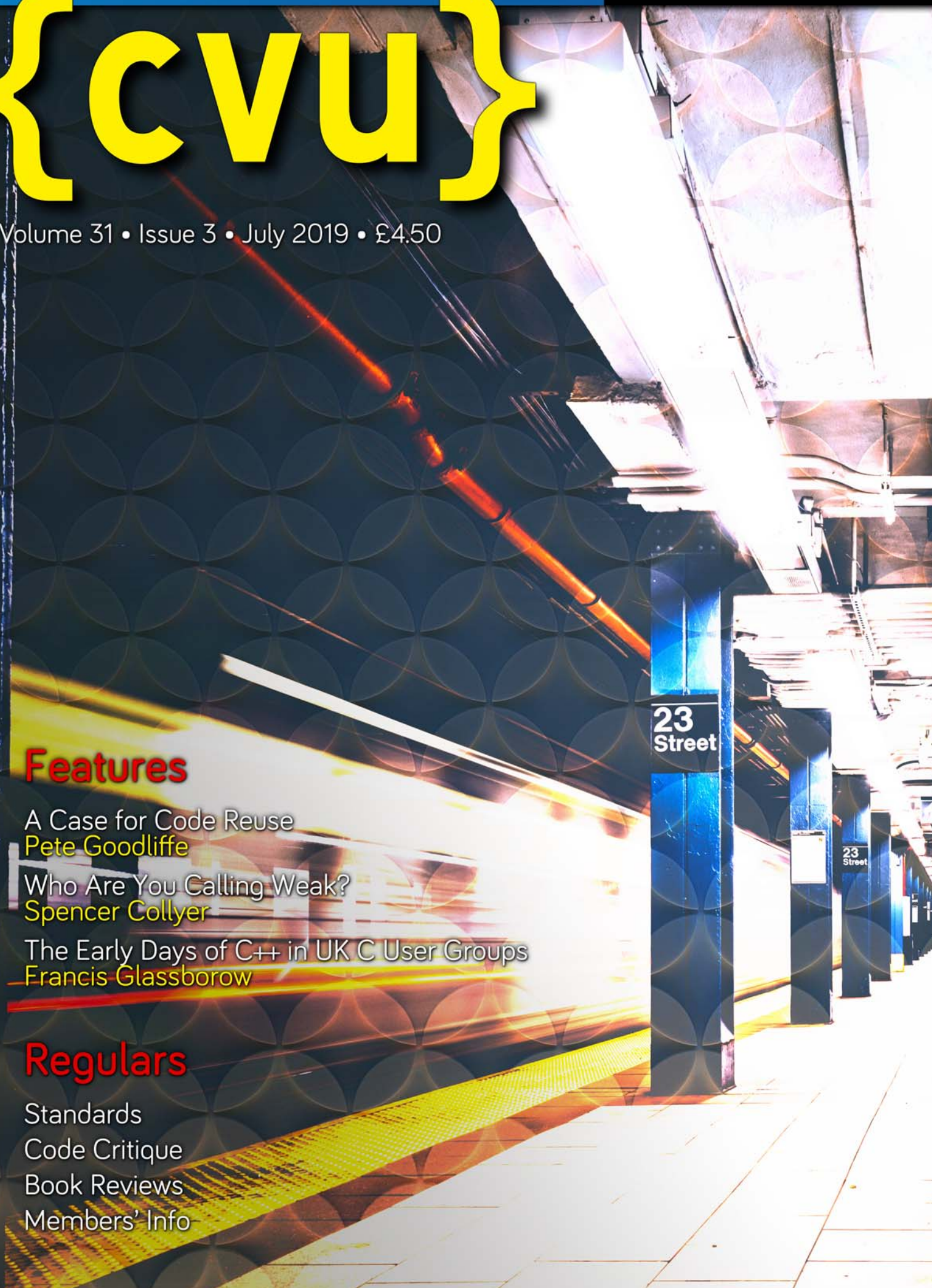
the magazine of the accu

www.accu.org

{cvu}

Volume 31 • Issue 3 • July 2019 • £4.50



Features

Regulars

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

{cvu}
Volume 31 Issue 3
July 2019
ISSN 1354-3164
www.accu.org

WRITEME

As I sit down to write an editorial for *CVu* as guest editor – helping out while Steve Love is moving house – I find myself thinking about communication. I don't mean electronic communication between computers but written and spoken communication between us, the programmers: it's how we share our thoughts and ideas, it's how we collaborate, it's how we learn our trade – and it's how we improve our practice of that trade by picking up tricks and wrinkles from other practitioners who are more experienced than we.

We communicate in lots of different ways: we write books and articles in journals such as this; we write documentation for the projects we work on; we write web pages, wikis, and blogs, that are available through an intranet or through the internet; we use social media, we speak at conferences, and make podcasts and videos; and we have earnest technical conversations around the coffee-machine or the office water cooler – or sometimes in the pub over a few drinks.

Not everyone does all of these things, of course, and some do more than others, but most of us do some of them at some stage in our careers, even if it's only internal project documentation for our employers, or a README file or a github page for an Open Source project. It's a part of what we do, and it is as important that we do it well as it is that we write good code and good unit tests.

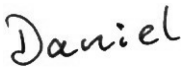
Spoken communication – face-to-face – is the most immediately effective, because it allows for instant feedback. A kind of mental handshaking goes on, that lets the speaker know that the listener has understood, and gives a chance for repetition or rephrasing when the listener seems lost. Unfortunately it's also the least effective in the long term because – unless the listener was taking notes – much of the detail will soon be forgotten.

Written communication is more permanent, but the writer – lacking the benefit of instant feedback that one gets in a spoken conversation – has to work harder to try to ensure that the reader will understand the message: that the writing is clear and unambiguous. Once something is written, though, it can be read and reread by many people and the investment in effort may be repaid many times over.

Of course, if you write nothing you won't succeed in imparting any knowledge to posterity – and if nobody reads your book or your article, nobody reads your blog or watches your video, nobody visits your website or listens to your podcast, then – again – you have communicated nothing.

At one place I worked there was an internal blogging platform on which developers could share tips and write about useful tools and libraries we'd written. The platform was provided but nobody insisted that it be used, there was no code on our timesheets for time spent keeping up to date with work being done, or for writing up one's own work. This meant that hardly anyone read it, and even fewer contributed. It was a great tool, but all it amounted to was a missed opportunity.

Looking around our industry, though, I don't think this is unusual. Our managers want us to keep abreast of technologies and to know what our coworkers are working on but they don't encourage us to share the information and don't provide time for us to do it. As professionals, though, I think it's our duty to do it anyway – it helps keep all our skills honed, and makes us more employable – and it'd be boring not to!



DANIEL JAMES
GUEST EDITOR

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 31.4: 1st August 2019
C Vu 31.5: 1st October 2019

Overload 152: 1st September 2019
Overload 153: 1st November 2019

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

COPYRIGHTS AND TRADE MARKS

A Case for Code Reuse

Pete Goodliffe considers the case for code reuse.

If it can't be reduced, reused, repaired, rebuilt, refurbished, refinished, resold, recycled or composted, then it should be restricted, redesigned or removed from production.
~ Pete Seeger

We hear about a mythical thing called 'code reuse'. For a while it became incredibly fashionable; another software silver bullet, something new for the snake-oil vendors to peddle. I'm not sold on it.

We often talk in terms of 'use cases' when developing software. We also see these *reuse cases*:

Reuse case 1: The copy-pasta

Code copied out of one app is surgically placed into

another. Well, in my book that's less *code reuse* and more like *code duplication*. Or, less politely: *copy-and-paste programming*. It's often evil; tantamount to *code piracy*. Imagine a bunch of swashbuckling programmers pillaging and hoarding software gems from rich codebases around the seven software seas. Daring. But dangerous. It's coding with the bad hygiene of a salty seaman.

Remember the DRY mantra: *do not repeat yourself*.

This kind of 'reuse' is a *real* killer when you've duplicated the same code fragment 516 times in one project and then discover that there's a bug in it. How will you make sure that you find and fix every manifestation of the problem? Good luck with that.

Having said that, you can argue that copy-and-paste between projects actually gets stuff done. There's a lot of it about and the world hasn't come to a crashing end. Yet. And copy-and-paste code avoids the unnecessary coupling which overly DRY code can suffer.

However, copy-and-paste is a nasty business and no self-respecting programmer will admit to this kind of code 'reuse'.

> Avoid copy-and-paste coding. Factor your logic into shared functions and common libraries, rather than suffer duplicated code (and duplicated bugs).

Whilst it is tempting to copy-and-paste code between files in a codebase, it is even more tempting to copy in large sections of code from the Web. We've all done it. You research something online (yes, Google is a great programming tool, and good programmers know how to wield it well). You find a snippet of quite plausible-looking code in a forum or blog post. And you slap it straight into your project to see whether it works. *Ah! That seems to do it.* Commit.

Whilst it's awesome that kind souls provide online tutorials and code examples to enlighten us, it's dangerous to take these at face value, and not apply critical judgment before incorporating them into our work.

Consider first:

- Is the code genuinely *completely* correct? Does it handle all errors properly, or was it only illustrative? (Often we leave error handling and special cases as an *exercise for the reader* when publishing examples.) Is it bug free?
- Is it the best way to achieve what you need? Is it an out-of-date example? Does it come from a really old blog post, containing anachronistic code?
- Do you have rights to include it in your code? Are there any licence terms applied to it?
- How thoroughly have you tested it?

> Don't copy code you find on the Web into your project without carefully inspecting it first.

Reuse case 2: Design for reuse

You design a library from the outset for inclusion in multiple projects. That's *clearly* more theologically correct than yucky copy-and-paste programming. However, I'm sorry: this is not code 'reuse'. It's code *use*. The library was designed to be used like this from the very start!

This approach could also be a huge unnecessary sidetrack.

Even if you suspect that a section of code will be used by more than one project, it's usually not worth engineering it for multiple uses from the start. Doing so can lead to overly complex, bloated software, with high-ceremony APIs that try to cover all general use cases. Instead, employ the *YAGNI* principle: if *you aren't going to need it* (yet), then don't write it (yet).

Focus on constructing the simplest code that satisfies the requirements right now. Write only what's needed, and create the smallest, most appropriate API possible.

Then, when another program wants to incorporate this component, you can add or extend the existing, working code. By only producing the smallest amount of software possible, you will reduce the risk of introducing bugs, or of constructing unnecessary APIs that you'll have to support for years to come.

Often your planned second 'use' never materialises, or the second user has surprisingly different requirements than anyone expected.

Reuse case 3: Promote and refactor

Write small, modular sections of code. Keep it clean and neat.

As soon as you realise that it needs to be used in more than one place, refactor: create a shared library or a shared code file. Move the code in there. Extend the API *as little as possible* to accommodate the second user.

It's tempting at this stage to think that the interface must be dusted off, reworked, and filled out. But that might not be a good idea at all. Aim to keep your changes minimal and simple, because:

- Your existing code works. (It does work well, doesn't it? And you have the tests to prove it?!) Every gratuitous change you make moves it further from this working state.
- It's possible that a third client will appear shortly with slightly different requirements. It would be a shame (as well as a waste of effort) to have to rip up the adjusted API again and adapt it.

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

Who Are You Calling Weak?

Spencer Collyer muses on the surprising strength of `weak_ptr`s.

Management of memory allocated from the heap is one of the fundamental processes in program development. Many programs beyond the simplest toy or test programs will need heap memory to do their work.

Modern programming languages often take the responsibility for handling heap memory away from the programmer, with techniques like garbage collection operating in the background. While this makes life easier in many ways for the programmer, it can make it hard to determine when memory is going to be released, and some GC systems can cause unpredictable slowdowns when the garbage collector runs.

C++ on the other hand exposes heap management to the programmer, giving finer control over when the memory will be allocated and deallocated. The use of raw pointers to allocated memory can lead to well-known problems, with the necessity to take care that allocated memory is released in every possible path through a program, especially in the presence of exceptions. To get around these problems, smart pointer classes have been developed that take control of the ownership of the pointed to memory, so the programmer doesn't need to worry about it.

