benefit from a bit of an overview before it launches into the steps. It is hard for the newcomer to see where and how things are going.

The section on the news feed component says that RSS stands for Real Simple Syndication. That is actually an unofficial abbreviation. RSS officially stands for Rich Site Summary.

Some part of the book were perhaps a bit too light on detail. For example, I couldn't get the banner tracks statistics to show. This part could have done with a bit more detail to explain what banner tracks means.

The description on adding a news feed shows you how to add an RSS feed that joomla provides but doesn't give any details on how other RSS feeds are typically added.

Joomla currently has some problems parsing RDF feeds. I got this error when I tried to set up an RSS feed for slashdot:

```
No registered feed parser for type rdf:RDF
```

I tried to set up a feed for The Register and this worked even though it is an atom feed rather than RSS. So it would have been worthwhile for the book to mention that joomla does support atom feeds.

There were some aspects where I couldn't quite get things to work. For example, I could not get the bit to work where you change the pages on which modules appear. Joomla has changed its behaviour a little when compared to the book description. Changes such as this are anticipated in the book and mostly make little difference but it looks like it makes a difference here.

As another example, I couldn't get the gmail plugin to work. Perhaps this is because I have two factor authentication enabled. The book was silent on this aspect of login. I think that it ought to mention it, given how important and prevalent two factor authentication is.

Initially, I could not get the jevents calendar display to work properly. It displayed the same event for every day in the month for the months in the range. This turns out to be due to a change that has recently occurred in the implementation. Jevents 1.5 follows the ical method of specifying events (unlinke Jevents 1.4). The start and end date/time relate to a specific repeat so for a 1 day event you set the end date the same as the start date and then use the repeat section to specify the repetitions.

A more technically-minded reader might feel slightly disappointed in certain bits of the book that lack detail or extensive explanations. However, despite this and despite some minor criticisms and problems, the book does an excellent job of helping the reader set up a working joomla website, in a structured logical way that covers most of the functionality of joomla. It is especially good for non-technical readers.