In the time before the C++11 standard was released, one of the commonest smart pointer classes in use was the Boost **`shared_ptr`** class. This class has been present in Boost since at least the 1.31.0 release (the earliest release for which the Boost website has documentation). Ultimately it formed the basis of the **`std::shared_ptr`** class added to the C++11 standard. This article is about the **`std::shared_ptr`** class, but in general it also applies if you are still using the Boost version.

How shared_ptr works

This section describes how the **`shared_ptr`** objects work under-the-hood. You can skip it if you are already familiar with this.

The primary facility offered by **`shared_ptr`**s is that they allow shared ownership of an object. What this means is that you can have multiple pointers to the object, and as long as at least one is pointing to it, it will not be deleted.

In order to offer this shared ownership facility, the **`shared_ptr`** class uses a reference count for each object pointed to. Each time a **`shared_ptr`** to the object is set, the reference count is incremented. When a **`shared_ptr`** goes out of scope, the reference count is decremented. Only when the reference count goes to zero is the referenced object deleted. Also the memory is usually reclaimed, but see later for why this might not happen immediately.

The reference count for an object obviously has to be stored somewhere. Some classes that use reference counting to handle object ownership are intrusive in that they store the reference count within the owned object itself. The **`shared_ptr`** class is non-intrusive, which means they don't need to do that. Instead they use a control block that is allocated separately to the memory allocated for the pointed-to object. Figure 1(a) illustrates this.

The make_shared template

Allocating memory is generally an expensive operation, so allocating two blocks of memory when creating a new object pointed to by a **`shared_ptr`** effectively doubles the time to carry out the operation. To avoid this double allocation, the template **`make_shared`** can be used. The C++11 standard states that **`make_shared`** should perform no more than one memory allocation.

Figure 1(b) illustrates what this would look like in memory. As **`make_shared`** has other advantages over using **`new`**, such as better exception safety, you would generally prefer to use it.

SPENCER COLLYER

Spencer has been programming for more years than he cares to remember. In his younger years, he worked on projects as diverse as monitoring water treatment works and television programme scheduling. For the last 20-odd years, he has mainly worked for banks in the City of London and Canary Wharf.

A Case for Code Reuse (continued)

> Code should be 'shared' because it is useful to multiple clients, not because the developers want to create a nifty shared library.

Reuse case 4: Buy in, or reinvent the wheel

When you need to add a new feature, there may already be third-party libraries available that provide the functionality. Carefully consider whether it is economically more sensible to roll your own code, to pull in an open source version (if license terms permit), or to buy in a third-party solution with vendor support.

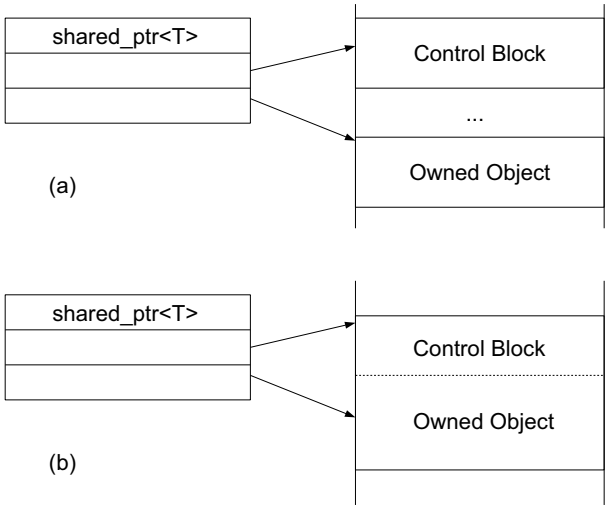
> Don't dismiss other people's code. It may be better to use existing libraries rather than write your own version.

You need to weigh the ownership costs against build costs, the likely code quality, and the ease of integration and maintenance of each solution. Developers tend to want to write things themselves, not just for the intellectual exercise, but also due to a distrust of the unknown. Make an informed decision. ■

Questions

- How much duplication is there in your codebase? How often do you see code copied and pasted between functions?
- How can you determine how different sections of code have to be before it is acceptable to consider it *not* duplication, and to not try to refactor the versions together?
- Do you often copy code examples from Stack Overflow into your work? How much effort do you invest in 'sanitising' the code for inclusion? Do you mercilessly update the layout, variable names, etc.? Do you add tests?
- When you add code from the web, should you place comments around it stating the source of the implementation? Why?

Figure 1



It is worth being aware of this behaviour of the **make_shared** template with regards to increasing the size of the block allocated when creating a new object on the heap. For instance, if you are using a memory allocator that uses slab allocation, you need to take account of the size of the control block when working out which slab size the object will use if you use a **shared_ptr** to point to it.

Examples of shared_ptr memory usage

The following examples illustrate how memory is allocated when creating an object and using a **shared_ptr** to store the address. The code was compiled on a 64-bit Arch Linux system using GCC 8.3.0.

The file `common.ipp` (shown in Listing 5 on page 7) is used in all examples to provide overrides for global new and delete operators so we can log when memory is allocated and deallocated, and the **DataHolder** class for the objects we create. This is a 'large' class so it is too big for any small-object optimization that the implementation of **shared_ptr** may use.

Creation with operator new

The code in Listing 1 allows us to examine how memory is allocated and dealloacted when creating an object on the heap using **new** and storing the returned pointer in a **shared_ptr**.

Sample output from this code is shown in Figure 2.

Line 2 is the memory allocation for the **DataHolder** object, and it is constructed at line 3.

Line 4 is the memory allocation for the **shared_ptr** control block. As can be seen in line 5, the **shared_ptr** is pointing to the memory allocated at line 2, i.e. the memory for the object.

The following three lines show the destruction of the **DataHolder** followed by the deallocation of the memory previously allocated. Note that the memory for the object is deallocated before the memory for the control block.

Line 9 shows that the object destruction and memory deallocation occurs at the end of the inner block, as would be expected by the **shared_ptr** going out of scope when the inner block ends.

Creation with make_shared

The code in Listing 2 allows us to examine how memory is allocated and dealloacted when creating an object on the heap using **make_shared**.

Sample output from this code is shown in Figure 3.

Comparing this to Figure 2 shows some obvious differences.

Firstly, there is only one allocation of memory from the heap, at line 2. The memory allocated is not equal to the size of the control block and object block from the previous example because there is some optimization of memory usage occurring.

Listing 1

```
#include "common.ipp"
#include <memory>
int main()
{
  {
      std::cout << LINENO
        << "Creating from pointer\n";
      auto p1 = std::shared_ptr<DataHolder>
        {new DataHolder};
      std::cout << LINENO
        << "Finished creating from pointer, "
        "use_count=" << p1.use_count()
        << ", object @ " << std::hex << p1.get()
        << std::dec << "\n";
  }
  std::cout << LINENO << "Exiting program\n";
}
```

Figure 2

```
1: Creating from pointer
2: Allocated 84 bytes at address 0x0x55a792b45e88
3: Constructing DataHolder 1
4: Allocated 24 bytes at address 0x0x55a792b45ef8
5: Finished creating from pointer, use_count=1,
   object @ 0x55a792b45e88
6: Destroying DataHolder 1
7: Deallocating 84 bytes at address
   0x0x55a792b45e88
8: Deallocating 24 bytes at address
   0x0x55a792b45ef8
9: Exiting program
```

Secondly, on line 4, you can see that the **shared_ptr** does not point to the start of the block allocated in line 2, but part way through. This is because the initial bytes of the memory are used by the control block, and the object pointed to starts after the control block.

Finally, line 6 shows there is just one memory deallocation, matching the single memory allocation from line 2, and line 7 shows that the object destruction and memory deallocation occur at the end of the inner block.

Listing 2

```
#include "common.ipp"
#include <memory>
int main()
{
  {
    std::cout << LINENO
      << "Creating with make_shared\n";
    std::shared_ptr<DataHolder> p2 =
      std::make_shared<DataHolder>();
    std::cout << LINENO
      << "Finished creating with make_shared, "
      "use_count=" << p2.use_count()
      << ", object @ " << std::hex << p2.get()
      << std::dec << "\n";
  }
  std::cout << LINENO << "Exiting program\n";
}
```

Figure 3

```
1: Creating with make_shared
2: Allocated 104 bytes at address
   0x0x560de08cbe88
3: Constructing DataHolder 1
4: Finished creating with make_shared,
   use_count=1, object @ 0x560de08cbe98
5: Destroying DataHolder 1
6: Deallocating 104 bytes at address
   0x0x560de08cbe88
7: Exiting program
```

Using weak_ptr

The automatic deallocation of the object pointed to when the last reference to it goes out of scope is a major advantage of **shared_ptr**. However, problems arise if you have two objects which have **shared_ptr** members and which directly or indirectly reference each other. This is because the **shared_ptr** members will only be destroyed, and hence decrease the reference count of their pointed-to objects, when the containing object is destroyed. But if object A has a **shared_ptr** pointing to object B, and object B has a **shared_ptr** pointing to object A, the reference counts in the respective **shared_ptr**s will never be decremented to 0, so A and B will never be deleted.

For instance, if you have a data structure representing a graph where the pointers between nodes in the graph are **shared_ptr**s, as soon as you have a cycle in the graph, the graph objects in the cycle will never be deleted.

One way to get around this problem is to explicitly call **reset()** on the **shared_ptr** member. This causes it to no longer point to an object, so breaking the cycle explicitly.

Another way to break the cycle is to use **weak_ptr** instead of **shared_ptr**. A **weak_ptr** provides *potential* ownership, not *actual* ownership. What this means is that if the only pointers pointing to an object are **weak_ptr**s, the object can be deleted.

In order to actually use an object pointed to by a **weak_ptr** you have to call **lock** on the pointer to obtain a **shared_ptr** to the object, and from then you just treat it as normal. If the result of the **lock** call is a **shared_ptr** that points to nothing, it means the object pointed to has been destroyed, and cannot be used.

How weak_ptr works

So that the **lock** call can work, there has to be some bookkeeping data associated with an object that exists as long as there is a **weak_ptr** that could potentially refer to the object. This bookkeeping data is generally kept in the control block used by the **shared_ptr** to hold the reference count.

An additional count is held, indicating how many **weak_ptr**s point to the object. As long as the weak reference count is above zero the control block will not be deleted, even if the pointed-to object is deleted as a result of the normal reference count going to zero.

Effect of weak_ptr on allocated memory

The effect of using **weak_ptr**s on allocated memory usage can be seen in the following examples.

Creation using operator new

The code in Listing 3 shows how using **weak_ptr** affects memory used by a **shared_ptr** created using **new**.

Sample output from this code is shown in Figure 4. Compare this to the corresponding output with no **weak_ptr** given in Listing 2.

Lines 1–5 are identical except for the memory addresses. The only thing of note here is that the initial creation of the **weak_ptr** does not allocate any memory, because **weak_ptr**s do not have their own control blocks.

Lines 6 and 7 show the **weak_ptr** being assigned from the **shared_ptr**. As expected, the reference count is not updated after the **weak_ptr** is pointed to the object. Under the hood the weak reference count will have been updated, but the public interface of **shared_ptr** does not allow access to the weak reference count to confirm this.

Problems arise if you have two objects which have shared_ptr members and which directly or indirectly reference each other

Listing 3

```
#include "common.ipp"
#include <memory>
int main()
{
  std::weak_ptr<DataHolder> wp1;
  {
    std::cout << LINENO
      << "Creating from pointer\n";
    auto p1 = std::shared_ptr<DataHolder>
      {new DataHolder};
    std::cout << LINENO
      << "Finished creating from pointer, "
      "use_count=" << p1.use_count()
      << ", object @ " << std::hex << p1.get()
      << std::dec << "\n";
    std::cout << LINENO
      << "Assigning to weak_ptr\n";
    wp1 = p1;
    std::cout << LINENO
      << "After assigning to weak_ptr, "
      "use_count=" << p1.use_count() << "\n";
  }
  std::cout << LINENO << "Exiting program\n";
}
```

Figure 4

```
 1: Creating from pointer
 2: Allocated 84 bytes at address 0x0x55d7616a7e88
 3: Constructing DataHolder 1
 4: Allocated 24 bytes at address 0x0x55d7616a7ef8
 5: Finished creating from pointer, use_count=1,
    object @ 0x55d7616a7e88
 6: Assigning to weak_ptr
 7: After assigning to weak_ptr, use_count=1
 8: Destroying DataHolder 1
 9: Deallocating 84 bytes at address
    0x0x55d7616a7e88
10: Exiting program
11: Deallocating 24 bytes at address
    0x0x55d7616a7ef8
```

The interesting part is lines 8–11. Lines 8 and 9 show that when the inner block is exited, the **DataHolder** that was created is destroyed and its memory released just as happens in lines 6 and 7 in Listing 2. But for the control block, if you compare lines 10 and 11 to Listing 2 lines 8 and 9, you will see that it is not deallocated when we exit the inner block. This is because the weak reference count is not zero as the **weak_ptr** is still referencing it. Only when the program exits and the **weak_ptr** is destroyed, setting the weak reference count to zero, is the control block destroyed.

Creation using make_shared

The code in Listing 4 (overleaf) shows how using **weak_ptr** affects memory used by a **shared_ptr** created using **make_shared**.

Sample output from this code is shown in Figure 5. Compare this to the corresponding output with no **weak_ptr** given in Figure 3.

Once again, the initial lines 1–4 from the two outputs are identical, and lines 5 and 6 in Figure 5 show that assigning a **shared_ptr** to a **weak_ptr** doesn't affect the use count of the **shared_ptr**.

The interesting part is again when the **shared_ptr** goes out of scope at the end of the inner block, leading to the **DataHolder** being destroyed. As can be seen at line 7, the **DataHolder** is destroyed when the inner block is exited. Once again the memory allocated is not released until the program exits and the **weak_ptr** is destroyed, so releasing the control block.

Listing 4

```
#include "common.ipp"
#include <memory>
int main()
{
  std::weak_ptr<DataHolder> wp2;
  {
    std::cout << LINENO
      << "Creating with make_shared\n";
    std::shared_ptr<DataHolder> p2
      = std::make_shared<DataHolder>();
    std::cout << LINENO
      << "Finished creating with make_shared, "
      "use_count=" << p2.use_count()
      << ", object @ " << std::hex << p2.get()
      << std::dec << "\n";
    std::cout << LINENO
      << "Assigning to weak_ptr\n";
    wp2 = p2;
    std::cout << LINENO
      << "After assigning to weak_ptr, "
      "use_count=" << p2.use_count() << "\n";
  }
  std::cout << LINENO << "Exiting program\n";
}
```

Figure 5

```
1: Creating with make_shared
2: Allocated 104 bytes at address
   0x0x562fb038fe88
3: Constructing DataHolder 1
4: Finished creating with make_shared,
   use_count=1, object @ 0x562fb038fe98
5: Assigning to weak_ptr
6: After assigning to weak_ptr, use_count=1
7: Destroying DataHolder 1
8: Exiting program
9: Deallocating 104 bytes at address
   0x0x562fb038fe88
```

Importantly, note that even though the memory for the referenced object is not needed once the last **shared_ptr** pointing to it has gone out of scope, it cannot be released until the control block is also destroyed. This is because the allocated memory block cannot be partially released, but only as a complete block.

Conclusion: weak_ptrs are stronger than you might think

As can be seen, **weak_ptrFigure 5**s are only 'weak' in the sense of not preventing an object being destroyed while they still point at its control block. They still prevent some or all of the memory allocated when creating the object from being released until all the **weak_ptr**s pointing to it have been destroyed.

Whether this is an important consideration when designing your program is obviously dependent on a number of factors.

If you are in an environment with huge amounts of memory, where memory consumption is only a secondary concern, you probably wouldn't need to worry about it.

If you just use **weak_ptr**s to hold parts of a logical structure together, such as a graph where you use **weak_ptr**s to hold some of the links between the nodes, and the whole structure is destroyed in one go, you probably won't be concerned because all the **weak_ptr**s will be destroyed at the same time, so no memory will be held onto.

On the other hand, if you are working in a restricted memory environment where you need control over how much memory is allocated at any time, you would need to be aware of how **weak_ptr**s hold onto memory, even if it is just the control block associated with the objects pointed to, and more so if you use **make_shared** to create the objects in the first place. ■

Listing 5

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <new>

int lineno=0;
#define LINENO std::setw(2) << ++lineno << ": "

void* operator new(std::size_t sz)
{
  auto ptr
    = std::malloc(sz + sizeof(std::size_t));
  std::size_t* iptr
    = static_cast<std::size_t*>(ptr);
  *iptr = sz;
  ++iptr;
  ptr = static_cast<void*>(iptr);
  std::cout << LINENO << "Allocated " << sz
    << " bytes at address 0x" << std::hex << ptr
    << std::dec << "\n";
  return ptr;
}
void operator delete(void *ptr) noexcept
{
  std::size_t* iptr
    = static_cast<std::size_t*>(ptr);
  --iptr;
  std::cout << LINENO << "Deallocating "
    << *iptr << " bytes at address 0x"
    << std::hex << ptr << std::dec << "\n";
  ptr = static_cast<void*>(iptr);
  std::free(ptr);
}
struct DataHolder
{
  DataHolder()
  : num(++dh)
  {
    std::cout << LINENO
      << "Constructing DataHolder "
      << num << "\n";
  }
  ~DataHolder()
  {
    std::cout << LINENO
      << "Destroying DataHolder " << num << "\n";
  }
  int num;
  int i[20];
  static int dh;
};
int DataHolder::dh=0;
```

if you are working in a restricted memory environment where you need control over how much memory is allocated at any time, you would need to be aware of how weak_ptrs hold onto memory

The Early Days of C++ in UK C User Groups

Francis Glassborow looks back on the formation of the ACCU.

From about 1990, I had started to include material on C++ in *C Vu*. I was also sneaking the '++' into the consciousness of the membership. For a time I slipped a '++' inside the 'C' of *C Vu* (an ancient battle, long lost but there was meant to be a space between the 'C' and 'Vu').

I had excellent contacts with most of the main producers of compilers and in particular with Borland. In autumn 1992, my Borland contacts alerted me to a proposal from a group of enthusiasts to create a Borland C++ User Group. That seemed a little silly to me so I contacted them and suggested that they might like to be a sub-group of CUG(UK) as it then was. After quite a bit of negotiation, they decided to give the idea a try.

Unlike the founders of CUG(UK) whose newsletter *C Vu* was originally almost entirely reprints from the US, the Borland C++ enthusiasts had spent many months putting together the first issue of a magazine that they proposed to call *Overload*. One problem was that they had put all their planning into the first issue and had not given much thought to subsequent issues.

We launched *Overload* in April 1993 and the editorial control was firmly in the hands of the nascent Borland C++ User group. It soon became clear that the limitation to a single tool provider was not what people wanted. They wanted a robust publication on C++ even if the Borland compiler was the one they were using. At that time, the Borland Compiler was ahead of the Microsoft one in that it tried to track the developments coming out of WG21. There were quite a few other contenders. Most have withered away.

There was always a look of disbelief on the faces of prospective organisers when we claimed we could provide top speakers from around the world.

As the first year progressed, the small group of enthusiasts began to lose interest in the actual production of a magazine and it became increasingly reliant on the more general membership, at least those who wanted to read about C++ and were happy to pay a higher membership fee to include that publication.

Coincidentally another C++ user group was testing the waters in 1992–93. That was the European C++ User Group. Their main motivation was to run economically priced conferences for C++ users. I submitted a speaking proposal for their second conference which was scheduled to precede the WG21 meeting that was in July 1993 in Munich. I have forgotten exactly what my talk was on and it is probably best forgotten. The content was pretty naïve and it was my first venture as a conference speaker. However, attending the event was life changing. I was well entertained and the evening spent with Frank Buschmann (our host) Bjarne Stroustrup and Alan Bellingham in the English Beer Garden was memorable not least for Bjarne insisting that he bought a round after we had all imbibed 3 Steins of excellent German beer. We did eventually manage to get back to the hotel thanks to the excellent Munich public transport system.

The next day, three of us (I think Alan was planning to return to England – by car – and so missed out) together with Josée Lajoie (a wonderful Canadian academic and, in those days, chair of one of the three WG21 core work groups) went off to the Kloster Andechs Biergarten located within the Andechs monastery about 40km south west of central Munich. It was a lovely summer day and we had a relaxed and enjoyable time there. Josée was driving but we only drank in moderation. It was there that Bjarne, on learning that I was reviewing a 1500-page book on Borland C++ opined that you could not write a book on C++ over a thousand pages. The type face was larger than that in his book so maybe it was not an entirely fair comparison. I reminded him of that claim when the 4th edition of *The C++ Programming Language* substantially exceeded 1000 pages. As C++ has grown so has the page count for that book.

Sadly, EC++UG struggled and the Munich conference was its last. I offered to take over its membership, allowing them a certain number of free issues of *C Vu* and, I think, *Overload*. The membership records were chaotic so I am not surprised that it struggled. However, the result of injecting a nucleus of a hundred-plus hardened C++ enthusiasts/professionals was that *Overload* now had a solid readership.

In absorbing EC++UG, it had always been my intent to honour the EC++UG conference plan. It was to take a while to achieve but when WG21 was due met in London (well, it is listed as Cambridge but I am sure it was actually in London though hosted by Cambridge Analytica) in July 1997, I arranged that we would have a conference in Oxford. The event was held in Oxford Town Hall on July 18/19. July 18 (Friday) was a multi-track event. I do not recall much of the content. Saturday was a single track with Bjarne Stroustrup, Tom Plum, Bill Plauger (I think) and Dan Saks giving an excellent set of talks well pitched at the audience.

One very important decision right from the start was that ACCU (as it had now become) would provide the speakers but the administration/organisation had to be done by professionals who would keep the money. We went through several organisers over the early years. There was always a look of disbelief on the faces of prospective organisers when we claimed we could provide top speakers from around the world.

Some would be delighted to do the organising and charge us for it. That very definitely was not what we wanted. We did not want conference moneys flowing through our books with the result that we would become liable for VAT on everything we did. Volunteer treasurers do not want the complexity and legal liability that would entail.

The Borland C++ User Group and the European C++ User Group are distant ghosts in our past. Nonetheless they live on in ACCU and the ACCU Conference as we have successfully absorbed them and, I hope, fulfilled what each set out to do. ■

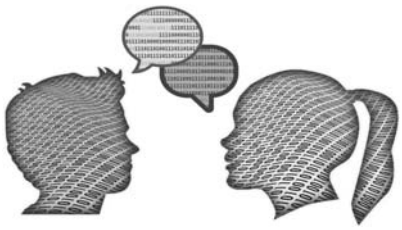
FRANCIS GLASSBOROW

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

If you read something in *CVu* that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

You can contact us at cvu@accu.org

Code Critique Competition 118



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

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

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

Last issue's code

I'm trying to do some simple statistics for some experimental data but I seem to have got an error: I don't *quite* get the results I am expecting for the standard deviation. I can't see what's wrong: I've checked and I don't get any compiler warnings. I ran a test with a simple data set as follows:

```
echo 1 2 3 4 5 6 7 8 9 | statistics
mean: 5, sd: 1.73205
```

The mean is right but I think I should be getting a standard deviation of something like 2.16 for the set (without outliers this is [2,8].)

Can you solve the problem – and then give some further advice?

Listing 1 contains `statistics.h` and Listing 2 is `statistics.cpp`.

Listing 1

```
namespace statistics
{
  // get rid of the biggest and smallest
  template <typename T>
  void remove_outliers(T& v)
  {
    using namespace std;
    auto min = min_element(v.begin(), v.end());
    auto max = max_element(v.begin(), v.end());
    v.erase(remove_if(v.begin(), v.end(),
      [min, max](auto v) {
        return v == *min || v == *max;
      }),
      v.end());
  }
  template <typename T>
  auto get_sums(T v)
  {
    typename T::value_type sum{}, sumsq{};
    for (auto i : v)
    {
      sum += i;
      sumsq += i * i;
    }
    return std::make_pair(sum, sumsq);
  }
  template <typename T>
  auto calc(T v)
  {
    remove_outliers(v);
    auto sums = get_sums(v);
    auto n = v.size();
    double mean = sums.first / n;
    double sd = sqrt((sums.second -
      sums.first * sums.first / n) / n - 1);
    return std::make_pair(mean, sd);
  }
}
```

Listing 2

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <iterator>
#include <vector>
#include "statistics.h"
void read(std::vector<int>& v)
{
  using iter = std::istream_iterator<int>;
  std::copy(iter(std::cin), iter(),
    std::back_inserter(v));
}
int main()
{
  std::vector<int> v;
  read(v);
  auto result = statistics::calc(v);
  std::cout << "mean: " << result.first
    << ", sd: " << result.second << '\n';
}
```

Critique

Dave Simmonds <daveme@ntlworld.com>

`remove_outliers` will run down the container 3 times, which could be costly on a large dataset – we'll come back to this later.

`remove_outliers` actually removes the outliers from its input. That sounds OK, and looking at the way `calc` takes a copy, it works here. But we could be more efficient by taking a const ref and not changing the input. More later.

The lambda in `remove_outliers` reuses the variable `v`. This is OK but confusing. We could call it `e` for element.

In `calc`, the formula for the standard deviation is the version which means we only have to run down the container once. But as mentioned above we've already gone down it three times in `remove_outliers`, so this could be better. We actually could run down the container only once, keeping track of and counting outliers and adding up `sum` and `sumsq`, then removing the amount for the outliers at the end. Code below.

Now for the issue that gives us the wrong result: the formula is trying to divide by n-1, but actually is dividing by n and then subtracting 1. So we need to change `n - 1` to `(n - 1)`

And now another issue. All our calculations here are based on integers, we will get rounding down to nearest integer – which will give us the answer 2 in the example. We need to convert to `double` before we do any division.

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

The line(s) in question becomes:

```
double sd = sqrt((sums.second -
  (double)(sums.first * sums.first) / n)
  / (n - 1));
```

This also applies to the mean calculation, which should be:

```
double mean = (double)sum / n;
```

The next issue is what if n is less than 2 – we will divide by zero – so there needs to be a check for that.

Here's my version of `statistics.h`, which addresses all these issues:

```
namespace statistics
{
  template <typename T>
  auto calc(T const & v)
  {
    typename T::value_type sum{}, sumsq{};
    typename T::value_type min{}, max{};
    int min_count{}, max_count{};
    // run down the container just once,
    // collecting what we need
    for (auto i : v)
    {
      if (!min_count || i < min)
      {
        min = i;
        min_count = 1;
      }
      else if (i == min)
      {
        ++min_count;
      }
      if (!max_count || i > max)
      {
        max = i;
        max_count = 1;
      }
      else if (i == max)
      {
        ++max_count;
      }
      sum += i;
      sumsq += i * i;
    }
    auto n = v.size();
    // now we remove the outliers
    if (min_count)
    {
      sum -= min*min_count;
      sumsq -= min*min*min_count;
      n -= min_count;
    }
    if (max_count && max != min) // check if
    // max == min, if so we have already
    // removed it above
    {
      sum -= max*max_count;
      sumsq -= max*max*max_count;
      n -= max_count;
    }
    // check we have enough input
    if (n < 2)
    {
      throw(std::runtime_error(
          "not enough data"));
    }
    // and do the final calcs
    double mean = (double)sum / n;
    double sd = sqrt((sumsq -
      (double)(sum * sum) / n) / (n - 1));
      return std::make_pair(mean, sd);
    }
  }
```

James Holland <James.Holland@babcockinternational.com>

As the student mentions, the compiler issues no warnings. This suggests that there is something wrong with the program logic. It turns out that the line of code that calculates the standard deviation, **`sd`**, is in error. Replacing the statement with the following code results in the correct answer for the input provided.

```
double variance = 0.0;
for (auto x : v)
{
  variance += std::pow(x - mean, 2);
}
variance /= n - 1;
double sd = sqrt(variance);
```

Unfortunately, the program is still not correct as demonstrated by entering the test data again but omitting the first number, (1). Removing the outliers leaves [3 … 8] which has a mean of 5.5 and a standard deviation of 1.87083. The program gives a mean of 5 and a standard deviation of 1.94936. Something is clearly still wrong. It turns out that the mean is being calculated incorrectly. It is, in effect, being rounded down to the nearest integer. This is because the sum, an integer, is being divided by the number of samples, another integer. The result is also an integer. The integer is then converted to a **`double`** and used to initialise **`mean`**. But it is too late; the fractional part of the mean has already been lost. The reason why the correct answer was produced with the original test data is that the mean just happened to be an integer value. To correct the problem I suggest casting **`sums.first`** to a **`double`** before dividing by the number of samples. The program should now work correctly with any input.

The general advice one receives is not to hand-craft loops oneself but to make use of algorithms already provided by the C++ standard library. The original code already uses some library function, such as **`std::copy()`**, but let's see what else the library has to offer.

Instead of using the two library functions **`min_elenent()`** and **`max_element()`**, the single function **`minmax_element()`** can be used with a slight gain in efficiency. This function returns a **`std::pair`** of iterators, the first pointing to an element with the smallest value and the second pointing to an element with the largest value. The **`std::pair`** of iterators can then be captured by the lambda of the **`remove_if()`** function and used by selecting the first and second elements of the pair.

It is possible to get away without actually erasing the 'removed' elements of the vector. We could modify **`remove_outliers()`** to return the logical end of the vector as returned by **`remove_if()`**. The logical end of the vector can be used in further processing instead of the actual end of the vector. The small amount of execution time gained may be beneficial in some circumstances.

We now turn are attention to **`get_sums()`**. This function contains a single loop that calculates the sum and the sum of the squares. While there is nothing disastrously wrong with the code, the loop could be replaced by two library functions as shown in the listing below. Whether using the library makes the code more readable is debatable, however. At least the number of lines of code in the **`get_sums()`** function has been reduced from seven to three.

It is also possible to make use of the standard library in calculating the standard deviation, **`sd`**. Instead of employing the code shown at the beginning of this text, the standard library function **`accumulate()`** can be used to calculate the square of the standard deviation. This value is called the variance. Obtaining the square root of the variance, therefore, gives the standard deviation as shown in the complete listing of my version of the `statistics.h`.

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

```
namespace statistics
{
  // Get rid of the biggest and smallest.
  template <typename T>
  auto remove_outliers(T & v)
  {
    const auto min_max =
      std::minmax_element(v.begin(), v.end());
    return std::remove_if(v.begin(), v.end(),
     [min_max](auto element) {
     return element == *min_max.first ||
       element == *min_max.second;});
  }
  template <typename T>
  auto get_sums(const T & v,
    typename T::const_iterator logical_end)
  {
    const auto sum =
      std::accumulate(v.begin(), logical_end,
        typename T::value_type{});
    const auto sumsq =
      std::inner_product(v.begin(),
      logical_end, v.begin(),
      typename T::value_type{});
    return std::make_pair(sum, sumsq);
  }
  template <typename T>
  auto calc(T & v)
  {
    const auto logical_end =
      remove_outliers(v);
    const auto sums =
      get_sums(v, logical_end);
    const auto n =
      distance(v.begin(), logical_end);
    const auto mean =
      static_cast<double>(sums.first) / n;
    const auto variance =
      std::accumulate(v.begin(), logical_end,
      0.0, [& mean](auto a, auto b){
        return a + std::pow(b - mean, 2);})
        / (n - 1);
    const auto sd = sqrt(variance);
    return std::make_pair(mean, sd);
  }
}
```

Incidentally, when I was experimenting with various ways of rewriting **remove_outliers()**, I came up with solutions such as that shown below.

```
std::vector<int> v;
const auto [min, max] =
  std::minmax_element(v.begin(), v.end());
std::remove_if(v.begin(), v.end(),
  [min, max](auto element) {
    return element == *min ||
      element == *max;});
```

This code would not compile with Clang although the following would.

```
std::vector<int> v;
const auto [minimum, maximum] =
  std::minmax_element(v.begin(), v.end());
const auto min = minimum;
const auto max = maximum;
std::remove_if(v.begin(), v.end(),
  [min, max](auto element) {
    return element == *min ||
      element == *max;});
```

Both versions compiled without complaint with GCC and Microsoft. Does this represent a bug in the Clang compiler?

Editor: this was a troublesome issue where compilers were found to disagree in their interpretation of the wording in C++17. Initially p0588r1 in Nov 2017 made it illegal, pending further discussion; p1091r3 in Feb 2019 removed this restriction. However, these both only apply to the working paper until a new C++ standard is published.

Hans Vredeveld <accu@closingbrace.nl>

The first thing we notice is that `statistics.h` misses an include guard and that it makes use of several standard library functions, but that it does not have any **#include**s. It should include **<algorithm>**, **<utility>** and **<cmath>**, so that it is self contained and can be used easily in other implementations than `statistics.cpp`. The include of <cmath> can then be removed from `statistics.cpp`. (**<algorithm>** still needs to be included because of the use of **std::copy**.) Note that an include of **<utility>** was missing altogether. In the implementations of the standard library that I have installed (GCC 7.4.1's libstdc++, and LLVM 5.0.1's libc++) **<algorithm>** includes **<utility>**, and I guess that in other implementations it will be similar. Still, it looked more like an implementation detail to me than anything else, so we should not rely on it. Putting the includes for the project's own headers before the includes for third party and standard library headers would have caught at least part of these issues.

In the three function templates in the header there are two recurring issues. First, naming of identifiers. The template parameter in all three templates is **T**. This doesn't convey any information about what it represents. When reading the code, it becomes clear that it stands for a container type, so I suggest naming it **Container** instead. Similar, I suggest renaming the function parameter **v** to container in all three function templates. In **remove_outliers**, the lambda's parameter **v** is different from, and shadows the function parameter **v**. In this case, **v** stands for the value of a container element, and I suggest renaming it to **value**.

The other recurring issue is that of taking things by value instead of by reference. The lambda in **remove_outliers** and the **for**-loop in **get_sums** both operate on elements of the container. In the generic case of templates, where we don't know anything about the element size, it is better to use const references to the elements. If the element size is small, this has no, or a small impact on the generated code (depending on the compiler and compiler options). If the element size is large, this avoids some expensive copying.

get_sums and **calc** take a container as argument. In all but the most trivial cases this will be a large object that is expensive to copy. **get_sums** could easily take its argument as **const&**. For **calc**, it is a bit more complicated. **calc** calls **remove_outliers**, which operates on the container in place. We could have **calc** take its argument as non-const reference, but then **calc** would have some unexpected side effects, where calling **calc** twice with the same container would give different results. We can improve on this by changing **remove_outliers** so that it doesn't change its argument in place, but only reads from it and creates a new container with the outliers removed. More about that in a moment.

The body of **remove_outliers** starts with a using directive for the **std** namespace. I prefer to write using declarations for the things that I need (**min_element**, **max_element** and **remove_if**), or use them with their fully qualified names. It will save me from the compiler choosing an overload that I didn't intend to be used, now or in the future when compiling with a new version of the library. Note that the **using** directive is already useless if we want to use **std::min** or **std::max**, because that conflicts with the local variables **min** and **max**.

In the next two statements, **std::min_element** and **std::max_element** are used to determine the minimum and maximum value. This means going through the container twice. We can reduce this to once by using **std::minmax_element**. The result of **std::minmax_element** is a pair of **const_iterator**s and it would be nice to use a structured binding to capture the result. Unfortunately, we cannot use the names introduced by a structured binding in a lambda capture. To circumvent this problem we use **std::tie** (and include the header **<tuple>**) with previously declared **min** and **max**.

The rest of the function is a single statement that removes the minimum and maximum values in place. As mentioned before, it would be nice if **remove_outliers** and **calc** could take their argument as **const auto&**. We can achieve this by copying all elements we want to keep (using **std::copy_if**) into a new container and return that from the function. Putting it all together, **remove_outliers** now becomes

```
template <typename Container>
Container remove_outliers(
  const Container& container)
{
  typename Container::const_iterator min, max;
  std::tie(min, max) =
    std::minmax_element(container.begin(),
      container.end());
  Container result{};
  std::copy_if(container.begin(),
    container.end(),
    std::back_inserter(result),
    [min, max](const auto& value) {
      return value != *min && value != *max;
    });
  return result;
}
```

There's not much to tell about **get_sums** that isn't told already, so let's move on to calc. In the calculations of **mean** and **sd** we see repetitive use of **sums.first** and **sums.second**. This doesn't tell us much. If we replace **sums.first** by **sum** and **sums.second** by **sumsq**, it becomes more informative. To do so, we have to put the result of **get_sums** in a structured binding **[sum, sumsq]**.

With a template type of **std::vector<int>**, as is the case in `statistics.cpp`, **sum** and **sumsq** have the type **int**. The type of **n** is the implementation defined **std::vector<int>::size_type**, which in most, if not all, implementations is either **unsigned int** or **unsigned long**. In either case the usual arithmetic conversions apply and **sum** and **sumsq** are converted from a signed integral type to an unsigned integral type, changing the value of **sum** if it was negative.

In the calculation of **mean**, the division is performed on the unsigned operands, after which the result is converted to **double**. For a proper calculation, the operands have to be converted to **double** before the division. In the calculation of **sd** we have a similar issue. One way to solve this is to declare **n** of type **double** (alternatively, have **get_sums** return a pair of **double**s). Implicit conversions will take care of the rest. This solves part of the problems. In the calculation of **sd** the parentheses around n – 1 are missing. Also, because we include **<cmath>**, we should use **std::sqrt** instead of **sqrt**.

With all these changes, **calc** now becomes

```
template <typename Container>
auto calc(const Container& container)
{
  auto pruned = remove_outliers(container);
  auto [sum, sumsq] = get_sums(pruned);
  double n = pruned.size();
  double mean = sum / n;
  double sd = std::sqrt((sumsq -
    sum * sum / n) / (n - 1));
  return std::make_pair(mean, sd);
}
```

That covers the technical aspects of the header. We can also make some improvements in the implementation file `statistics.cpp`. The function read hides its use of **std::cin** in its internals and fills the vector it gets as argument with values. An interface that better states **read**'s purpose is

```
std::vector<int> read(std::istream& is)
```

It doesn't hide any input stream in its internals and it is clear that it produces a vector of **int**s. Changing **read** and **main** is left to the reader.

In **main**, again, we can use a structured binding **[mean, sd]** instead of **result**, improving readability.

Finally, from a functional point we can question the validity of **remove_outliers**. If we have measurements consisting of 10 times the value 1, 10 times the value 6 and only 4 values in the range 2 to 5, can we really say that all values 1 and 6 are outliers? Or if we have measurements with 100 values in the range 10 to 50, a single value 1 and a single value 2, why is 1 an outlier and 2 not? To answer those questions we need to know more about the data and how it was measured.

Also the code is not robust. **calc** will go wrong when we have no value or only one value after removing outliers, and **read** will stop processing input when it sees anything that is not an integer or whitespace.

Balog Pál <pasa@lib.hu>

First thing I checked the claim. ☺ Excel indeed provided STDEV 2.16 for the numbers 2...8 and even provided the formula.

Reading the code, I see plenty of problems with efficiency, style and error handling, but for the input in the example nothing that's a show-stopper. That leaves the **result** calculation as the main suspect. The formula I get starts with **n*** but that can be rearranged by simplifying with **/n**. After that it looks almost fine, except we need to divide by **n-1**. And here we divide by **n** then do a **-1**. That part should look like **/ (n-1)**. Let's recalculate between the two: 1.73205, square, +1, *n (that is 7), /6, sqrt: 2.1602 that matches the excel figure and what OP expected.

With this small change the code 'works' for the test input, so we're ~~done~~ looking at the other things mentioned in the intro.

remove_outliers starts fetching **min** and **max** with 2 calls to an **std::algorithm**. That is much better than the still usual approach of writing some loop and doing it manually. However, it could be improved by doing it in a single step with **minmax_element**. Could even use the parallel version.

Then it runs ahead erasing what we found, before making sure we did find something. I'd bet an overlook and not OP's careful consideration that it will actually work with empty vector never calling into the lambda. I base the bet on that I have no good idea what is really expected from this function to do in the first place. Currently it does remove all instances of **min** and **max**, so a data set with seven 1s and four 2s will be emptied. Maybe it intended just remove one instance of those numbers. Maybe it should not even do anything if we have just 2 or less data points. On my review this would not pass without having proper comments on what the intention is.

get_sums is poorly named, but at least it is evident that it calculates the sum of all elements and their squares. It takes the input by copy for no good reason. While the for loop is pretty straightforward, if we used an algorithm earlier why not do that again, **accumulate** should be up to this task. And then we could also run it parallelized without effort.

One problem to look for here is overflow. We add up numbers in the vector in the same type. Even the simple sum is suspect to not fit, let alone the sum of squares. The caller of the function uses **double** for calculation regardless what is in the vector, having **sum** and **sumsq** be **double** instead of **element_type**, or provided by the caller would be more sensible.

Both in this for and the previous lambda OP used **auto**. I prefer **auto const&** as a baseline, especially when we don't really know what the type is. In this particular case it is likely some numeric type and we only use the value, still just an extra point of fragility.

calc also takes the input by value, though it at least has a reason: it needs a copy that will be stripped of the outliers. Here we run into disaster if the input was empty or became such after stripping as we divide by **n**.

mean looks fine as written and provides the desired value in the 'test' too, but probably not what we meant to do really: its type is **double** but the calculation will be done with types **int** and **size_t**, losing any fraction. And losing the sign too. I decided to go paper-only with this, those with a compiler at hand may try some runs with a few negative numbers. That would sort-of go away if we used **double** in sums as was suggested earlier. I say sort-of because the user of **double**s should be aware of the precision limit and other quirks attached to floating point types.

For readability **sums.first** and **second** are clearly suboptimal, with structured bindings already in C++ we can better say **auto const [sum, sumsq] =** … .

Also the naked call **sqrt()** is suspect. We included **<cmath>** so it should be **std::sqrt()** really. We might get away with it, but AFAIK it may not find the function unless we include **<math.h>**.

In the implementation file we have **read** that is **void** and has an _Out_ parameter. That's very last century, in most cases it's better to **return** the result and leave the **T&** only for _InOut_ params. That relieves the headache about what to do if the incoming **v** is not empty and the caller can make the result **const**. Also it is not nice to have **std::cin** hard-wired here, I'd make that passed in from above.

In **main** again we could use structured bindings. Also OP should learn the keyword **const** and use it wherever possible. In the current state `statistics.h` has no reason to exist, but if it is, should be self-contained with its includes (uses plenty of **std::**) and multi-inclusion-protected.

That leaves us with one more elephant: the **statistics** namespace. At this state it has no reason whatsoever to exist. With a single name that belongs there (and 2 more sneaked in). And that name is as awful as it gets. If it is meant as a 'component', then design it properly, a clear public interface and all the rest hidden from sight. Are the templates needed at all? If so, all of them? And all templated on the collection, instead of, say, iterators?

Finally, a few points that some might point out but that I don't consider to be issues:

- no **endl** or **flush** of **cout**: that's supposedly mandated to happen on exit
- **auto** function returns and variables: some people prefer to spell out the type for {reasons} that I don't find very strong in a work environment.

Marcel Marré < marcel.marre@gerbil-enterprises.de> and Jan Eisenhauer <mail@jan-ubben.de>

The main issue with CCC 117's code is a misunderstanding of C's and C++'s data type conversion and calculation rules. For both **mean** and **sd**, the code in **calc** employs integer calculations (up to the **sqrt** in the case of **sd**). Thus, **mean** would only ever take integer values, and **sd** would only ever take the square root of an integer (and the result in the example run is the square root of three).

There are multiple ways to deal with this. **get_sums** could return a **pair<double,double>**, explicit **double**-casts could be added to the code or the line **auto n = v.size()** could be changed to **double n = v.size()**. We chose the third option, since it is small and local, makes all denominators **double**s and thus forces **double** divisions.

In addition, the term **n-1** in the calculation of the standard deviation requires brackets since otherwise there is no division by **n-1** but by **n** followed by subtraction of **1**, followed by the calculation of the square root.

The function **remove_outliers** also deserves a close look. First off, a semantic question is whether, as the code does, all values equal to the largest and smallest element should be removed from the container. In the case of 'pathologic' samples in which either the minimum or maximum value appears often, the results could be skewed markedly. If only one minimum and maximum should be removed, we do not need **remove_if** to delete the items:

```
auto min = min_element(v.begin(), v.end());
v.erase(min);
auto max = max_element(v.begin(), v.end());
v.erase(max);
```

Note that iterators become potentially invalid when the container is changed, so one should not, in this case, get both iterators and then erase each, since this can lead to a segmentation fault, or incorrect results.

If the implemented semantics are correct, the use of iterators within a function changing the container is also problematic. Consider the following two runs, which should lead to the same results, but do not (with the type problems in **calc** already fixed):

```
$ echo 1 1 2 3 4 5 6 8 7 9 9 | statistics
mean: 5, sd: 2.16025

$ echo 1 2 3 1 4 5 9 6 8 7 9 | statistics
mean: 5, sd: 2.73861
```

Rather than storing the iterator and dereferencing them each time in the lambda, it is better to store the value of the minimum and maximum rather than their position in the container. For better readability, considering **v** is already used in the outer function, we also suggest renaming the parameter of the lambda to e.g. **item**.

We would also suggest not **using namespace std** in **remove_outliers**, but rather **using std::min_element**; and **using std::max_element**. This makes accidental use of identifiers from the **std** namespace less likely, while still having the advantage (over fully qualified usage) of allowing iterator-specific **min_element** and **max_element** functions to be found by argument dependent lookup.

Instead of using **min_element** and **max_element** individually, **minmax_element** will only walk the container once to get iterators to both extrema.

Looking beyond **remove_outliers**, the whole code is rather fragile, since it does not protect against empty containers either before or after the run of **remove_outliers**. Even the case of a container with one element (after **remove_outliers**) will lead to a division by zero in **calc** (although on g++ 7.4.0 without specific options this leads to nan rather than an exception).

However, the call **echo | statistics** does lead to a segmentation fault and dumped core. It is a matter of design who should be responsible for relevant checks. As it is, **remove_outliers** and **get_sums** are really implementation details for **calc**, so it might be natural to make the checks in **calc**.

It is generally advisable for a header to include those headers it requires, whereas in the presented code the source file using the header includes all required headers:

> Including the `.h` file as the very first line of the `.c` file ensures that no critical piece of information intrinsic to the physical interface of the component is missing from the `.h` file […].
>
> *Large-scale C++ software design* by John Lakos, ISBN 0-201-63362-0, page 111

get_sums could also be replaced with **std::accumulate** and **std::inner_product**, although in this case going through the container once in **get_sums** rather than once in each of the standard algorithms may yield performance benefits.

If the semantically correct choice for **remove_outliers** is the deletion of only one maximum and minimum each, the code could even be restructured to handle the numbers from **std::cin** incrementally without storing them in a vector at all, thus allowing huge datasets to be handled with a very moderate (and constant) memory footprint. A detailed description how this could be done is beyond this code critique, especially as it isn't clear what **remove_outliers** should really do.

Commentary

I think we had great coverage from our critiques this time so there's really very little left to say.

It would, in practice, be important to know more about the use to which this program would be put: what sort of size are the intended data sets and how many times is the program required to be executed. The answers to these questions would help to decide how much emphasis to place on raw performance and on memory reduction.

In my experience, C++ programmers can over-emphasise performance over clarity; although sometimes, of course, you can get both – as several critiques pointed out by suggesting **minmax_element**.

It was good that the original code had actually been tried out on a known data set (and an error detected), although no-one explicitly suggested ways to improve the test set.

The oddity in the program is **remove_outliers** – this is a good case where I think in a code review you'd ask for a comment explaining the purpose of the function and, hopefully, providing a link to some broader explanation of the algorithm; without this it is hard to know whether the code is doing the job it is supposed to.

The winner of CC 117

All the entrants identified the two main problems with the code presented. It can be hard to spot this sort of problem in code since the C++ code looks very like the mathematical formula being used and this familiarity can blind us to the errors.

Marcel and Jan also pointed out another, more subtle, error with using the dereferenced values ***min** and ***max** in **remove_outliers**.

Various critiques suggested additional standard algorithms that could be used and also pointed out the sub-optimal use of value and reference semantics in the function calls.

Pal independently verified the claim about the anticipated value from the test dataset, which was a good idea.

It was hard to choose between the critiques, but on reflection I decided to award the prize for this issue to Hans' critique. Thank you to all who entered!

Code Critique 118

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

I'm writing code to process sets of address lists, and was hoping to avoid having to make copies of the data structures by using references. However, I can't get it working – I've produced a stripped down example that reads sample data from a file but it doesn't work at all.

Sample data:

```
Roger Peckham London UK
John Beaconsfield Bucks UK
Michael Chicago Illinois USA
```

Expected output:

addresses, sorted by country and then county

Actual output:

three blank lines!

Listing 3

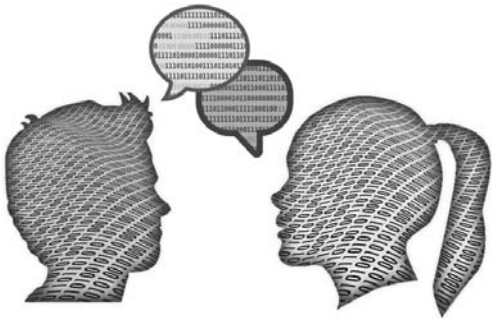
```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <map>
#include <string>
#include <vector>
struct address /* simplified */
{
  std::string city;
  std::string county;
  std::string country;
};
std::istream& operator>>(std::istream& is,
  address& t)
{
  is >> t.city >> t.county >> t.country;
  return is;
}
std::ostream& operator<<(std::ostream& os,
  address& t)
{
  os << t.city << ' '<< t.county << ' '
     << t.country;
  return os;
}
```

Listing 3 (cont'd)

```
// sort addresses by country, county, and
// then city
bool operator<(address const &a,
               address const & b)
{
  if (a.country < b.country) return true;
  if (a.country == b.country &&
      a.county < b.county) return true;
  if (a.country == b.country &&
      a.county == b.county &&
      a.city < b.city) return true;
  return false;
}

// This is just for testing, real data
// is supplied differently
auto read(std::istream &is)
{
  std::map<std::string, address> ret;
  while (is)
  {
    std::string name;
    address addr;
    if (is >> name >> addr)
    {
      ret[name] = addr;
    }
  }
  return ret;
}

int main()
{
  std::map<std::string, address> map;
  map = read(std::cin);

  // Don't copy the addresses
  std::vector<std::tuple<address&>> addrs;
  for (auto entry : map)
  {
    addrs.push_back(entry.second);
  }

  // sort and print the addresses
  std::sort(addrs.begin(), addrs.end());
  for (auto& v : addrs)
  {
    std::cout << std::get<0>(v) << '\n';
  }
  std::cout << '\n';
}
```

Listing 3 contains the code.Can you solve the problem – and then give some further advice?

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



Beyond Code Criticism

Daniel James considers issues that might lie behind code like that in Code Critique 117.

The Code Critique Competition (CCC) is a regular feature of CVu, and has been running for years. In each issue of *CVu* there's a new snippet of code that doesn't work quite as it should – something a beginner might write – and readers are asked to correct the code, and to comment on any other way in which it might be improved. The focus is on the code, the bugs it contains, and the overall quality of coding in the snippet. We're not asked to look beyond the code provided.

CCC 117, whose solutions appear above, caught my eye in particular for the function named `remove_outliers`. This function is documented with the single comment – the only comment in the whole code sample – "get rid of the biggest and the smallest", with no further explanation.

That rang a mental alarm bell for me straight away because I know that the term 'outlier' is a specific term in the field of statistics. It doesn't just refer to the data at the end of the range – 'the biggest and the smallest', mentioned in the comment – but rather to any datum that is so different from the rest of the data set as to be incongruous, and so possibly erroneous. I knew this from school maths, more years ago than I care to admit, and I had a vague recollection that the term applied to data points that differed from the mean by more than three standard deviations. I see now from Wikipedia [1] that the picture is rather more complicated, but it is clear, at least, that a given data set may or may not contain values that can be considered to be outliers and if it does they will not be simply the highest and the lowest values in the set.

I also recall from my time as a research scientist that not every surprising measurement is an error. A data value that lies outside the expected range cannot just be ignored – the value should be checked, the experiment repeated, the method of measurement refined, until we can be sure that the value is correct. If it still doesn't fit the theory perhaps the theory is wrong? That is how scientific advances occur – not by ignoring inconvenient data but by revising theory to fit them.

As guest editor I've already had the chance to have a look at the solutions to CCC 177 sent in by readers. All have taken the sample code and its comment at face value, and corrected the code to do what the comment said – which is probably the right response in the context of a competition about code quality. However, as Roger Orr writes in his summary: "… this is a good case where I think in a code review you'd ask for a comment explaining the purpose of the function and, hopefully, providing a link to some broader explanation of the algorithm...".

Luckily, the code in CCC 117 was made up as a challenge for programmers, it's not supposed to solve any real-world problems, and it doesn't actually matter if it discards perfectly reasonable data. It will never have been through a code review as it's not supposed to be production code – it's sole purpose is to be reviewed by the readers of *CVu* for the CCC – and that review really is only about code quality.

What, though, if the code snippet were real code that was supposed to do something meaningful? In that case we'd hope that the programmer would already have asked a lot of searching questions and discovered what was actually required, and would have written code that explained itself better and did the right thing. That's easier said than done, though – the programmer often doesn't have access to the people who understand the requirement, and can only ask questions of managers who don't themselves fully understand the problem. The answer, all too often, is "Don't overthink it. Just write the code." This is a serious failure in communication, and a sure-fire way to achieve code that doesn't do what's wanted!

Then again, what if the snippet were from a legacy codebase that has been found to produce incorrect outputs? There might be nobody around to ask what it should be doing. We'd then want to look at the specification, and maybe the original requirement. We'd want to know a lot more about the problem that the code was supposed to solve. Without that knowledge we couldn't begin to say what might possibly be wrong with the code. With luck, everything would be clear from the documentation – if it existed, and if we could get access to it.

It could be something innocuous – perhaps the programmer has used the term 'outlier' without knowing its significance in statistics, and there is a perfectly good reason to discard the highest and lowest values. A lot has been written about self-documenting code, and how the choice of good names for the parts of a program can make the code easier to read. This is the opposite: a situation in which an poorly chosen name – chosen poorly because the programmer was unaware of the specific meaning of 'outlier' – leads to confusing code. Communication is hard – especially when you don't know the jargon being used. If this is the case, here, the error is simply one of naming – `remove_extremes` would be a better name – and we can turn our attention back to improving the implementation. It would still be worthwhile to add a comment about why the extreme values are to be discarded, with a reference to the specification and/or the requirement.

It could be that `remove_outliers` was intended to remove actual outliers in one of the statistical senses, but that what we see is just a placeholder for actual code that should have replaced it later in development. We've all mocked out a function during development, but how many of us have forgotten to remove the mock from production code? It shouldn't be possible as it should get caught in testing … but this code clearly wasn't written for testing! If the code was intended as a placeholder, the programmer should certainly have communicated that intention by commenting it as such.

The explanation may, of course, be that `remove_outliers` just doesn't do the right thing, and maybe doesn't do anything sensible at all. The requirement wasn't communicated effectively to the original programmer, or the programmer didn't correctly translate that requirement into code – we can't tell which it is without going back to the requirement. To fix the code we need to fix the failure in communication.

The problem is ultimately a communication problem. We have a function name that suggests a meaning that may not have been intended and a comment that says what the code does rather than why, combined with code that is suspect. We're left scratching our heads and wondering what it all means. Of course, in the CCC that's the whole point – to make us think – but production code should tell the whole story and leave no room for guesswork.

Reference

[1] 'Outlier' – https://en.wikipedia.org/wiki/Outlier

DANIEL JAMES

Daniel James left postgraduate chemical research to work in software development. He has worked on small and large systems, on small and smaller computers, and has come to specialize in IT security. His first interest is in making the computer do something useful. He can be contacted at accu@sonadata.co.uk

The Standards Report

Guy Davidson introduces himself as the new Standards Officer and makes his first report.

I was elected standards officer at the last ACCU AGM on April 13th. This is my first report on the Standard, so I'm going to start by briefly describing the workings of the committee, how I serve, and how I will be writing my reports for this and future editions of *CVu*.

C++ was first standardised in 1998: however, that wasn't the end of the story. Since then Working Group 21 (C++) of Sub Committee 22 (Programming Languages) of Joint Technical Committee 1 (IT) of the International Organisation for Standards (ISO) has been meeting regularly to respond to proposals for improving the language. The first revision of the standard was published 13 years later in 2011, and since then a new revision has been published every three years.

The committee consists entirely of volunteers. Some are sent by their employers while others are private individuals, but all of them are interested in assisting the development of the language. At each committee meeting, a subset of these volunteers meet for six days, Monday to Saturday, to consider proposals. Because of the sheer number of proposals, the committee is divided into four working groups and 21 study groups to provide more focus. At the last meeting in Kona, Hawai'i (yes, it's a hard life) there were 177 proposals offered for consideration.

The four working groups divide into two parts, one for the language and one for the library. Each part has two components, one where proposals are considered, and one where the actual words of the proposals to be added to the standard are considered: the standard is a legal document, which needs to be unambiguous. The language groups are called Evolution Working Group and Core Working Group. The library groups are called Library Evolution Working Group and Library Working Group.

The study groups, numbering 20 now, are even more focused, covering areas such as concurrency, reflection, HMI and several others. I have been attending committee meetings since Summer 2017, when it was convened in Toronto. My particular interests are the HMI study group, SG13, which focuses on bringing 2D graphics to the standard library as well as audio capture, audio playback, controller input and haptic feedback. I also participate in SG14, which focuses on low-latency environments such as game development, high frequency trading and other similar areas of endeavour.

A proposal typically starts life in one of the study groups, is refined in one of the evolution groups, and is finalised in the appropriate wording group. Such a proposal is described as 'in flight' when it is between launch and finalising. Once it has been finalised, it is offered to the committee for voting on inclusion in the standard at the closing plenary session on the morning of the last day of work. Although most proposals gain unanimous support, since they have been refined in several stages and given considerable feedback, this is not always the case. Voting can be surprisingly exciting.

To complete this throat-clearing preamble, observe that the committee meets three times a year, while *CVu* is published six times a year. I shall attempt to cover language and library updates in alternating issues, although for this issue I shall cover a summary of the Hawai'i committee. Motions to add proposals to the standard come from two groups: the Core Working Group (CWG) and the Library Working Group (LWG). There were 34 motions in total, so I shall cover a few highlights. Let's start with some significant CWG motions.

GUY DAVIDSON

Guy Davidson is the Principal Coding Manager at Creative Assembly. He has been writing games for about 75% of his life now. He is climbing through his piano grades and teaching Tai Chi while parenting in Hove and engaging in local political activism.

The headline additions to the standard were the Coroutines Technical Specification (TS) and the Modules proposal. These features have not been without controversy. The coroutines proposal introduces a general control structure which allows flow control to be cooperatively passed between two different routines without returning. Rather than the pre-empting of thread switches, keywords highlight where control may be passed, enabling asynchronous programming. The naming of these keywords has been discussed at considerable length, and the committee has settled on **`co_await`**, **`co_yield`** and **`co_return`**.

Modules offer a whole new mechanism of abstraction. At the moment C++ implements modularity using **`#include`**, a pre-processor mechanism without any knowledge of the language. New keywords have now been added, module and import, while export has been resurrected. In addition to internal and external linkage, identifiers can now have module linkage. Name and argument dependent lookup are changed. This results in an easier way to package C++ objects, with obvious implications for package management tools such as conan and vcpkg.

In addition, the Reflection TS reached publication stage. I will be part of the review committee which will approve the correctness of the TS, which introduces the new keyword **`reflexpr`**. Introducing reflection to C++ allows a program to examine its own structure and behaviour at runtime. This is useful for things like serialisation, where you might want to generically serialise classes without having to specify each member datum.

Moving on to the LWG motions, polymorphic allocators were shown a little love with a proposal for an explicit specialisation of **`pmr::polymorphic_allocator`** for use as a vocabulary type. This allows for the reduction of the amount of template declaration in classes which require allocators.

My award for most amusing proposal name went to 'I Stream, You Stream, We All Stream For **`istream_iterator`**' which addressed some of the problems with the design and specification of **`std::istream_iterator`**. The proposal does some polishing of the standard which it is to be hoped you won't notice, and I would have omitted this proposal entirely were it not for the efforts of the author (Casey Carter) in introducing a little levity to proceedings.

The shortest proposal of the session went to 'Making **`std::underlying_type`** SFINAE friendly', which weighed in at 136 words. Again, you should not notice this change.

Five proposals were passed which changed how requirements are specified in the library. All of them begin with a preamble and this text:

The changes in this series of papers fall into four broad categories.

- Change 'participate in overload resolution' wording into 'Constraints' elements
- Change 'Requires' elements into either 'Mandates' or 'Expects', depending (mostly) on whether or not they can be checked at compile time.
- Drive-by fixes (hopefully very few)

These papers are a response to the arrival of concepts and contracts, and how this should be incorporated into the standard library.

Finally for this article, there was a proposal to introduce **`ssize()`**, a signed version of **`size()`**. This contributes towards cleaning up the use of unsigned in the standard library which is usually not the right choice.

In my next report, I will be covering events from the Cologne meeting, which takes place in July.

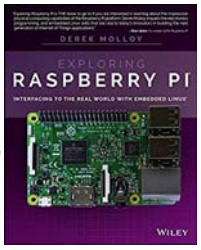
Reviews

The latest roundup of reviews.

We are always happy to receive reviews of anything that may be of use, or of interest, to software developers. The emphasis will tend to be on technical books, but we are also interested in less-technical works – for example, the humanities or fiction – and in media other than traditional print books.

Do you know of something you think deserves – or does not deserve – a wider audience? Share it here. We're keen to highlight differing opinions on the same work, so if you disagree with a review here do send in your own! Contact Ian Bruntlett at reviews@accu.org.

Exploring Raspberry Pi: Interfacing to the Real World with Embedded Linux



By Derek Molloy, published by Wiley (2106), ISBN: 978-1-119-18868-1

Reviewed by Daniel James

This book – with a glowing endorsement by Eben Upton on the front cover – promises a lot. Can it deliver?

When we hear 'Embedded Linux' we normally think of some sort of appliance – a router, maybe – that runs some custom application on a trimmed-down Linux distro from ROM. The Raspberry Pi is most emphatically not an embedded system, in that sense, but rather a fully-featured computer running (typically, but not necessarily) a full desktop Linux distro. The book's sub-title therefore led me, initially, to fear the worst.

The book's 700-odd pages are divided between sixteen chapters organized in three sections. Each chapter is well-organized, in most cases beginning with a preamble and a list of any hardware parts required to complete any exercises in the chapter, and ending with a summary and in some cases a 'Further Reading' section listing both online and hard-copy resources. Photographs, diagrams, and code samples are plentiful, and are used effectively.

The first section is entitled 'Raspberry Pi Basics'. The first couple of chapters describe the basic hardware and software environment provided by a Raspberry Pi running the default Raspbian (Debian-derived) Linux distribution. The availability of other Linux and non-Linux operating software is mentioned in passing, but the emphasis is (quite rightly, in my view) on Raspbian. OS installation and set-up are discussed, as is headless use (over a network or a serial connection). Chapters 2 and 3 contain as good a short introduction to using the Linux command line as I can remember seeing, together with a quick introduction to git (which is important as many Raspberry Pi code samples and Open Source libraries are available from github).

Chapter 4 contains a good tutorial on the theory of digital electronics and a description of the components one may encounter and how to use them. Chapter 5 ends the first section of the book with an introduction to programming on the Pi. The efficiency of various languages is compared, and examples of code to drive a simple flashing LED circuit are given in a variety of languages from bash to C++, passing through the inevitable Python as well as Java, Javascript, and Lua. None of the languages is covered in great detail, but there is enough to give a flavour, and although the author confusingly refers to 'C/C++' as though that were a single language he does describe them separately and give both a C example and a separate Object-Oriented example using C++ classes.

The second section of the book is entitled 'Interfacing, Controlling, and Communicating'. Chapter 8 describes the Pi's GPIO bus and gives examples of input and output using the `/sys` filesystem, using memory-mapped i/o via `/dev/mem`, and using the WiringPi C library. The examples here all use the Linux command line or programs in C or C++.

Chapter 7 describes development of Pi software on another (Linux) computer. This leads to a discussion of cross-compilation and remote debugging using Eclipse, and finishes with cross-compiling a custom Linux kernel for the Pi.

The next three chapters deal with communication between the Raspberry Pi and other devices – such as sensors, motors, and displays – using the Pi's GPIO pins as serial, I2C, and SPI buses. The techniques are all well explained and example code (most of it in 'C++ as a better C') is given. The last chapter of the second section discusses using an Arduino computer as a slave to the Pi, communicating over I2C – because there are some things a dedicated microcontroller can do better than a general-purpose computer running a multi-tasking OS!

The final section of the book is entitled 'Advanced Interfacing and Interaction'. Chapter 12 describes web client and server software on the Pi and using the Pi as an IoT device. Chapter 13 adds handling of wireless interfaces and interfacing to XBee devices, as well as Bluetooth and NFC communications. Chapter 14 discusses programming a GUI on the Pi itself, using Qt, and chapter 15 adds images, sound, and video to this and discusses the Raspberry Pi camera interface.

Finally Chapter 16 discusses Linux kernel programming and Pi-specific device drivers written as loadable kernel modules.

That's a lot of material! If the coverage of some of it is a little light that's understandable. Some authors might have not have chosen to include some of the more advanced and less Raspberry-Pi-specific topics, but I would say that one of the strengths of this book is the breadth of its coverage. The reader may need to look further to learn how to use some of the techniques presented to best effect, but the coverage does at least illustrate what can be done and suggest avenues for further reading. It may not contain all the answers, but it does at least help in concocting good internet search terms!

Two things niggled. One is to do with software, and concerns the author's tendency to blur the distinction between C and C++. One might also hope that a book published in 2016 and claiming to use C++11 would use more modern features and idioms.

The other niggle concerns hardware. Everything else I've read that describes controlling an LED with a Raspberry Pi uses a simple circuit in which a current-limiting resistor and an LED are connected in series between one of the Pi's GPIO pins and GND, but here it is stated that the Pi's GPIO lines can provide only 2-3mA of current – not enough to drive an LED – and will be damaged by any attempt to draw more. The author suggests using a transistor to switch a circuit drawing current from another source instead – and, indeed, all the examples in the book that use LEDs driven by the Pi employ such a technique. That surprised me – were all

those other books, articles, and websites wrong? Was I in danger of blowing up my Pi?

No. The book is misleading, here, because it perpetuates an early misconception that seems to arise from the fact that the original Raspberry Pi had only about 50mA of power available in total for its 17 GPIO pins, and so just under 3mA per pin. Later Raspberry Pi models – including those current when the book was published – have more power available, and the actual limit per GPIO pin is 16mA (a limitation of the Broadcom BCM2835 system-on-a-chip family, versions of which are used by all models of Raspberry Pi). 16mA is plenty to drive typical LED with (say) a 260 Ohm current limiting resistor in series. I've had 26 LEDs running simultaneously using all 26 GPIO pins of a Raspberry Pi Zero-W in this way.

There is some discussion of this on the book's support website. It seems that the author was unable to find authoritative data on the BCM2835 when writing the book, and based his advice on the total current limit for the original Pi. This doesn't make the book wrong; it just means that some of the example circuits in the book could have been made simpler. The use of a transistor to switch a circuit is a valid technique, and is necessary for circuits whose current requirements really do exceed the capability of the GPIO bus.

I found this an interesting and surprisingly exhaustive book that does, indeed, live up to its hype. As a programmer I found it to be a good guide to the Raspberry Pi, its capabilities, and all the things you can do with it – as well as a useful course in digital electronics. The old C++ style in the examples was disappointing, but this book is more about hardware than software, so perhaps I'm being unfairly critical in this regard.

For experienced programmers who are looking specifically for information on Raspberry Pi hardware and interfacing, and who will not be led astray by any deficiencies in the sample code, I'd give this a 'Highly Recommended' rating, for everyone else I'd give it 'Recommended'.

Language Implementation Patterns

By Terence Parr, published by Pragmatic Bookshelf (2009), ISBN: 978-1-93435-645-6

Reviewed by Paul Floyd

This was my summer holiday reading. I should perhaps have taken a slightly longer book, but I wanted to keep the weight down.

The author of this book is also the creator of ANTLR (Another Tool for Language Recognition, something that I'd always imagined be some sort of backronym from LR, but no, ANTLR is LL – LR and LL being parser types). Parr clearly knows his stuff, and does a good job of explaining the processes. The explanations are clear and well laid out. The sample code is mostly in Java, but it is simple enough that anyone with a grounding in C or C++ should be able to follow it.

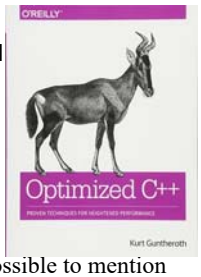
The chapters are mostly split into two parts. First is an overall introduction to the theory being covered. This then leads to 'Patterns', which cover the specifics and example implementations in more details. Often design choices are presented along with their advantages and disadvantages. The book progresses from basic parsing to template-driven generators in a steady fashion. Perhaps this comes from the study of grammars, but the explanations all seemed to progress without there ever being a need to refer to matter covered later in the book.

I only have a fairly small amount of experience with parsers and lexers (mainly with flex and bison), so the parts that were the most interesting to me were the extras that such old-school tools don't offer (that I know of) like data models and rule rewriting.

Finally, I enjoyed this book so much that on return from my holidays I bought *The Definitive ANTLR 4 Reference* by the same author (watch this space, but don't hold your breath).

Recommended.

Optimized C++



By Kurt Guntheroth, published by O'Reilly (2016), ISBN: 978-1-491-92206-4

Reviewed by Paul Floyd

I've always been an advocate of avoiding gratuitous premature pessimization (is it ever possible to mention optimization without quoting or paraphrasing Knuth/Hoare?). So this book was pretty much up my street. It's well written, and I liked the fact that it uses C++11 – there's no point in pandering to the laggards that are stuck in the last century. There are 13 chapters that cover an introduction, hardware and timing, algorithmic optimization, low level optimizations (code changes, memory use, I/O and threads) and memory management.

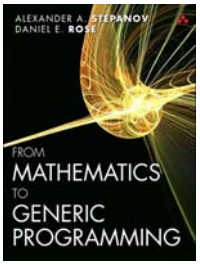
I enjoyed the coverage of timing, and in particular the limitations and evolution of the timing features available on x86 based systems. Another high point was the debunking of some myths like `std::vector` always being significantly faster than `std::list`, along with fairly wide ranging tests. However, this leads me to one of the weaknesses. Pretty much all of the testing was done with one compiler (MS Visual Studio) and platform. So without a wider range of platforms and compilers I found myself taking the results with a pinch of salt.

There were three things that struck me as missing. In Chapter 4, there is a lengthy discourse on optimizing a function that strips control characters from a string. One of the techniques used is pass by reference. This results in an 8% performance degradation, which is not explained in any way. I would expect a book like this to get to the bottom of such as issue. The second thing that is missing is detailed coverage of profiling tools. The author seems to prefer instrumenting code with timer calls, which is good enough, but I thought that there should have been some explanation of the different kinds of profilers and some example usage. Lastly, a lot is said about `std::string` and its performance not being great. However, there is no mention of the Short String Optimization (SSO). I was expecting some mention of it and perhaps characterizing the performance 'knee' is string lengths exceed the SSO threshold.

In summary I enjoyed reading this book and learned a few things but there were a few annoying nits.

Recommended.

From Mathematics To Generic Programming



By Alexander Stepanov and Daniel Rose, published by Addison-Wesley (2014), ISBN: 0-321-94204-3

Reviewed by Paul Floyd

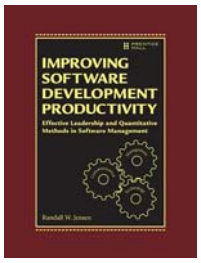
This is a kind of follow-on from *Elements of Programming*, though this book has much more of a cultural feel to it. It's a kind of History of Art for Mathematics, Algorithms and Programming. The early chapters cover mathematics from the Ancient Greeks to the Middle Ages. As well as descriptions and examples of the mathematical concepts being introduced, there are short profiles of the men and women that discovered them. Some of these are fascinating – I didn't realise that Lagrange was born in Italy, I knew little of Peano and I'd never heard of Simon Stevin.

The maths is mixed with examples in C++, more numerical earlier in the book and then more abstract later as the notions of group theory are introduced. One theme running through the book is the Greatest Common Divisor (GCD) which takes us from Euclid to Chapter 13 which covers cryptology. I found the explanations to be clear and in some cases (like cyclic groups) I understood properly for the first time in my life. This isn't a book to learn any programming skills from, but it does show that some of the ideas that underlie C++ and programming in general go back over thousands of years.

I enjoyed reading this book so much that I'm tempted to search for a more compendious book on the history of mathematics.

Highly recommended.

Improving Software Development Productivity



By Randall W. Jensen, published by: Prentice Hall (2015), ISBN: 978-0-13-356267-5

Reviewed by Paul Floyd

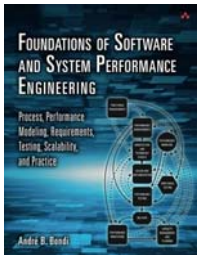
I've read a lot of books, and I usually look in the bibliography to see if there is anything there that looks interesting. Usually when it comes to bibliographic references, they are of the works of Barry Boehm and Capers Jones. I don't remember seeing any references to Randall Jensen, which I thought was perhaps a bad sign. Well, this initial slight prejudice turned out to be unfounded.

It seems that to be an authority in software productivity you either need to have developed a model or have some sort of methodology. In the case of Boehm, it is the Cocomo model, in the case of Jones it is the function points methodology. In Randall's case, it's two models, Sage and Seer, which seem reasonably similar to Cocomo. Whilst the bulk of the book does go into the application of these models, there is also plenty of discussion of the rationale behind the factors that are employed. For me the high-points of the book are in the earlier chapters where Jensen covers the fundamental problems in software development, some of the history of techniques and that have succeeded and failed. The section on measurement dysfunction definitely struck a chord with me.

On the negative side, there were a few oddities. Jensen clearly has a bit of a bee in his bonnet about the Western education system that favours individual competitiveness rather then teaching group work which would be more suited to most people's working lives. He also says a fair bit about the working environment, eschewing cubicles and proselytising open-space. I've always considered DeMarco and Lister's *Peopleware* to be the reference on this matter, and they advocate private offices.

Recommended.

Foundations of Software and System Performance Engineering



By Andre Bondi, published by: Addison-Wesley (2015), ISBN: 0-3183382-1

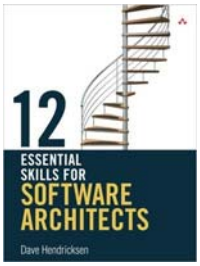
Maybe the problem that I had with this book is that I was expecting something else, something on the same lines as *Systems Performance* by Brendan Gregg (which has much detail on actual measurement of performance). Bondi covers a great deal of the theory around performance engineering. There's a whole chapter that is devoted to queuing theory. Then there is much on planning and requirements and how to interpret and present measurements.

I did quite enjoy the chapter on 'Scalability and Performance' (which has an amusing example of the cloakrooms at the New York Met and Modern Art museums and the Louvre). The following chapter on pitfalls in measurement contained some sound practical advice.

The parts of the book that are 'down to the metal' are a brief mention of 'ps' on Unix for memory measurement and another of 'perfmon' on Windows for network measurement. That's too abstract for me – I guess that you need to be working on a large system with hundreds or more staff and a team dedicated the system performance in order to be able to benefit from a book like this.

Not recommended.

12 Essential Skills for Software Architects



By Dave Hendrickson, published by: Addison-Wesley (2011), ISBN: 0-321-71729-5

Reviewed by Paul Floyd

The 'essential skills' that are covered in this book are soft skills: interpersonal, finances and politics. As such the material is fairly subjective, which makes it difficult to judge whether the advice is good or bad. It also means that the advice can be quite vague and general. Clearly, however, the perspective is that of someone in tune with upper management. Whilst there are sections on saying 'no' and handling conflict, the overall tone is more 'alignment of vision'.

Based on my experience, I see that there is a lot of truth in what Hendricksen says. Early on he describes the 'technical ceiling', basically a barrier beyond which it is difficult for careers to progress without developing these soft skills. I also see that it's a difficult act to balance. When it comes 'relationships over correctness', it's a fine line to tread between being sycophantic, getting the balance right and getting into conflict.

I would have liked fewer buzzwords, and I thought that the diagrams looked a bit amateurish – perhaps the services of a graphic artist could have been called upon?

So in summary, a decent overview of business relationships within software development.

12 More Essential Skills for Software Architects

By Dave Hendrickson, published by Addison-Wesley (2015), ISBN: -321-90947-X

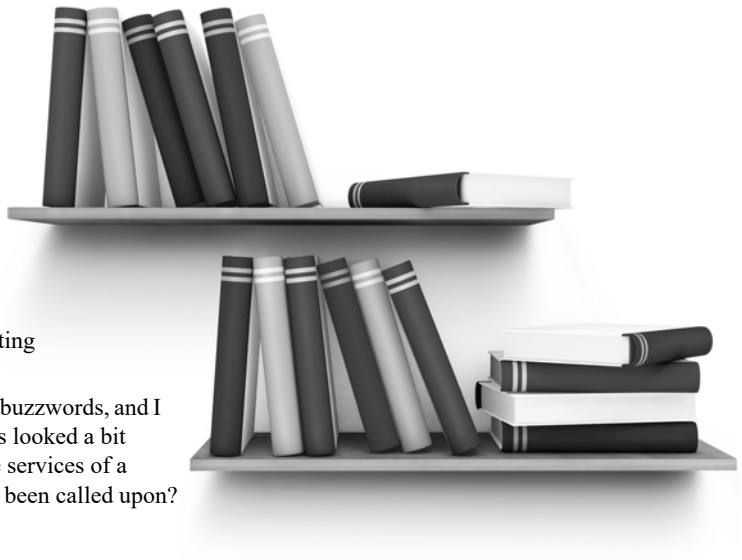
Reviewed by Paul Floyd

In this follow-on book to *12 Essential Skills for Software Architects*, we get coverage of the technical skills that mirror the soft skills. There is some overlap with the first book, and some of the chapters here seemed fairly 'soft' to me: partnership, governance, roadmapping and entrepreneurial execution. It wasn't particularly clear to me how 'software architecture' differs from general software management and development, not that the title matters much.

For a book covering technical skills, I did expect some more concrete examples, perhaps some real world case studies would have helped. As it was, I was left wondering whether I was just reading sensible sounding platitudes, or whether there were any success/disaster stories when the advice was followed/ignored.

One chapter seemed a little odd to me, chapter 6 on 'Platforms'. This more or less assumes that you work on a big system that will evolve to become platforms. Perhaps I haven't worked on enough (or indeed any) many major governmental IT projects, but my experience is that after a couple of decades, code tends to evolve into multi-million line monoliths.

As with the previous book, the diagrams look a bit cheesy and don't add much. For instance, is there really any reason why 'Relevance', 'Excellence' and 'Currency' should form a triangle?



View from the Chair

Bob Schmidt
chair@accu.org

ACCU 2019

ACCU's 2019 conference was held in April, and was a great success. Russel Winder, assisted by Felix Petriconi and the conference committee (Gail Ollis, Roger Orr, CB Bailey, Anastasia Kazakova, Jon Kalb, Francis Glassborow, Timur Doumler, and Guy Davidson), created a conference program with both depth (in C++, of course) and breadth (Rust, Python, Kotlin, Java, Git, Nim, and more). We had keynotes from M Angela Sasse, Herb Sutter, Paul Grenyer, and Kate Gregory; lightning talks; a welcome reception; and a tasty and entertaining conference dinner featuring Echoborg and Code Club.

We had approximately 435 attendees over the four days of the conference. Attendance was down slightly from last year's record attendance of approximately 450 attendees; however, it was higher than expected given the uncertainties generated by Brexit.

Thanks go out to Julie Archer and the Archer-Yates team for their excellent conference organization and on-site management:

Charlotte Tanswel	Laura Nason
Anna Munday	Helen Wormall
Marsha Goodwin	

Thanks also to the conference sponsors for their support:

Bloomberg	Undo
QBS	JFrog
Mosaic Financial Markets	Riverblade
Graphcore	Jump Trading
#include <c++>	

ACCU Autumn 2019

ACCU is putting on a two-day conference in Belfast, Northern Ireland on the 11th and 12th of November 2019, in conjunction with and immediately following the November WG21 (C++) standards meeting. The conference will be held at the same Hilton Hotel being used for the WG21 meeting. Registration information will be posted to the ACCU website [1], so check there for more information when it becomes available.

2019 AGM

ACCU's Annual General Meeting was held on April 13, 2019. Results of the election were announced – there were no surprises, given that everyone was running unopposed.

- Secretary – Patrick Martin
- Local Groups Coordinator – Phil Nash
- Membership Secretary – Matt Jones
- Standards – Guy Davidson
- Publications – Roger Orr
- At-Large – Ralph McArdell

There were two motions on the ballot, both of which passed. The first gave authority to the committee to enter into insurance arrangements; the second allowed for the nomination and election of officers at the AGM (because there were no nominations for Chair or Treasurer).

I asked the members attending the AGM if anyone was willing to stand as Treasurer; there were no volunteers. Rob Pauer had indicated that he would be willing to stand again in the absence of a new volunteer, and pursuant to the authority granted by the approval of the motion to allow election of officers at the AGM, Rob was nominated and re-elected Treasurer by unanimous vote.

I also asked if anyone was willing to stand as Chair; again, there were no takers. I volunteered to serve for an additional year, and was nominated and re-elected. Thank you.

Finally, we needed to find two new auditors. Guy Davidson has taken a position on the committee, so we needed someone to finish his final year, and Niall Douglas concluded his two-year term. Alan Griffiths volunteered to finish Guy's term, and Dietmar Kühl volunteered for the two year term.

(The AGM Pack [2] and draft minutes of the 31st AGM are available online [3]; members must log in to the website.)

Thank you to everyone who came to the meeting; our new committee members and audit team members; and to outgoing committee members Nigel Lester and Emyr Williams; and auditor Niall Douglas.

Reviews

Ian Bruntlett has volunteered to try his hand as Reviews Editor. He has plans to expand this *CVu* feature from just book reviews. As he said in his introductory email, he would like to include "anything that helps a software developer cope with things […] – the traditional technical stuff, favourite old books, podcasts, e-books, the humanities, [and] fiction…"

We have a new(ish) email address for ideas and submissions: reviews@accu.org. Please use this address to send your ideas and reviews to Ian.

Please join me in thanking Ian for volunteering for this role.

Advertising

Seb Rose has notified the committee that he is stepping down from the advertising position after many years of service.

The committee and I thank Seb for his years of service to ACCU.

Call for Volunteers

We have several open positions on the committee. Please consider volunteering for one of the following:

- Publicity
- Study Groups
- Advertising
- Social Media
- Web Editor

Also, we need a new treasurer. Rob Pauer was kind enough to continue in the role when no one else stepped up in April, but he wants to retire, and he should be able to do so. This is one position that requires residence in England, in order to have access to the bank with which we do business.

References

[1] 2009 Autumn Conference: https://conference.accu.org/

[2] 31st AGM Pack: https://accu.org/content/agm/AGM-2019-Pack.pdf

[3] 31st AGM Draft Minutes: https://accu.org/content/agm/AGM-2019-Minutes(Draft).pdf

We need your help!

ACCU is a volunteer organisation. Without volunteers, we cannot function. We need:

- Volunteers for vacant posts on the committee
- People to write articles (regularly or occasionally)
- People who can help out with particular short-term and long-term projects

If you would like to help but are not sure how you can – you may not have a lot of time, or may not be able to commit to anything long-term – please get in touch. You may have just the skills we need for a short-term project or to reduce the workload of another volunteer.

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel code.

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner

To find out more about Intel products please contact us:
020 8733 7101 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE