the magazine of the accu

www.accu.org

{cvu}

Volume 32 • Issue 2 • May 2020 • £4.50

Features

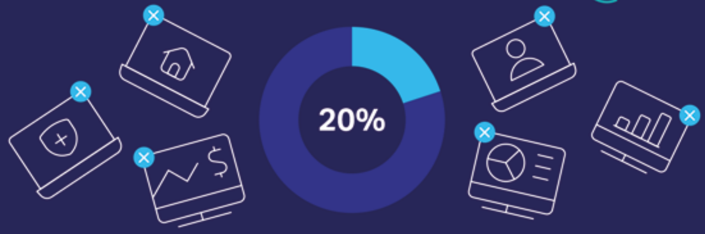
Regulars

Remote access software:

Ensuring productivity by eliminating single points of failure

Unresponsive programs

Knowledge workers on average see productivity drop by as much as **20%** because of unresponsive applications.[1]



Increase productivity and eliminate downtime

A remote access platform maximizes end user productivity with:

- Extremely fast responsiveness resulting from highly efficient data compression protocol.
- Access to Windows®, UNIX® and Linux® applications and desktops.
- Auto-resume for network interruptions.
- Remote collaboration and built-in screen sharing.
- Stable sessions.
- Highly available architecture out of the box.

How it works

Example:
Ensure a highly available, load balanced remote desktop infrastructure for users.

1. Establish load balancing rules based on the number of current sessions and CPU or memory utilization of available hosts.
2. Assign users to the least loaded servers in their host pool with the load balancer.
3. Install connection nodes to offload CPU-intensive tasks and provide faster performance of application servers.
4. Ensure a user's work is never lost with auto-reconnect, which resumes a session when the device is back online.
5. Configure servers in a high availability (HA) cluster to distribute website load and eliminate single points of failure.

The result

Provide users with reliable and fast access to work desktops and server applications from any platform and location, avoiding costly downtime, interruptions and loss of work.

OpenText™ Exceed™ TurboX empowers a global workforce with a high-performance remote access solution that ensures accessibility to graphically demanding applications and desktops through a web browser.

Key partners include:

Agisoft, ASPOSE, ATLASSIAN, axure, BrowserStack, DevExpress, embarcadero, Hex-Rays, intel Software, JetBrains Platinum Reseller, Microsoft Silver Partner, qbs publishing, Sketch, SMARTBEAR, SPARX SYSTEMS, Telerik Develop experiences, think-cell, Visual Paradigm

For your latest software needs, contact our team on:

020 8733 7101 **sales@qbs.co.uk** **www@qbs.co.uk**

QBS PUBLISHING

{cvu}

Volume 32 Issue 2
May 2020
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Silas S. Brown, Pete Goodliffe, Matthew Jones, Roger Orr, Simon Sebright

Reviews
Ian Bruntlett
reviews@accu.org

ACCU Chair
[Vacancy]
chair@accu.org

ACCU Secretary
[Vacancy]
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
Patrick Martin
treasurer@accu.org

Advertising
[Vacancy]
ads@accu.org

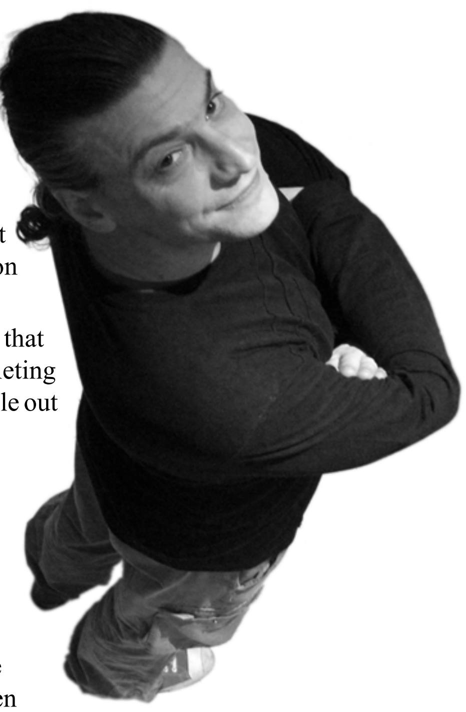
Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

The Story of Code



Over the last several years the tools for revision – or version – control have grown enormously in sophistication and features. It's been a core tool in most software development teams I've encountered in … however many years it has been. I still occasionally encounter places that don't use version control for their software – and even people who seem not to know what it is! – but it seems to me that the free and straightforward availability of facilities like BitBucket and GitHub is causing popularity to grow. Why is version control important? What's it for?

Most developers I speak to have much the same response: that it's a safety net. If you make a mistake, like accidentally deleting all your code just before accidentally pulling the power cable out of your machine, you can revert your changes from the version control system. It's a kind of 'global undo'. An extension of this is that if you discover a problem introduced by a change *in the past*, you can revert to a previous revision.

Another common response is that it provides a well-known 'golden source' of a particular stable state of the code. Coupled with a suite of unit tests, if commits to the version control repo are made only when the tests pass, then each version is a snapshot of a known good state. If you make changes that cause the tests to fail, then your search area is reduced to only those changes since the last revision. Along the same lines is that a software release is pegged to a specific version, so that you can always re-create *exactly* what went into a release.

Both of these are focused on the technical benefits: for the code, or for the released product. My view is that version control is most important for the people that use it. This is particularly true on a shared codebase: if I pull the latest version of the code, I can see the changes made, who made them and why, if the comments are any good. A version control system is a channel of communication. It's so much more than a global undo. It tells the story of the code.

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

SUBMISSION DATES

C Vu 32.3:	1st June 2020
C Vu 32.4:	1st August 2020
Overload 157:	1st July 2020
Overload 158:	1st September 2020

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

Expect the Unexpected (Part 1)

Pete Goodliffe looks into dealing with the inevitable.

We know that the only way to avoid error is to detect it, that the only way to detect it is to be free to enquire.
~ J. Robert Oppenheimer

At some point in life, everyone has this epiphany: *The world doesn't work as you expect it to*. My one-year-old friend Tom learned this when climbing a chair four times his size. He expected to get to the top. The actual result surprised him: He ended up under a pile of furniture.

Is the world broken? Is it wrong? No. The world has plodded happily along its way for the last few million years, and looks set to continue for the foreseeable future. It's *our expectations* that are wrong and need to be adjusted. As they say: *Bad things happen, so deal with it*. We must write code that deals with the Real World and its unexpected ways.

This is particularly difficult because the world *mostly* works as we'd expect it to, constantly lulling us into a false sense of security. The human brain is wired to cope, with built-in fail-safes. If someone bricks up your front door, your brain will process the problem, and you'll stop before walking into an unexpected wall. But programs are not so clever; we have to tell them where the brick walls are and what to do when they hit one.

Don't presume that everything in your program will always run smoothly. The world doesn't always work as you'd expect it to: You *must* handle all possible error conditions in your code. It sounds simple enough, but that statement leads to a world of pain.

From whence it came

To expect the unexpected shows a thoroughly modern intellect.
~ Oscar Wilde

Errors can and will occur. Undesirable results can arise from almost any operation. They are distinct from bugs in a faulty program because you *know* beforehand that an error can occur. For example, the database file you want to open might have been deleted, a disk could fill up at any time and your next save operation might fail, or the web service you're accessing might not currently be available

If you don't write code to handle these error conditions, you will almost certainly end up with a *bug*; your program will not always work as you intend it to. But if the error happens only rarely, it will probably be a very subtle bug!

An error may occur for one of a thousand reasons, but it will fall into one of these three categories:

- User error

  The stupid user manhandled your lovely program. Perhaps they provided the wrong input or attempted an operation that's absolutely absurd. A good program will point out the mistake and help the user to rectify it. It won't insult them or whine in an incomprehensible manner.

- Programmer error

  The user pushed all the right buttons, but the code is broken. This is the consequence of a bug elsewhere, a fault the programmer introduced that the user can do nothing about (except to try and avoid it in the future). This kind of error should (ideally) never occur.

  There's a cycle here: Unhandled errors can cause bugs. And those bugs might result in further error conditions occurring elsewhere in your code. This is why we consider 'defensive programming' an important practice.

- Exceptional circumstances

  The user pushed all the right buttons, and the programmer didn't mess up. Fate's fickle finger intervened, and we ran into something that couldn't be avoided. Perhaps a network connection failed, we ran out of printer ink, or there's no hard disk space left.

We need a well-defined strategy to manage each kind of error in our code. An error may be detected and reported to the user in a pop-up message box, or it may be detected by a middle-tier code layer and signalled to the client code programmatically. The same principles apply in both cases: whether a human chooses how to handle the problem or your code makes a decision – *someone* is responsible for acknowledging and acting on errors.

Errors are raised by subordinate components and communicated upwards, to be dealt with by the caller. They are reported in a number of ways; we'll look at these in the next section. To take control of program execution, we must be able to:

- Raise an error when something goes wrong
- Detect all possible error reports
- Handle them appropriately
- Propagate errors we can't handle

Errors are hard to deal with. The error you encounter is often not related to what you were doing at the time (most fall under the 'exceptional circumstances' category). They are also tedious to deal with – we want to focus on what our program *should* be doing, not on how it may go wrong. However, without good error management, your program will be brittle – built upon sand, not rock. At the first sign of wind or rain, it will collapse.

Take error handling seriously. The stability of your code rests on it.

Error-reporting mechanisms

There are several common strategies for propagating error information to client code. You'll run into code that uses each of them, so you must know how to speak every dialect. Observe how these error-reporting techniques compare, and notice which situations call for each mechanism.

Each mechanism has different implications for the *locality of error*. An error is local in *time* if it is discovered very soon after it is created. An error is local in *space* if it is identified very close to (or even *at*) the site where it actually manifests. Some approaches specifically aim to reduce the locality of error to make it easier to see what's going on (e.g., error codes). Others aim to extend the locality of error so that normal code doesn't get entwined with error-handling logic (e.g., exceptions).

The favoured reporting mechanism is often an architectural decision. The architect might consider it important to define a homogeneous hierarchy of exception classes or a central list of shared reason codes to unify error-handling code.

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

No reporting

The simplest error-reporting mechanism is *don't bother*. This works wonderfully in cases where you want your program to behave in bizarre and unpredictable ways and to crash randomly.

If you encounter an error and don't know what to do about it, blindly ignoring it is *not* a viable option. You probably can't continue the function's work, but returning without fulfilling your function's contract will leave the world in an undefined and inconsistent state.

Never ignore an error condition. If you don't know how to handle the problem, signal a failure back up to the calling code. Don't sweep an error under the rug and hope for the best.

An alternative to ignoring errors is to instantly abort the program upon encountering a problem. It's easier than handling errors throughout the code, but hardly a well-engineered solution!

Return values

The next most simple mechanism is to return a success/failure value from your function. A boolean return value provides a simple yes or no answer. A more advanced approach enumerates all the possible exit statuses and returns a corresponding *reason code*. One value means *success*, the rest represent the many and varied abortive cases. This enumeration may be shared across the whole codebase, in which case your function returns a subset of the available values. You should therefore document what the caller can expect.

While this works well for procedures that don't return data, passing error codes back *with* returned data gets messy. If `int count()` walks down a linked list and returns the number of elements, how can it signify a list structure corruption? There are three approaches:

- Return a compound data type (or *tuple*) containing both the return value and an error code. This is rather clumsy in the popular C-like languages and is seldom seen in them.
- Use an 'optional' data type that can represent 'no value, or a specific value'. This is a syntactically nicer version of a compound data type.
- Pass the error code back through a function parameter. In C++ or .NET, this parameter would be passed by reference. In C, you'd direct the variable access through pointers. This approach is ugly and non-intuitive; there is no syntactic way to distinguish a return value from a parameter.

  Alternatively, reserve a range of return values to signify failure. The `count` example can nominate all negative numbers as error reason codes; they'd be meaningless answers anyway. Negative numbers are a common choice for this. Pointer return values may be given a specific invalid value, which by convention is zero (or `NULL`). In Java and C#, you can return a `null` object reference.

  This technique doesn't always work well. Sometimes it's hard to reserve an error range – all return values are equally meaningful and equally likely. It also has the side effect of reducing the available range of success values; the use of negative values reduces the possible positive values by an order of magnitude. (If you used an `unsigned int` then the number of values available would increase by a power of two, reusing the `signed int`'s sign bit.)

Error status variables

This method attempts to manage the contention between a function's return value and its error status report. Rather than return a reason code, the function sets a shared global error variable. After calling the function, you must then inspect this status variable to find out whether or not it completed successfully.

The shared variable reduces confusion and clutter in the function's signature, and it doesn't restrict the return value's data range at all. However, errors signalled through a separate channel are much easier to miss or wilfully ignore. A shared global variable also has nasty thread safety implications.

The C standard library employs this technique with its `errno` variable. It has very subtle semantics: Before using any standard library facility, you must manually clear `errno`. Nothing ever sets a succeeded value; only failures touch `errno`. This is a common source of bugs, and makes calling each library function tedious. To add insult to injury, not all C standard library functions use `errno`, so it is less than consistent.

This technique is functionally equivalent to using return values, but it has enough disadvantages to make you avoid it. Don't write your own error reports this way, and use existing implementations with the utmost care.

Whistle-stop tour of exception safety

Resilient code must be *exception safe*. It must work correctly (for some definition of *correctly*, which we'll investigate below), no matter what exceptions come its way. This is true regardless of whether or not the code catches any exceptions itself.

Exception-neutral code propagates all exceptions up to the caller; it won't consume or change anything. This is an important concept for generic programs like C++ template code – the template types may generate all sorts of exceptions that template implementors don't understand.

There are several different levels of exception safety. They are described in terms of guarantees to the calling code. These guarantees are:

- **Basic guarantee** If exceptions occur in a function (resulting from an operation you perform or the call of another function), it will not leak resources. The code state will be *consistent* (i.e., it can still be used correctly), but it will not necessarily leave in a known state. For example: A member function should add 10 items to a container, but an exception propagates through it. The container is still usable; maybe no objects were inserted, maybe all 10 were, or perhaps every other object was added.
- **Strong guarantee** This is far more strict than the basic guarantee. If an exception propagates through your code, the program state remains completely unchanged. No object is altered, no global variables changed, nothing. In the example above, nothing was inserted into the container.
- **Nothrow guarantee** The final guarantee is the most restrictive: that an operation can *never* throw an exception. If we are exception neutral, then this implies the function cannot do anything else that might throw an exception.

Which guarantee you provide is entirely your choice. The more restrictive the guarantee, the more widely (re)usable the code is. In order to implement the strong guarantee, you will generally need a number of functions providing the nothrow guarantee.

Most notably, every destructor you write must honour the nothrow guarantee. (That's the case in C++ and Java, at least. C# stupidly called `~X()` a *destructor*, even though it was a finalizer in disguise. Throwing an exception in a C# destructor has different implications.) Otherwise, all exception handling bets are off. In the presence of an exception, object destructors are called automatically as the stack is unwound. Raising an exception while handling an exception is not permissible.

Exceptions

Exceptions are a language facility for managing errors; not all languages support exceptions. Exceptions help to distinguish the normal flow of execution from *exceptional* cases – when a function has failed and cannot honour its contract. When your code encounters a problem that it can't handle, it stops dead and throws up an *exception* – an object representing the error. The language runtime then automatically steps back up the call stack until it finds some exception-handling code. The error lands there, for the program to deal with.

There are two operational models, distinguished by what happens after an exception is handled:

- The termination model

  The *termination model* (provided by C++, .NET and Java), in which execution continues after the handler that caught the exception.

throwing exceptions through sloppy code can lead to memory leaks and problems with resource clean-up

- The resumption model

  The *resumption model*, in which execution resumes where the exception was raised.

The former model is easier to reason about, but it doesn't give ultimate control. It only allows *error handling* (you can execute code when you notice an error), not *fault rectification* (a chance to fix the problem and try again).

An exception cannot be ignored. If it isn't caught and handled, it will propagate to the very top of the call stack and will usually stop the program dead in its tracks. The language runtime automatically cleans up as it unwinds the call stack. This makes exceptions a tidier and safer alternative to hand-crafted error-handling code. However, throwing exceptions through sloppy code can lead to memory leaks and problems with resource clean-up. (For example, you could allocate a block of memory and then exit early as an exception propagates through. The allocated memory would leak. This kind of problem makes writing code in the face of exceptions a complex business.) You must take care to write *exception-safe* code. The sidebar explains what this means in more detail.

The code that handles an exception is distinct from the code that raises it, and it may be arbitrarily far away. Exceptions are usually provided by OO languages, where errors are defined by a hierarchy of exception classes. A handler can elect to catch a quite specific class of error (by accepting a leaf class) or a more general category of error (by accepting a base class). Exceptions are particularly useful for signalling errors in a constructor.

Exceptions don't come for free; the language support incurs a performance penalty. In practice, this isn't significant and only manifests around exception-handling statements – exception handlers reduce the compiler's optimization opportunities. This doesn't mean that exceptions are flawed; their expense is justified compared to the cost of not doing any error handling at all!

Signals

Signals are a more extreme reporting mechanism, largely used for errors sent by the execution environment to the running program. The operating system traps a number of exceptional events, like a *floating point exception* triggered by the maths coprocessor. These well-defined error events are delivered to the application in signals that interrupt the program's normal flow of execution, jumping into a nominated *signal handler* function. Your program could receive a signal at any time, and the code must be able to cope with this. When the signal handler completes, program execution continues at the point it was interrupted.

Signals are the software equivalent of a hardware interrupt. They are a Unix concept, now provided on most platforms (a basic version is part of the ISO C standard [1]). The operating system provides sensible default handlers for each signal, some of which do nothing, others of which abort the program with a neat error message. You can override these with your own handler.

The defined C signal events include program termination, execution suspend/continue requests, and maths errors. Some environments extend the basic list with many more events.

Detecting errors

How you detect an error obviously depends on the mechanism reporting it. In practical terms, this means:

- Return values

  You determine whether a function failed by looking at its return code. This failure test is bound tightly to the act of calling the function; by making the call, you are implicitly checking its success. Whether or not you do anything with that information is up to you.

- Error status variables

  After calling a function, you must inspect the error status variable. If it follows C's **`errno`** model of operation, you don't actually need to test for errors after every single function call. First reset **`errno`**, then call any number of standard library functions back-to-back. Afterwards, inspect **`errno`**. If it contains an error value, then one of those functions failed. Of course, you don't know what fell over, but if you don't care, then this is a streamlined error detection approach.

- Exceptions

  If an exception propagates out of a subordinate function, you can choose to catch and handle it or to ignore it and let the exception flow up a level. You can only make an informed choice when you know what kinds of exceptions might be thrown. You'll only know this if it has been documented (and if you trust the documentation).

  Java's exception implementation places this documentation in the code itself. The programmer has to write an *exception specification* for every method, describing what it can throw; it is a part of the function's signature. Java is the only mainstream language to enforce this approach. You cannot leak an exception that isn't in the list, because the compiler performs static checking to prevent it (C++ also supports exception specifications, but leaves their use optional. It's idiomatic to avoid them – for performance reasons, among others. Unlike Java, they are enforced at run time).

- Signals

  There's only one way to detect a signal: Install a hander for it. There's no obligation. You can also choose not to install any signal handlers at all, and accept the default behaviour.

As various pieces of code converge in a large system, you will probably need to detect errors in more than one way, even within a single function. Whichever detection mechanism you use, the key point is this:

Never ignore *any* errors that might be reported to you. If an error report channel exists, it's there for a reason.

It is good practice to always write error-detection scaffolding – even if an error has no implication for the rest of your code. This makes it clear to a maintenance programmer that you know a function may fail and have consciously chosen to ignore any failures.

When you let an exception propagate through your code, you are not ignoring it – you *can't* ignore an exception. You are allowing it to be handled by a higher level. The philosophy of exception handling is quite different in this respect. It's less clear what the most appropriate way to document this is – should you write a **`try`**/**`catch`** block that simply re**`throw`**s the exception, should you write a comment claiming that the code *is* exception safe, or should you do nothing? I'd favour documenting the exception behaviour.

Next time

So that's an investigation of the landscape of 'error conditions' in our code. We've seen what errors are, what causes them, how we detect and report error situations, and why we care. Errors are not (necessarily) caused by failures of the programmer. But not considering error conditions would be a failure of the programmer.

In the next instalment, we'll consider the best strategies to handle and recover from error situations. We'll see the very practical code implications of good error case handling. ■

Reference

[1] ISO: The C Standard, the original was published in 1999 but has been superseded by the 2018 version of the document: https://www.iso.org/standard/74528.html

What Is Your Name?

Simon Sebright considers the importance of names as identification.

When requirements get to developers, there is inevitably decision making as to how to name your functions, parameters, members, variables, XML elements or attributes, etc. Often these things will head in the direction of what we would normally think of as a schema, i.e. if we were to be designing a database, what would we call the tables and columns.

Furthermore, how these things might appear in the UI in an online form, or on paper in a printed form also need consideration to be correct and as widely-applicable as possible.

With the example of personal names, I aim to show that this is far from obvious and straightforward, and developers as well as requirements people should occasionally take the time to think about it, look for established conventions, standards, etc.

That way, we avoid confusion, which leads to bugs, slower development, rework, etc.

Sadly, many code examples and snippets you will find online exhibit these problems, as they were probably thought up by people focussed on their own limited personal experience of the world.

if you wish to exchange your data with others or to provide some kind of programmatic interface, you are much better off being standard-conforming

So, what is your name?

As with many questions, it depends! If I were to be asked at a party, “Simon” would be my answer. In other settings, more formal, “Simon Sebright” or at school, all the boys were addressed by their surnames, so “Sebright, Sir”.

If I were filling out a form and there was just one field ‘Name:’, I would write ‘Simon Sebright’. In German culture, I would write ‘Sebright, Simon’ by default.

That brings us to the concept of a *Full Name*, i.e., that my name has parts and if you put these together in some way, you get something else, more complete. This is, however, complicated by the fact that I also have a ‘middle name’, so in some circumstances, my full name might include that, for example in identification documents or financial applications, etc.: Simon Xxxxx Sebright.

The parts of a name which make the whole

OK, so we have seen that there are some components to a name. What are they? How should we best name them? When I was growing up in England, although not religious myself, the ‘default’ religion is some form of Christianity, and I was actually christened. That points to the ‘Simon’ being my *Christian Name*, i.e. the *Name* with which I was christened. The ‘Sebright’ bit in this scheme is my *Surname* from the Latin *sur* meaning ‘super” or ‘above’.

If we want to remain religion-neutral, then we need to drop at least the Christian part, so we often see ‘First Name’ and ‘Last Name’ on a form.

Then the idea of *Middle Name(s)* becomes easier, they just slot in between: Simon Xxxxx Sebright. How I write that in the German way, I’m not sure... maybe Sebright, Simon Xxxx.

Also, as an alternative for Christian name, we sometimes see *forename*.

Now we have the problem, that in many countries in the eastern world, often populous, this scheme has issues with the ordering. Consider the chess player Ding Liren, the snooker player Ding Junghui or illustrious leaders, Kim Jong-Un, Kim Jong-Il for example. The last two are father and son, so what is going on there?

Well, they write what we might call their *Surname* first. If a snooker commentator is saying “Trump” for Judd Trump, then they should say “Ding” for Ding Jung-Hui, likewise, if they say “Judd”, they should also say “Junghui”. This seems to be lost on most commentators I have listened to. On a similar line, the Japanese Prime Minister recently opened a debate about how Japanese names should be represented in western culture, with the preference being how they do it themselves, surname first.

This leads us to the concept of *Family Name* and *Given Name*, saying that part of your name is inherited by default from your family and part of it is given to you by your parents choosing it. This now covers the typical western world and eastern world conventions.

Of course, one of them has to come first on a form, and which way round should reflect the majority of the users you anticipate.

Middle names can be addressed as ‘additional names’ or ‘other given names’, etc., more on that below.

Standards

One important point about all of this is, why bother trying to work all this out for yourselves? At best you arrive at the same solution as many others, at worst you make a mess internally and externally and/or give a bad impression to your users.

Also, if you wish to exchange your data with others or to provide some kind of programmatic interface (including http REST services, for example), you are much better off being standard-conforming. Less to explain and get wrong. You can simply reference the standard you chose.

To get around these issues, always seek to look for an established standard. In the case of names, you could start at schema.org. (they have a Person schema which includes names). [1]

Other types of name

Of course, there are other types of names and conventions, which would probably struggle to be contained in one generic schema (I suspect that these people have problems filling out online forms):

- Single names: Sting, Fish, Madonna. I suppose these would be given names (by themselves).
- Nicknames: These can be of single or multiple word form, but can’t sensibly be split, e.g. Marble Head, CB Bear, CeBes, Bonzo for me.

SIMON SEBRIGHT

Simon has been in Software and Solution development for over 20 years, with a focus on code quality and good practice. He can be reached at simonsebright@hotmail.com

- Posh people: Duchess of Kent, The Queen, Melbury (short for Lord Melbury, see *Faulty Towers, A Touch of Class* [2]), Sir Elton, etc.
- People from somewhere: Leonardo da Vinci, etc.
- Iceland, where you inherit your family name from the first name of your parents with a son/daughter suffix, e.g. a parent called John would have a son called Peter Johnsson, a parent called Anna a daughter called Anita Annasdottir.
- Uniquely identifying people with similar names, e.g. for authors/composers of copyrighted works. For this, I found the International Standard Name Identifier (ISNI) [3]
- Probably many more...

Development with names

If you as a developer think you can parse a full name into parts, think again. Just consider Jamie Lee Curtis and Chris de Burgh. Handy in some circumstances for ease of entering data, but only if you give the user a chance to correct it manually.

Another interesting idea came from sitting next to a Dutch guy at the German SoCraTes conference a couple of years ago. He told us, it is common to sort names by the 'main' part of the family name, so the 'van' in Dick van Dyke would be ignored for sorting, instead the 'Dyke' part would be used. They get used to that in their own country, and miss it, when they end up always in a big lump together at the end of lists in other places. ☺

Usernames

Sort of an aside, these became popular when websites started becoming applications and required you create an account. I think the main point was to have a more friendly name visible rather than Member1234 or your email address.

However, I find that because of the seemingly random restrictions of format and allowed characters, this became a real pain, at least for me. Not only choosing a name became a pain, but many sites insisted on using this as part of the login, so if you can't remember what you picked (due to restrictions), you constantly had to retrieve it via the "I forgot my f***ing username" link.

I would naturally choose 'Simon Sebright' as a username, where I am not concerned about my name being seen. However, some sites did not allow spaces, so Simon.Sebright. However, some sites do not allow punctuation, so simonsebright. I even encountered length restrictions where that was too long, so just Sebright for example. Also, some sites restrict your username to be unique across all users, then it's just pot luck if you are the first John Smith to sign up...

That is a complete joke and I suspect that some of these restrictions were not in explicit requirements, rather developers used 'common sense' during the implementation. Please think!

I have no idea what most of my usernames are any more. Thankfully, most sensible sites also accept the email address of the account, which I don't tend to forget. ☺

Middle names revisited

Thinking about middle names a bit further: We could consider the forename and middle names simply as *Given Names*, i.e. an ordered collection of strings, each of which is a given name. The 'real' forename is simply `GivenNames.First()`, i.e. a property of the class holding the name information. Note that we can't simply have one string with all these together, unless we devise a suitable separation, escaping mechanism. Run-of-the-mill names don't normally include much punctuation, but best not to rule it out.

Conclusion

Think about what you call your elements in your designs, be it in the UI, in a programmatic interface, in your database tables, or even just in class members (remember with reflection, these often creep out into the wider world). Turn to established standards where you can.

Don't place artificial constraints on what you think are normal names, be it format, characters, length, etc.

When doing user acceptance tests, if you create personas, include some with 'weird' names and unusual desired usernames, etc. Pretend to be Sting or Madonna for a day, or one of the Royals with loads of given names! ■

References

[1] schema.org Person: https://schema.org/Person

[2] A Touch of Class, Faulty Towers: https://en.wikipedia.org/wiki/A_Touch_of_Class_(Fawlty_Towers)

[3] ISNI: http://www.isni.org

If you read something in *C Vu* 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?

Thoughts on 'Computational Thinking'

Silas S. Brown considers the drawbacks of skill assessments.

I met a PhD student from Cambridge University's Faculty of Education who is engaged in an 'educational psychology' project in collaboration with Cambridge Assessment (a university-affiliated company that runs rather a lot of school examinations), and has been gathering the viewpoints of computing teachers and anyone in the computing background with a view to developing a method of assessing a skill called 'computational thinking', that is, the ability to think about a problem in such a way as to be able to program a computer to solve it.

Frankly, I am concerned about what Cambridge Assessment is going to do when they think they have a method of assessing computational thinking. Will they believe that this abstract skill called 'computational thinking' can be measured in a child before they learn any computer science, to find out whether it's likely to be worth teaching them or not? If so, the worst that can happen is it will become like the 'Eleven Plus' examination feared nationwide throughout the mid twentieth century, segregating children into streams of apparent ability before they had the chance to demonstrate how they will really develop, setting up late developers for further failure and generally increasing the gap between the haves and the have-nots. (I do have a slight bias here as my grandmother taught children who failed their Eleven Plus.) If the assessment ends up measuring 'people who are good at passing exams' instead of anything else, then it has failed and I'd hate to see it be used to screen out children from having the opportunity to learn how to program.

In the early 1990s, my parents saw an advertisement for a 'high IQ' club called Mensa and thought I might be clever enough to join. For this, I had to take a test to measure my 'IQ', and the school kindly arranged to supervise it. The exact nature of how Mensa tests are graded has probably changed over the years, so I don't know if it's the same today, but, at the time, I recall they said anyone who scored an IQ of 140 or above can join a special elite inner circle of the club, and I only got 139. (I might now be misremembering the exact figures, but I do recall that I very narrowly missed out on the elite membership. I could still get a normal membership, and I did participate for a year, but I didn't feel it was delivering enough value to justify the renewal cost to my low-income family.) But the reason why I'm telling the story now is this: Two or three weeks after I took that Mensa test, I happened to learn in a maths class how to solve simple simultaneous equations, and I immediately realised that this was the tool I was missing for one of the Mensa questions I couldn't answer. If only I'd taken the test after that maths class instead of before it, I might have scored that extra mark and got the elite membership. And then I realised the IQ test was flawed. It was supposed to be measuring something called 'intelligence', which we are told is an intrinsic quality that does not depend on what you know or what you've been taught, but here I had evidence that the result can indeed be influenced by what the test-taker has or hasn't been taught in the run-up to the test. Perhaps a child with an IQ of 200 who did not know simultaneous equations could re-invent them on the spot and get the mark anyway, but down here in the 130s it clearly did make a difference what I had or hadn't been taught, so at the very least I had discovered the IQ mark has error bars we hadn't been told about, which makes it less valuable than the advertising suggested, and if you set sharp cut-offs then the fate of borderline cases might be more random than you think.

And this shows what I am afraid of with regards to the idea of measuring computational thinking. I won't say 'there is no such thing as computational thinking' any more than I will say 'there is no such thing as intelligence', but in both cases I don't think we can reasonably expect to reduce it to a single number and measure it to three significant figures, nor do I think it possible for an assessment to separate off skills from knowledge. If there is indeed such a thing as an intrinsic level of skill you have before you learn anything, then I wouldn't expect any test to detect the difference between someone who's had a 'head start' by virtue of having more of this skill, versus someone who had less to begin with but worked harder to make up the difference.

How do we know the thing you're trying to measure has only one dimension? Someone might be better at some aspects but not so good at other aspects. Imagine trying to assess every tool in a carpenter's toolbox: does a hammer score more than a drill? Perhaps it does if we specify that the job involves banging in nails, but if we don't yet know about the job then the comparison starts to lose meaning. Or, as it's useful for all programmers to have at least some level of skill outside their main speciality, perhaps the comparison should be between two different Swiss army knives, one of which happens to have a really good blade, versus another that happens to have a really good corkscrew. Yes you could give them overall scores, but a higher score given to the one with the good blade might not help you so much when you have to open bottles. My personal weak points include so-called 'front-end' programming: yes I can just about do it, but I very much prefer if somebody else does that while I fill in the back-end code that does the actual processing. What does it mean to rank me higher or lower than a front-end programmer? True, all good tools have something in common (they're well-made, they don't fall apart as soon as you start to use them) and it might be possible to measure the programmer's equivalent of that (for example, the ability to act professionally in our craft), but if trying to measure this, we have to be very careful not to confuse it with the particular strong-points that will manifest differently in different cases.

How do we know the thing you're trying to measure has only one dimension?

With regards to learning and training, well we all know there exist different teaching methods, and, while some learners might be so good that they're going to learn anyway no matter how badly it's taught, there are other cases where the quality of teaching is going to make a difference, and there might even be different methods that are suitable to different people. I have seen this even at university level, when I meet a student who hasn't understood what a lecturer said but finally 'gets it' when I explain it differently (am I better than the lecturer? no, I'm just different, which is what that particular student needed), so students' performance at assessment might at least partly depend on whether there happened to be a good match between their best learning style and the teaching styles they happened to encounter. That might be OK as long as we realise that's what we're getting and we don't fool ourselves into thinking we're measuring some kind of innate ability that's completely removed from experience.

If there is a good predictor about whether a child would do well in computing, then if my experience is anything to go by I would suggest it may have a lot to do with how good that child is at patience and at seeing value in small things. I've lost count of the number of times someone has asked me to 'teach them programming' and then backed off when they realised they won't be able to write the next blockbuster video game on a

SILAS S. BROWN

Silas is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition and part-time for Oracle. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

a reasonable balance of letting you get into programming quickly while also showing that you can't expect to build Rome in a day

one-hour crash course. I, on the other hand, got left in the public library as a child (to save on heating bills at home), found a book that showed me the workings of a 'half-adder' (an essential part of the arithmetic/logic unit in processors), and drew out on paper the circuit diagram for a full adder (I think I even made it a 32-bit one although it was still the 8-bit era), saying one day I'll figure out how to build this so I can have a computer to use (although just the adder by itself wouldn't have been very useful, but I saw it as a good start). I salvaged circuit boards from the rubbish skip at the local telephone exchange and hoarded them, thinking I'll eventually find out how to get the transistors off and build a processor (although family members didn't think so and threw them away). I would spend hours writing out simple programs on paper, which my family tended to throw away as rubbish until a grandparent showed one to a local computer repairman who said it made sense. At one point, I could hardly look at any printed page without thinking about how the word-processor had wrapped the lines and how the print-head had moved over the paper, responding to the print-driver's commands to put down the dots one small group at a time (if you looked carefully you might occasionally have seen me trace how I thought the print-head was programmed to move). Others called me crazy, but I saw value in such details, not taking them for granted.

When I was about 8 years old, I went on a school trip to the Alum Bay glass workshop on the Isle of Wight, and after demonstrating various aspects of glass-making, they pulled off part of the molten bulb, stretched it into a long thread, let it solidify and gave it to the teacher who broke it up and gave pieces to the children (just a few centimetres each) to see what we'd make of it. I, of course, had read the right library books and knew it was an optical fibre, and I felt like I was holding the very future in my hands. Just make it 3,000 miles longer and put it in the ocean, and we could exchange huge quantities of data with America (this was before the Web had been invented, but I knew something like that would come), or perhaps one day I'd be able to make a fully-optical processor and this will be one of the links. That is, until the child next to me was cruel enough to shatter it and say, "Look how upset he gets about only a bit of glass." He saw a bit of glass; I saw the future it represented, and I found his lack of respect, his lack of interest, his unwillingness to even hear me tell him why it was special, to be even more upsetting than the loss of my fibre.I don't know who's going to be good at computational thinking, but I'm afraid people who only live in the here-and-now wanting instant gratification, people who don't see the value in starting with small things, people who break the glass without thinking how it can run an Internet, are going to be less likely candidates unless they mend the error of their ways.

Who hath despised the day of small things?
~ Zechariah 4:10

It is the responsibility of the student to be interested.
No one can be interested for you, and no one can
increase your interest unless you so will.
~ William H. Armstrong, *Study Is Hard Work*, 1956

Sadly, children nowadays are less likely to stumble across those books in the junior section of the local library, partly because fewer libraries stock such things and partly because children don't spend time in libraries. Some of them get given smartphones, each with more computing power than I could shake a stick at, but with no apparent need to program. One rather active child recently asked if I could show him how to 'make an app', and I asked if he had the patience to sit still and watch a timer count down from five minutes to zero, which seems a bit mean, but he'll need more patience than that if I start saying, "Well, you need to learn how to use a text editor, and you need to set up a compiler, and to spend time learning about some basic language constructs, and learn how to browse the class-library documentation, and some principles about adaptive display layouts, and top-down design, and principles of user interaction, not to mention meeting the acceptance criteria for the Store" etc.

When I first got access to a real computer (as opposed to paper ones), it was a BBC Micro at school which booted into a BASIC interpreter: there are better languages to learn with, but at least that environment had a reasonable balance of letting you get into programming quickly while also showing that you can't expect to build Rome in a day. Nowadays, programming languages are often not included at all, and, while Rome can be downloaded in seconds, the effort required to build it is not apparent and potential learners are ill-prepared for the shock of what they might be up against. Not that I want to discourage anyone, but they should realise this mountain means serious climbing, not just riding to the top in the tourist train. One organisation that is doing something about it is Cambridge's very own Raspberry Pi Foundation (which seems to have succeeded more than MIT's One Laptop Per Child project did a few years previously), putting 'properly programmable' computers into the hands of children and schools, although the take-up still has some way to go. MIT's *Scratch* language is also encouraging, as despite its limitations it is being used in schools to introduce many children to the idea of programming (probably not the way I would have done it, but it's something). Those who get somewhere with such things might be good candidates for further instruction, although I wouldn't like to categorically rule out those who don't.

A more advanced skill, which might need some practice, is that of looking at a set of instructions and figuring out how to 'break' them. Many instances of what we call 'bugs' are due to some programmer not completely thinking through all the possible branches their code could take. Back when I was a child in that public library, I also found a book that used an imaginary toy robot as a tool to introduce the idea of flowcharts (although I wouldn't recommend using flowcharts in learning these days), and it didn't take me long to realise there was a certain set of inputs the book's authors probably weren't expecting, which would send their robot into an infinite loop. The ability to think of such things can be a valuable asset, because you can then go back and fix your code to avoid that problem (it's especially important in security, when you're up against somebody who's not merely being 'stupid' but is deliberately looking for ways to break what you've done), and the software ecosystem in general could be better if more programmers thought 'what if this silly thing happens' before it does.

A more advanced skill is that of looking at a set of instructions and figuring out how to 'break' them

Another important skill that often gets overlooked (but which again might need some practice) is the simple realisation that a symbol or a name might carry a meaning that's different from the one you've used before. Take, for example, the humble assignment operator, which in most programming languages is an equals sign (`=`). In algebra, if you see $x=5$ then it's reasonable to take that as a statement of truth: for the purposes of this question or discussion, x stands for 5, and it always has done and always will do. Then we move over to coding, and we find **`x=5`** is now an instruction, performed by a computer at a certain point in time, meaning 'make **`x`** equal to 5'. So at times before the instruction takes place, **`x`** may or may not have been equal to 5, but at times after the instruction, **`x`** will be equal to 5, until some other instruction comes along and changes the value of **`x`** to something else. The ability of a student to get that flow of time into their head, instead of viewing $x=5$ as a statement of truth as they've been taught to do in algebra, is a surprisingly good predictor of how well they're likely to get on with the rest of programming, but I certainly hope it is a skill that can be taught rather than an innate tendency. But it gets worse: what if we see code that says **`if x=5`**? In some programming languages, this changes the meaning of '$x=5$' yet again, this time to mean 'test the value of **`x`** at this point in time and give me **`TRUE`** if it's equal to 5 or **`FALSE`** if it is not'. In other programming languages, **`if`**

Diving into the ACCU Website

Matthew Jones gives us an insight into the paddling going on under the water beneath the swan that is our website.

This article is mostly a war story, but is also a guide for anyone who wants to work on our website in the privacy of their own computer.

The requirement

Last Autumn, the ACCU committee decided it would be a good idea to offer a free 6-month trial membership to anyone attending the conference and who was not a member. The membership would only allow access to the online versions of the magazines, and would expire in September.

As the membership secretary, it naturally fell to me to make this happen. Assuming maybe 100 sign-ups, it would not be out of the question to simply gather names, addresses and emails, and enter it all by hand. It turns out that would have been quicker, and easier, but an incredibly tedious day's work.

I already have to poke around [1] in the SQL database that lives behind the website, but to do this I blindly repeat magic spells worked out by my predecessor, Mick Brookes. This gives me little knowledge about how it works, and how the new requirement fits. So I said I would have a go at adding the new feature…

A disclaimer

Many programmers will know that this is a classic LAMP [2] setup, and some readers might already know how to tackle this task. But my background is embedded, real time, and C++. I use Linux at home but as a user, not a hacker. My command line Linux fu is weak. I have never administered or written a proper website. I know only the very basics of

MATTHEW JONES

Matthew started programming with BBC Basic, and then learned C during a VI form summer job. He has been programming professionally for over 20 years, having moved on to C++, and usually works on large embedded systems. He can be contacted at m@badcrumble.net

Thoughts on 'Computational Thinking' (continued)

`x=5` is an error (which may or may not be pointed out by the computer) and if you wanted the 'test' meaning, you should have written `x==5` with two equals signs. You can argue the notation is badly designed, but coping with badly-designed systems is unfortunately one of the things we have to do. And what about variable names? Ideally, they should be well thought-out, but how often have you seen code using variable names that do not (at least to you) convey accurately the meaning of what is really stored in them? At some point, every learner has to grapple with the 'rose by any other name' idea and realise that the thing another programmer called X might be different from what you would call X yourself. Context is important, and although I wouldn't call the understanding of this a fundamental part of computational thinking, it's certainly a necessary hurdle to overcome unless you have the luxury of designing the language yourself from the ground up.

In the long term, the skills that make for good computational thinking might change. Right now, practically all computers you will program are based on the Von Neumann architecture (processor, memory, etc), and, at its most fundamental level, programming is the ability to ask yourself the question 'If I were the processor, what should I do next?' and writing down each step. (The ability to write out instructions for another human who is determined to follow them as 'stupidly' as possible might be a good start, since it gets you used to specifying everything and not leaving anything to 'common sense', which we can't expect computers to have.) Nowadays we have abstractions, like function libraries, which can make the steps bigger, meaning you won't have to write out the instructions in as much detail as you would if you didn't have some common sub-tasks pre-defined by the library, but on some level every programmer is going to have to think 'What should I do next if I were the processor?' and write it down. This will probably be the case for years to come. But there are two areas of research that just might have the potential to change it a bit: massively parallel processors, and quantum computation (although I'm not so sure about the latter). Traditional computer programmers find the programming of parallel processors to be more difficult: it's far easier to think 'What should I do if I were the one-step-at-a-time processor?' and come up with a series of steps in order, than it is to think 'What should I do if I were this huge collection of networked processing nodes?'; much research in parallelism goes into making it easier for old-style programmers to cope with without having to hold too much awareness of all the parallelism that's going on underneath. But could there be some child out there somewhere with an altogether different way of thinking that just happens to suit parallel processing better? If so, they just might be the future, and I wouldn't want to mark them down too much just because they find our old-fashioned sequential stuff too hard. If an assessment is introduced then I would like it to take an approach that is as broad as possible, not limiting oneself to only today's programming languages or design methods.

And regarding the idea of innate skills that are separate from what we learn, I am concerned that genetic traits in general are overrated. (It was the idea behind Nazi eugenics after all.) Suppose we turn it around and imagine a bad trait: suggest somebody comes to you and says 'I can't help being a serial killer, my doctor says it's in my genes'. Although there probably isn't a 'serial killer' gene, there may well be inheritable tendencies to assume a certain (mis-)balance of emotional states that make somebody more likely to become a killer. But that doesn't mean such a person 'can't help it'. It just means they might have to put in more effort to control themselves than is required by the rest of us. I personally do not get an urge to kill somebody every time they do something I don't like (which is just as well, otherwise I'd be going for all the smokers and Radio 1 listeners for a start), but if somebody out there does get such an urge yet successfully controls it, I admire them for having overcome what genetics threw at them and refusing to be a victim of their predispositions. Conversely, if someone's genes predispose them to do well at a certain skill, that does not mean others don't stand a chance: the one with the gene may have a head start, but it's surprising how quickly you can lose a head start if others practice and you don't. If we could measure just the predisposition, that measurement wouldn't tell us anything, because a person's final skills and characteristics will be made up not only of their inherited abilities but also of how they learned to control and use these – and the second is far more important than the first. ■

SQL, and I have never programmed in PHP. In fact, I only know enough [3] to truthfully rate myself at 0 or 1 out of 10 at all of these. I very quickly entered the 'valley of despair' stage and stayed there. Almost everything in this article is probably the wrong way to do it!

This is real code archaeology. It is also a classic 'how not to do it' story. I should have gathered all the information I could, read it and digested it, until I fully understood the system. Then, with skill and judgement, made a few perfectly crafted tweaks. But this is said with hindsight. When you start an apparently small job like this you just wade in assuming you can bash it out in a few evenings. Some of the references listed below I only discovered whilst writing this article. With a wry smile, I freely admit I only have myself to blame.

The system

The website was written, and for a long time supported, by Tim Pushman [4]. It was written over 10 years ago, when the chosen technology was current and appropriate. But about five years ago, he wisely decided to move on and left it to us.

The website uses a content management system called Xaraya [5]. It is so old, it doesn't even have a Wikipedia entry. It is no longer being developed or supported. Not even Stack overflow is any use. If you search for 'Xaraya', there are just five, yes FIVE, hits.

Xaraya is written in PHP and is built up from modules. There are many modules, including one for managing articles, one for book reviews, and one for Worldpay, our credit card payment system. And, of course, there is one for managing subscriptions to the site. Every member of ACCU has encountered this, whether or not you were aware of it. The subscriptions module manipulates our membership database. It is a cut & paste clone of the original Xaraya code, with many modifications for ACCU-specific features. The changes for our new trial membership fell entirely within this module; after all that is the whole point of a modular system.

I was ready to try some tentative changes ... this immediately broke the live website

Xaraya and SQL

Every Xaraya module has a `xarinit.php` file. Predictably, this gets run at startup. One of its jobs is to initialise the module's database tables if they don't exist. So the very first time the website runs, provided an empty **`accuorg_xar`** database exists, it will be set up for us. As I write this, I have just tried starting the website with an empty database but it stayed empty. After actually reading [5], it turns out there is an installation process, (again, a PHP module) that a new site must invoke.

Another initialisation task is the creation of global variables. These live in a database table called **`xar_module_vars`**, indexed by module and name, allowing each module to ask for one of its global variables by name through **`xarModGetVar()`**. This has an annoying side effect. Some of the website content is stored as strings in these global variables. So when you're trying to find some code and your only way in is a string on a website page, you have to grep the source code and the database. The original initialisation values are in `xarinit.php`, but the values might have been edited later, in SQL, meaning the live content can diverge from the source code.

Xaraya pages

I haven't been involved in the publications and book review aspect of the site, so I don't know how dynamic content works.

Static pages, such as those relating to subscriptions, come in two parts. There is a PHP file which contains a function that returns an array of data. The function will typically access the database, and maintain any state needed if the page interacts with it through buttons. The page appearance is described by a `.xd` template file. These templates are XML, which Xaraya transforms into HTML by combining it with the dynamic data from the PHP function.

Error logging

If the web server has a problem, it will log to `/var/log/apache2/error.log` or similar. This is entirely standard and predictable.

If Xaraya has a problem, it will log to `/home/accu/public_html/var/logs/log.txt`. Yes, that's `var/logs`, but 'logs' plural, and `/home/.../var`, not `/var`. And it will only log if the file already exists, and with the right permissions. All very fiddly and frustrating when you're expecting a file to appear automatically in `/var/log`. It took a while to work out this particular wrinkle.

Xaraya has two **`printf`** logging functions: **`xarLogMessage()`** and **`xarLogVariable()`**. They both do what they say, and are pretty much the only way to debug a live site. I spent a great deal of time instrumenting the code.

The work

After poking around in the PHP for a few evenings, I thought I had got the measure of the membership module and was ready to try some tentative changes. This immediately broke the live website, resulting in a couple of emails to the webmaster from puzzled members trying to renew. It was clear that working live was out of the question, especially as this is ACCU's busiest time of year: the 3 months leading up to the conference is when most of our new members join, and consequently a large number of existing members renew.

I took a step back and decide that no matter how hard it was going to be, I had to work on an offline version of the site. Time to saddle up for a long ride.

Thankfully, about 18 months ago Jim Hague put the entire website into git [6]. In principle, cloning that repo, and cloning the SQL database should get us our own private version of the ACCU website. But first we need a web server. There are many, many, how-to guides to this on the internet. It boils down to this:

1. Clone the git repo

1. Clone the repo to `/home/accu/public_html`. This is the same root path as on the real webserver. Keeping it the same gives the highest chance of it working right away, and allows us to focus on real work, rather than discovering later on that this was a mistake. Who knows how many hard coded paths there are in the code? (Answer: not zero!)
2. Create a new branch and check it out immediately.

2. Install MySQL

1. Set up an admin user, root, and the Xaraya user accuorg_xarad.
2. Install MySQL workbench.
3. Use workbench to check that our SQL server is alive and kicking.

3. Install Apache

1. Check `http://localhost` for signs of life.
2. Tell the Apache2 config file to serve a site from `/home/accu/public_html`.
3. Change the site config to use a port other than 80, just in case we accidentally expose something to the wider internet whilst making changes.

At this point, we might see a web page but we have no PHP so we just see `index.php` as raw text. (If you can't get to the git repo, use one of the trivial `index.php` examples below.)

4. Install PHP 5

At the time of writing, the default PHP version you get from apt-get install php is 7. Xaraya needs 5.x and does not work with 7, offering nothing but HTTP 500 errors.

The internet tells us to do this:

```
add-apt-repository ppa:ondrej/php
apt-get update
apt-get install php5.6 libapache2-mod-php5.6
```

At which point, the errors should change to **403 Forbidden**. Progress of sorts. Again, the internet helped me work out that we need to add this:

```
<Directory "/home/accu/public_html">
  Require all granted
</Directory>
```

in /etc/apache2/sites-available/000-default.conf

Now we can see this trivial index.php OK, but our full Xaraya home page still doesn't work.

```
<html>
  <head>
    <title>PHP Test</title>
  </head>
  <body>
    <?php echo '<p>Hello World</p>'; ?>
  </body>
</html>
```

Using this index.php:

```
<?php
  phpinfo();
?>
```

we get a very useful and detailed PHP information page that tells us everything we need to know. We just have to work out exactly what it is try to telling us...

5. Install PHP modules

We have some missing PHP modules. Each step forward results in a new and puzzling error message. We need to decipher each one to work out which module might be missing, then install it, and repeat until everything starts to work:

```
apt-get install php5.6-mysql
apt-get install php5.6-xml
```

But there is a shortcut! We can run **phpinfo** on the live website and compare the modules very carefully to find out what we are missing on our machine. Whatever the technique, by a process of iteration we end up getting Xaraya to sit happily at the top of our LAMP stack.

6. Install the database

Once the HTTP errors stopped, Xaraya error pages took their place. One of the messages was **Unknown database 'accuorg_xar'**. At least this is not cryptic. Luckily "here's one I made earlier" [7]: the real website. It just takes a few lines to copy it over:

```
ssh <username>@dennis.accu.org
mysqldump -u accuorg_xarad -p accuorg_xar > /tmp/
accu.sql
logout
scp <username>@dennis.accu.org:/tmp/accu.sql .
mysql -u root -p accuorg_xar <  accu.sql
mysql -u root -p -e 'show databases'
```

Keeping this accu.sql file handy also allows us to erase any mistakes, or newly created test users, by simply repeating the import, thus turning back time:

```
mysql -u root -p accuorg_xar <  accu.sql
```

Nice. At this point we have a snapshot of the real ACCU website running on our local host. Now the real 'fun' can begin.

The changes

I'm not going to go into detail on the changes for adding the trial membership. It isn't particularly interesting. It's pretty poor code, and I'm not proud of it, but it does the job. If you really are interested, ask for access to the git repo. Here is a summary:

1. Add a new subscription type so we can treat trial subscriptions as a special case.
2. Add a new tab to the user account page so new subscribers can choose to sign up for trial or full membership.
3. Make the trial subscription page bypass the Worldpay payment system, because its free.
4. Find every place in the existing code where the type of subscription is important and consider whether trial subscriptions need to be included (e.g. expiry reminders, renewals, etc).
5. Add a special case to subscription expiry so that trial members all expire on the 30th September (rather than the usual case of 365 days after they renewed).
6. Add a switch to the database so we can easily show and hide the trial membership feature around conference time.
7. Do an awful lot of testing with test users.

Conclusion

I have tried to explain the basics of the ACCU website system. I have shown how running your own local experimental copy of the system should be well within the capabilities of anyone familiar with the linux command prompt.

Due to its age and the difficulties in maintaining it, we tend to leave the website well alone. Routine activities centre around adding content. Actual functional changes such as this are very rare, although this work should give us some confidence that we could do more if pushed. However, the site is obviously nearing the end of its life, and it might be kinder to just call the RSPCA[8].

I hope this has given an insight into what it takes to maintain our beloved website, and by implication, what it would take to replace it. If this has piqued your interest, why not try following the instructions, or getting involved in some other way? ■

References

[1] Amongst other things, we generate the list of e-voters for the AGM with a SQL query.

[2] The LAMP software bundle, Wikipedia: https://en.wikipedia.org/wiki/LAMP_(software_bundle)

[3] The Dunning-Kruger effect: https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

[4] gnomedia codeworks: https://www.gnomedia.com/

[5] *The Official Xaraya User Guide*: http://www.xaraya.hu/files/xarayaguide.html

[6] Git repository of the ACCU website: https://git.accu.org/jim/xaraya-website [Note: Not publicly viewable since it also contains members-only content like CVu.]

[7] *Here's One I Made Earlier: Classic Blue Peter Makes* https://www.amazon.co.uk/Heres-One-Made-Earlier-Classic/dp/0857835130 (With apologies to our non-UK readers: this is a very British cultural reference!)

[8] RSPCA (Royal Society for the Prevention of Cruelty to Animals): https://www.rspca.org.uk/. Again, possibly a cultural reference. See https://en.wikipedia.org/wiki/Animal_euthanasia

The Standard Report

Guy Davidson reports on the changes that have been brought about as a result of the CoViD-19 pandemic.

According to the schedule we were following in February, the next face-to-face meeting was due to take place in Bulgaria at the beginning of June, in the coastal resort of Varna. As you can imagine, several of us were looking forward to a little extra time in the summer sun by the sea on either side of the meeting.

There was considerable concern, of course, as the coronavirus pandemic unfolded before us. We debated on the mailing lists about how we should proceed, should we cancel immediately, wait it out, take a view three months ahead or some mixture of these approaches.

Cancelling a meeting is not a trivial matter. They are expensive to host, requiring 10 conference rooms to accommodate over 200 people for a week. Attendance isn't cheap either: flying out to these locations can be an expensive proposition for self-funded attendees, so buying flights early is an important saving. Several committee members had therefore already bought flights and were reluctant to see their investment set aside by cancellation.

Ultimately though, amid conference cancellations and rumours of impending lockdowns, the decision was made for us when ISO announced that all face-to-face meetings were cancelled until the end of June. This date has since been extended to the end of July. There is no sign of this edict being lifted, and nobody is assuming that the November meeting in New York is going to take place.

We still have a standard to deliver in 2023. The 'train model' of standard delivery, where features ship in a standard if they are ready or hop on the next train to ship in the next standard if they aren't, rather relies on a solid three-year cycle. The motivation behind this approach is to ensure regular updates to the standard, rather than a repeat of the '00s where nothing was released except for the *Technical Report* in 2003.

This means that we still need to carry out the work of the committee but not in face-to-face meetings. Just as we are all getting used to teleconferences over Zoom, Teams, Hangouts, Webex and all the other video meeting platforms, so must the committee, and that nettle has been well and truly grasped. A new schedule of meetings has been devised, with weekly meetings of Library Evolution and Evolution. The schedule oscillates a little to ensure that it doesn't exclude people with other regular meetings, although it does run into the problem of having an international community, which means time zones.

Library Evolution meetings are scheduled for alternating Mondays at 15:00 UTC and Tuesdays at 17:00 UTC, while Evolution meetings are scheduled for alternating Thursdays at 17:00 UTC and Wednesdays at 15:00 UTC. This works out at early morning for the US West Coast, lunchtime for the US East Coast, teatime for the UK and dinnertime for Europe (or thereabouts).

The meetings run for 90 minutes and, at the time of writing, we've just finished the second week of this regime. It seems to be working out well. Initially I was rather aghast at the idea of remote meetings for the standard: it is easier to block out a week three times a year than it is to block out two ninety-minute meetings twice a week. I was concerned that experts would be absent from critical meetings and important decisions would be made without sufficient oversight. However, the purpose of these meetings is not to ram proposals through the pipeline, but to smooth over as many rough edges as possible, of as many proposals as possible, so that when we do finally meet face-to-face, there will be less need to spend committee time on individual papers.

The hope is that we can continue with our schedule of works for C++23 which you can see in P0592, available at https://wg21.link/P0592. In summary, we are prioritising the following for inclusion in C++23:

- Library support for coroutines
- A modular standard library
- Executors
- Networking

You will see that this list contains no language features. I'm quite pleased about this: C++20 was a huge upgrade to the language, and I think the user community needs time for the wave of upgrades to finish crashing and to recede a little.

We also expect to make progress on:

- Reflection
- Pattern matching
- Contracts

Reflection and contracts are handled in their own study groups, SG7 and SG21, rather than in EWG, but we expect to see papers emerge from those groups for EWG to consider, although not necessarily for inclusion in C++23.

Of course, this is NOT to the exclusion of all else. If you are interested in, say, ooh, I don't know, the linear algebra proposal, then that will continue to be discussed. There are several active subgroups still reviewing proposals: numerics, HMI, game-dev and low-latency, text and machine learning, to name a few, still have a queue of proposals to consider. They are likely to emit something for the consideration of LEWG or EWG: it will be up to the chairs of those groups to arrange proposals according to the priority listed above.

It's certainly not business as usual for the committee. However, we have a job to do, to continue improving C++ and shipping new standards every three years. We have adapted to new ways of working, increasing the use of teleconferencing and holding more frequent, shorter meetings to meet this goal. The committee process has never been more accessible: if you have ever wanted to participate and been thwarted by the cost of travel and accommodation, now is the time to test the waters by joining the meetings and experiencing the process.

If you need more details about how to do so, please contact me or the editor. We would be happy to point you in the right direction.

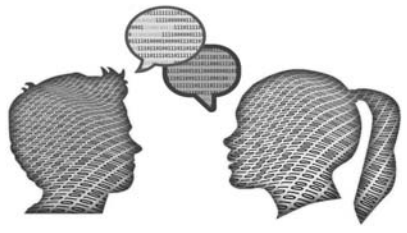
Please stay safe and healthy!

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.

Code Critique Competition 123

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 was set a challenge to read the 'Top 100 books' (from the BBC Big Read a few years ago) so thought I could start by seeing which ones I needed to get. Obviously this meant writing a program. Unfortunately, the output from the program isn't sorted, although I was expecting it would be; and in addition, it crashed on Windows.

Can you help the author fix their program so they can begin to catch up with their reading? The program (`the_big_read.cpp`) is in Listing 1.

Listing 1

```
#include <algorithm>
#include <iostream>
template <int A, int B>
void to_buy(const char *(&to_read)[A],
  const char *(&own)[B])
{
  const char** ptr = new const char *[A];
  const char **first = ptr;
  for (int i = 0; i != A; ++i)
  {
    *ptr++ = to_read[i];
    for (int j = 0; j != B; ++j)
    {
      if (!strcmp(to_read[i], own[j]))
      {
        ptr--; // already own it!
      }
    }
  }
  std::sort(first, ptr);
  for (const char **q = first; q != ptr; ++q)
    std::cout << *q << std::endl;
  delete ptr;
}
int main()
{
  const char* top_100 [] = {
    "The Lord of the Rings",
    "Pride and Prejudice",
    "His Dark Materials",
    "The Hitchhiker's Guide to the Galaxy",
    // 93 missing ...
    "Girls In Love",
    "The Princess Diaries",
    "Midnight's Children",
    };
  const char* my_library[] = {
    "The Hitchhiker's Guide to the Galaxy",
    "Vintage Jeeves",
    // and many more ...
    "His Dark Materials",
    "Pride and Prejudice",
    "Emma",
    "Where Eagles Dare",
    };
  to_buy(top_100, my_library);
}
```

Critiques

Francis Glassborow <francis.glassborow@btinternet.com>

First impressions

This is a C program with a couple of C++ elements thrown in.

Worse, it is a C program that exhibits the worst aspects of C's use of pointers.

This is a program where **`main()`** is abused to hold data

There is no protection against accidental variation in titles such as extra spaces

Updating the list of books requires changing the program with all the potential for more bugs.

Second impressions

Why is the programmer using a template?

Why single letter capital variable names?

Why have we got an explicit use of **`new`** and **`delete`**? A sure indication that the programmer has not got hold of the way C++ is designed to work at a high level.

Third impression

Do I really want to try to unravel all those pointers? I am sure the problem lies in those pointers, and almost certainly in the pointers passed to the **`sort`** function.

OK, time to put it into a compiler. Well, it compiles and runs without error. Well that means that the error is almost certainly buried in those pointers. Let us think about that call to **`sort`**. Exactly what is it sorting? Aha! An array of pointers and, of course, they are pointers to string literals (i.e. non-modifiable arrays of **`char`**) are already in numerical order. Had the programmer stuck with C and used **`qsort()`** they would have to have provided a comparison function because C lacks the idea of a defaulted parameter. Unfortunately the default for comparison used by **`std::sort()`** is wrong for this task.

So there is the clue, **`std::sort`** has a third, defaulted, comparison operator. Writing one requires delving into the mysteries of operator overloading or, more recently, writing a lambda function.

So the first solution is to add:

```
struct {
bool operator() (char const * & x,
                 char const * & y)
```

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.co.uk

```
  {return strcmp(x, y)< 0;}
  } custom_less;
```

I have deliberately used an anonymous `struct` (C style) and then provided an instance at the end. This code is likely to look weird to the original programmer but it effectively provides an object (`custom_less`) that behaves like a function that returns `true` if the first string literal is before the second one in a standard collation sequence.

Now replace the call to `sort` with:

```
  std::sort(first, ptr, custom_less);
```

And the result will be the one desired.

Even more magical for C programmers writing C++ is to use a lambda function where the comparison function is written inline and the call to `sort` is replaced by:

```
  std::sort (first, ptr,
  [](char const *& x, char const *& y){
  return strcmp(x, y) < 0;});
```

Rewrite

Having fixed the original program using arcane C++ magic it is time to rid ourselves of the arcane C magic used to write the original program. The following program is written in a fairly primitive version of C++ without much use of modern features so it should be reasonably easy for the original programmer to follow.

```
  #include <string>
  #include <vector>
  #include <fstream>
  #include <algorithm>
  #include <iostream>
  const std::string get_title(
    std::ifstream & s){
    std::string title;
    std::getline(s, title);
    return title;
    }
```

The function `get_title()` is a place holder for a function that can extract titles from a file with other data such as ISBNs and authors. The test files contain one title per line and are terminated with END as the final entry. Any necessary complexity in getting a title can be placed in this function.

```
  void load_data(
    std::vector<std::string> & data,
    std::string const & source){
      std::ifstream s(source);
      if( !s ) throw source + " does NOT exist";
      std::string title;
      std::string const END("END");
      title = get_title(s);
      while(title != END) {
        data.push_back(title);
        title = get_title(s);
      }
      std::sort(data.begin(), data.end());
      return;
  }
```

The function `load_data()` uses `get_title()` to extract titles from source and copy them into the provided `vector<string>`. Before returning it sorts the vector so that the titles are now in alphabetical order. It is called twice, once to capture the reading list and once to capture a list of books owned.

Note that it checks that `source` correctly names a file. If it does not the function aborts by throwing an exception.

```
  int main(){
    try{
      std::vector<std::string> read_list;
      std::string file
        {"D:\\data_files\\to_read.txt"};
      load_data(read_list, file);
      file = "D:\\data_files\\my_library.txt";
      std::vector<std::string> my_books;
      load_data(my_books, file);
      for(auto&& title :read_list){
        if(my_books.end() ==
          std::find(my_books.begin(),
            my_books.end(), title)){
          std::cout << title << '\n';
       }
      }
    }
    catch(std::string error_message) {
      std::cerr << "***MISSING FILE***\n"
                << error_message << std::endl;}
  return 0;
  }
```

The only thing that may seem strange to our novice C++ programmer is that I have used the range for that came into the language in 2011. It works so nicely and produces code that is easier to read.

I hope the reader notices that there is not a single explicit use of a pointer (of course they are there but deep inside library code which is where they belong.) Real C++ rather than C using a couple of C++ features is so much more comfortable to use.

Ovidiu Parvu <accu@ovidiuparvu.com>

The source code given in this Code Critique is platform independent. Therefore, I have analysed and updated it in my preferred environment, namely Ubuntu 18.04.4 using clang 10.0.0, and tested it on both Ubuntu and Windows. On Windows, the Visual C++ compiler 19.15.26732.1 for x86 was used.

There are two issues reported with the program compiled from the given source code. First of all, the program crashes on Windows. Secondly, the book titles printed to the standard output are not sorted in alphabetical order.

I was unable to reproduce the crash on Ubuntu or Windows. However, I assume that the program can crash. Due to the use of raw pointers and the manual memory (de)allocations, I hypothesized that the crashes are memory management related. To verify this hypothesis, I have compiled the source code with address sanitizer support enabled as follows:

```
  clang++ -Wall -Werror -Wextra -std=c++17 -g\
    -O3 -fsanitize=address the_big_read.cpp\
    -o the_big_read
```

Running the resulting program revealed the first major issue in the source code, namely that the array of string literals (i.e. `const char**`) created at the beginning of the to`_buy()` function is deallocated incorrectly:

```
  ./the_big_read
  the_big_read.cpp:29:3: error: 'delete' applied
  to a pointer that was allocated with 'new[]';
  did you mean 'delete[]'? [-Werror,
  -Wmismatched-new-delete]
    delete ptr;
  ...
```

There are two issues with the deallocation. Firstly, a `delete` instead of `delete []` expression is used to deallocate the array of string literals. Secondly, the pointer used in the delete expression is `ptr` instead of `first`, and `ptr` may not point to the beginning of the array when the `delete` expression is executed. To address this issue `delete ptr` should be replaced with `delete [] first`.

The second reported issue, namely that book titles are not printed to the standard output in alphabetical order, could be reproduced on both Ubuntu and Windows. The cause of the issue is that the book titles to be printed, represented as `const char *`, are sorted by character string address rather than character string contents. This issue can be fixed by calling the `std::sort()` overload that takes a comparator lambda function as input which compares character string contents rather than character string addresses.

Although the changes described above address the reported issues, the source code can be improved further.

There are three main outstanding issues. Firstly memory is managed manually, which is error prone. Secondly it is possible to inadvertently define duplicate book titles in the **top_100** and **my_library** collections. Thirdly, the time complexity of computing the collection of books to buy could be reduced. All of these issues can be addressed by changing the type of **top_100** and **my_library** to **std::set<std::string>** and computing the collection of books to buy using **std::set_difference()**.

Computing the collection of books to buy using **std::set_difference()** rather than two nested loops reduces the time complexity of the **to_buy()** function from **O(N * M)** to **O(N + M)**, where **N** and **M** are the sizes of the **top_100** and **my_library** collections, respectively.

The execution time impact of representing the collections of books as **std::set**s and computing the collection of books to buy using **std::set_difference()** was measured using [google benchmark] (https://github.com/google/benchmark). Results suggest that using **std::set** and **std::set_difference()** leads to an approximately 14x speedup compared to using **const char*** arrays and nested loops.

```
$ ./benchmark
2020-03-29 19:19:52
Running ./benchmark
Run on (4 X 3700 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 256 KiB (x4)
  L3 Unified 6144 KiB (x1)
Load Average: 0.34, 0.63, 1.12
***WARNING*** CPU scaling is enabled, the
benchmark real time measurements may be noisy and
will incur extra overhead.
...
UsingNestedLoops
     36772 ns        36770 ns        18864
...
UsingStdSetDifference
      2558 ns         2558 ns       271562
```

Note that the measurements included defining the collections of books.

Following on from replacing **const char*** arrays with **std::set<std::string>** collections, the `string.h` header include, which should have been **cstring**, was replaced with a **string** header include.

There are two additional minor issues that could be addressed. Firstly **std::cout** does not need to be flushed after each book title is printed. Therefore **std::endl** should be replaced with **'\n'**. Secondly the safety and readability of the source code could be improved respectively by enclosing the **std::cout** expression in curly braces, by renaming the **to_buy()** function to **print_to_buy()**.

Finally, given that book titles are known at compile time, the collection of books to buy could be computed at compile time as well. However, given that **constexpr** support is not yet available for the **std::set** type, the **constexpr** idea was not explored further here.

The revised source code for this Code Critique is made available below.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
void print_to_buy(
  const std::set<std::string>&to_read,
  const std::set<std::string>& own) {
  std::set<std::string> books_to_buy;
  std::set_difference(to_read.begin(),
    to_read.end(), own.begin(), own.end(),
    std::inserter(books_to_buy,
    books_to_buy.end()));
  for (const std::string& book : books_to_buy) {
    std::cout << book << '\n';
  }
}
int main() {
  const std::set<std::string> top_100 = {
    "The Lord of the Rings",
    ...
    "Midnight's Children",
  };
  const std::set<std::string> my_library = {
    "The Hitchhiker's Guide to the Galaxy",
    ...
    "Where Eagles Dare",
  };
  print_to_buy(top_100, my_library);
}
```

Hans Vredeveld <accu@closingbrace.nl>

Looking at the code, I get the impression that the author is a C-developer that learned some C++. With a few changes to the program it could compile as a C-program, where the biggest change would be in the signature of **to_buy**. It has to change from a function template to a normal function that takes the array sizes, **A** and **B**, as two additional function parameters.

Before we change the program into a modern C++-program, let's have a look at the things that can be improved.

Except for loading the lists of books to read and books in the library, **to_buy** does everything in its body: selecting the books to buy, sorting those books and printing that sorted list.

The meaning of **ptr** and **first** is a bit mixed up. When **ptr** is declared, it becomes the owning pointer to an array, but later in the function it is used to move forward and backward in that array. At the end of the function it is used to delete the array, which fails because **ptr** isn't pointing to the beginning of the array any more. Swapping **ptr** and **first** in the first two statements and in the last statement would solve this.

In the last statement of **to_buy**, **delete** has to be **delete[]** (together with the former point, this solves the program crash).

The statement ***ptr++ = to_read[i];** does two things. It assigns to ***ptr** and it increments **ptr**. I prefer to have a statement do only one thing and do the assignment and increment in two separate statements.

The **if**-statement **if (!strcmp(to_read[i], own[j]))** abuses the implicit conversion of an **int** to a **bool**. It reads like 'if not compares **to_read[i]** to **own[j]**', or 'if **to_read[i]** is not equal to **own[j]**'. The statement actually means 'if **to_read[i]** is equal to **own[j]**', that is better written as **if (strcmp(to_read[i], own[j]) == 0)**. Even better would be to make **to_read** and **own** arrays of **std::string**, in which case it would become **if (to_read[i] == own[j]**).

If an entry of **to_read** appears twice in **own**, **ptr** is decremented twice. In the best case, a book that should be bought is not bought. In the worst case, **ptr** is moved to before the array and then the location it is pointing to is written to, resulting in data corruption or a program crash.

The purpose of the inner for-loop is to find **if to_read[i]** appears in the array **own**. A safer way to achieve this is by using **std::find_if**, which also makes the code's purpose immediately clear.

The statement **std::sort(first, ptr);** doesn't sort the titles, but the pointers to the titles. This is the reason that the output is not sorted.

In the last **for**-loop in **to_buy**, the list of titles to buy is printed. This binds the function **to_buy** tightly to the standard output, making it unusable for other situations. To reduce this binding and make **to_buy** more versatile, it should return the list of titles to buy and leave the printing to the calling function, **main**.

In **main**, **top_100** and **my_library** are arrays of **const char*** and their content is hard-coded. Instead their type could be changed to **std::array<std::string, A>** and **std::array<std::string, B>**, but then we would have to hard-code the values of **A** and **B** (unless we can use C++20's **std::to_array**). When the program would work as expected, the first feature I would like to add, would be to read the contents for **top_100** and **my_library** from files. In that case we don't know their sizes and it would be logical to make their type **std::vector<std::string>**. So I advise doing that right away.

The author already found their way to the **<algorithm>** library for **std::sort**. But **<algorithm>** contains more useful functions. With **std::set_difference** we can easily select the titles to buy, under the condition that the top 100 list and the library are sorted and the lists don't contain duplicate entries.

Putting it all together, the program becomes:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
using book_list = std::vector<std::string>;
book_list to_buy(book_list const& to_read,
  book_list const& own)
{
  book_list new_titles;
  std::set_difference(
    to_read.begin(), to_read.end(),
    own.begin(), own.end(),
    std::back_inserter(new_titles));
  return new_titles;
}
void print(book_list const& titles)
{
  for (auto const& title : titles)
    std::cout << title << '\n';
}
int main()
{
  book_list top_100 = {
    // titles removed for brevity
  };
  book_list my_library = {
    // titles removed for brevity
    };
  std::sort(top_100.begin(), top_100.end());
  top_100.erase(
    std::unique(top_100.begin(),
    top_100.end()), top_100.end());
  std::sort(std::begin(my_library),
    std::end(my_library));
  my_library.erase(
    std::unique(my_library.begin(),
    my_library.end()),my_library.end());
  auto titles_to_buy =
    to_buy(top_100, my_library);
  print(titles_to_buy);
}
```

Balog Pal <pasa@lib.hu>

I almost sent an entry to the previous one. That was starting with "Hmm, we need a full book to cover the problems related to these few lines, or at least several chapters." Then I decided to take the lazy way out, with maybe writing a standalone article if the others leave something important out. This one looks much better fitting. And I really like that it is plausible to come from real life. And it even has the bait with the crash… ☺

The first thing we notice is that we have the whole solution in a single file, no headers, no fluff, and probably would be just a **main()** if the OP found a way around using template. That is great. This is meant as a single-use tool to be thrown away. The last thing we want is over-engineering. The problem can probably be solved in a text editor or using the unix shell tools, but those not familiar probably can solve it with python or C++ in a similar timeframe.

I start inspecting in **main()**. We have the inputs embedded, instead of read from some outside source. Again, that is a good thing. What I miss is top-level **const** on the collections, as we clearly have no intention to patching them up. Using **const char *** is not incorrect, however I mark it for attention to avoid comparing pointers instead of strings. Use of old-style array is also suspect. With the latest C++ the old arrays are not that hated, we can even iterate on them without fiddling with the size. But excuses on not using **std::array** are in short supply. And the rest is a single call to the template function, that seems to just serve to deduce the bounds. That could have been just done using **size()**. Or we could say **std::array top_100 const = {...};** and have everything deduced. (Finally… the need of the explicit bound was a serious impediment before C++17.)

Before inspecting how the material is processed, let's take a detour and think about the problem and the solution. That is my usual method on code reviews, reading the supplied solution can bend the thinking process too much. While if I came up with the same solution increases the confidence.

When I read the problem the first thing that jumped to my mind was that it sounds awfully like an interview question. I kinda heard the whole conversation.

Interviewer: I'm interested how you write programs, I'll fetch a keyboard, can you write some code for me?

Me: No.

I: Come again?

Me: You heard it right, I said no as in no way.

I (baffled): How come? Can't you write code?

Me: Sure I can, and I do a lot. When the circumstances are right. Not in ad hoc way. And especially not starting with a keyboard. It has a level requirement of like 4 and we did not even see the dungeon entrance. It is not even supposed to be in the room serving as distraction. But I'm happy to talk about you problem, figure out a solution. It has some chance to be used at some later point.

I: (Should I just kick this weirdo? I have nothing to do for another 5 minutes so let's roll with it for a while)

Ok, so suppose I need to complete reading a bunch of books listed on a website. I have a small library, but at a glance I miss plenty of those. I have an excel sheet with my inventory and want to get a shopping list for the missing items. Write me a C++ program that provides it.

Me: I think I understand the problem and believe it can be solved. Let me put it in formal terms and you verify that it matches your aim.

I: Ok.

Me: So, we have a set of books (I form a circle using my left thumb and index finger) that I already have, let's call it H. We have a set of books (showing similar circle with my right hand) the website requires, and so I want to have, let's call it W. So the intersection (I overlap the circles like an improvised Venn diagram) has the books that I already have. And the difference W-H has those I need to obtain. The goal is to have this latter.

I: Yeah, you got it right.

Me: Fine. That is definitely doable if we can extract the information. Can we? If so, what and in what format?

I: Yes. You can get a list of titles. You can choose the format within reason.

Me: Well, I can just export the Excel into CSV, and that I can even just paste right in the code. Can the website also give me a CSV too?

I: Yeah, that works.

Me: That settles the inputs and the processing, what do you want as the output?

I: Whatever goes, you can just dump the titles on the console one title per line.

Me: Anything else?

I: It would be nice to have it sorted.

Me: And you want a program in C++? I think I can just arrange it right in the Excel or using shell tools.

I: I agree it's somewhat arbitrary, but let's make C++ a hard requirement. Is that a problem? (Prepared for some really lame next excuse.)

Me: No, the client is entitled to have whatever requirements, my job is just to make sure he means them and that we have proper understanding. I certainly point out if it is possible but not feasible while better ways exist. For this case we can go ahead, as it doesn't look a big deal.

I (relieved): I'm glad to hear that.

Me: So, good news, C++ happens to have kinda native support for this case. So it can be done in just a handful lines of code that almost matches the formal description. We have a function **`std::set_difference`**. If we just feed it with our input lists, will provide the desired output. We can even directly make at as you described passing an **`ostream_iterator`** looking at **`cout`** and newline as the delimiter. It requires two sorted ranges with the content of sets W and H. We have luck with that too, as we have **`std::set`** that is auto sorted. So we just construct the **`set<Book>`** const using brace init with the text in the CSV files. And pass **`.begin()`** and **`.end()`**. That's it, we're done. By the way it happens to emit the output in its original order, that was sorted, so you have that part too.

I (tempted to ask to repeat the part after 'good news' to catch up): Are we?

Me: Well, almost, we have one open point on **`Book`**. To work in the setup it needs three things (beyond being fit in a container): implicit construction from string literal, less operator creating an order that fits for us and the standard ordering, and usable with **`ostream`**. Neither is a big deal. For details we need additional design discussion. Say in the constructor it could invoke some web service passing the title and maybe get an ISBN number. Or do some transformations on the title like losing the punctuation, the international characters, make all lowercase, maybe remove all the whitespace. I think I would be unhappy to buy a second copy of a book just because I had an extra space somewhere. Or a different apostrophe from the wide selection. We could store the original title and the converted/gathered info, using that for the compare. Unfortunately, that could mess with the order of the final output, say, by ISBN instead of title… We can compensate for that collecting in a vector then sort that using a different criteria before copying to **`cout`**. Any specific requirements?

I (feeling overwhelmed): No, I'm fine with the simplest solution, just **`strcmp`** is fine on the title as provided. So would you write a **`Book`** class?

Me: In that case there is no reason, we can just use **`std::string`** as **`Book`**, it passes all our criteria out of the box.

I: Couldn't we just use the **`const char *`**s from the literals without extra steps?

Me: It's possible too. Of the requirements stated earlier it passes all but the sorting one. For our desired semantics at least. But we can make up for that, both set and the algo can take a comparator as additional parameter. Easiest is to create a function like **`auto const cmp = [](char const* a, char const* b) { return strcmp(a, b) < 0;};`** and feed it as the second parameter of the set and 6th parameter of the function. We even save some characters, as we can just say set **`W = {{"a", "b"}, cmp};`** without template arguments and let it be deduced.

Me: I think we covered all the necessary ground. If you're still interested, you can bring the keyboard now and observe that my fingers are indeed pretty lame using an unfamiliar specimen in a foreign environment.

I: No, thanks, we're good. (Maybe this guy has a point there, all the other folks just jumped on the keyboard and the few who even provided something that works arrived at some arbitrary variant without alternatives considered or even a hint that the result will match the desires before a lot of time invested… Next time I'll at least wait till the keyboard is demanded.)

After the detour we covered the solution, so let's see what OP provided instead of that. In function **`to_buy`** (that shows poor naming that would not be a problem if we never had the function in the first place…) we spot use of **`strcmp`** and **`sort`** that are good signs. We also spot use of **`new`** and **`delete`** that is a very bad sign: we found no reason whatsoever to deal with raw memory to start with, and if we had a reason wild delete is still on top of the forbidden list: allocated stuff should go to an owner like a **`unique_ptr`**. To add injury to the insult we use array **`new`** matched with plain **`delete`**, that is undefined behavior – good enough reason for the observed crash. Even if we managed to pass the proper pointer, the one pointing to the allocation. In this case **`first`**, not **`ptr`**. Keep in mind to properly match your delete not using delete avoids this whole can of worms.

Now, what are we using the memory for? Looks we're collecting the titles for the output. Like as we had **`vector<char const*>`** reserved to size A, **`push_back`** the item under scrutiny, then **`pop_back`** if we found it in **`own`**. An obvious improvement would have been doing just that and avoiding the crash while making the code more obvious (while probably doing the exact same thing under the hood.) An obvious chance to further improvement would be to avoid the jojoing, and just collect the item when found worthy. Maybe OP tried first to issue a **`continue`** when **`strcmp`** came out with 0. Then realized that he meant to **`continue`** the outer loop, not the inner. Might even made some noises mentioning WG21 and evil killing all the proposals to have the labeled **`break`** and **`continue`** other languages have… Then resorted to use the bouncing instead of the prescribed good measure that sits in the language: **`label`** and **`goto`**. Or the more prescribed measures like replace the loop with the algorithm it tries to re-implement: that would be **`find_if`** in this case. What would also make the code more readable, meanwhile avoiding the **`continue`** of the scan after a hit for possible other instances of the title in the list, that are not there. Hopefully, as if they are we start popping other items from the collection, then hitting UB after passing first.

Suppose we fixed the mentioned issues, and get to sort with a proper state. Did I mention up front to look out for using **`strcmp`**? Well, we did at one place, did not with this second. Sort works fine, certainly, but it will create the order by the pointer values, not strings, so the output may be whatever for practical purposes. We need to pass in a sorting function here too. Could have avoided the whole thing by just using **`std::string`** as holder.

The rest is more like nitpicking, on using **`endl`** without a good reason, not including all the required headers, not using **`auto`**… probably boring details considering the amount of real problems and missing the simple and effective solution.

Effective? one could ask as objection. Doesn't using **`string`**, **`vector`**, **`set`** and the other goodies waste plenty of memory and processing cycles? Stupid question… NOT! (TANSTAASQ!) Performance is certainly a concern… WHEN it is. If we run this program and it reports out of memory, or crashes due to that, or takes hours to produce the output, then yes, we're wasting stuff that we might be able to conserve. And in that case it is worth to look closer. However if the output is on the screen before we release Enter all the way? Then we can just acknowledge that we have our goal completed. (Provided the answer is correct that is). If it took a fill 80 extra nanoseconds to run, so be it. Or used an extra 10k out of our 6G pool of free memory. (left from the 14G pool, by the 312 open tabs in our browser and 2 dozen gizmos in the alert area we know nothing about).

Anyway, let's give it some consideration. This solution has N^2 scaling over the NlogN using **`sort`** and **`set_difference`**. So we can expect the latter to consistently win as we get to big numbers on the list size. We know that one list has 100 members, the other is unknown, probably in the 50–3000 range. Well above the 8 that is a popular cut-off point.

As the memory footprint, **`set`** is fairly wasteful, if concerned we're better off using **`array`** and sort it (that could be better on time too.) **`std::string`** certainly has an overhead over the plain literal, and we only used it to avoid passing the compare function. On the execution level it is a poor tradeoff. But it has benefits on much harder to mess up and easier to read/review.

And as an extra point tied to performance, the whole consideration may be moot. Let's notice that our program is pretty shy on observable behavior. The only observable behavior is the output itself. So the compiler has freedom to just have the final output text in the image and dump that. On every variant (that we fixed to be UB-free and correct wrt. our goal). I would be surprised if none of the other entries have shown a solution that works consistently at compile time. It can be done. Sprinkling the code with **constexpr** and arranging it to be compile-time friendly may help today. But a good optimizer is allowed to go ahead without all that help. And some might even do it already.

James Holland <jim.robert.holland@gmail.com>

The *CVu* magazine's version of the program needs `string.h` included to provide a definition of **strcmp()**. The [accu-general] email version has **string.h** included.

There are two errors in the code that are causing undefined behaviour and probably causing the crash when run under Windows. Firstly, the wrong pointer is being used to delete the dynamically created array. The pointer **first** should be used, not **ptr**. Perhaps the mistake would not have happened if the first two lines of **to_buy()** had been written as follows.

```
const char ** const first = new const char *[A];
const char ** ptr = first;
```

In this way it makes it clearer that first is meant to be the custodial pointer and would, therefore, be used to **delete** the array. Also, by making first a **const** pointer prevents its value from being changed. Secondly, the non-array form of **delete** is used to destroy the array. The required form is **delete[]**. The code now compiles and runs. The problem of the output not being sorted remains.

The **sort** function is doing exactly what it is told. It is sorting the elements of the array pointed to by **first** according to the value of the elements. The elements of the array are already in ascending order as they point to strings that are at successively higher locations in memory. This can be demonstrated by sorting the pointers in descending order by adding the **greater()** predicate to **std::sort**. The list of books to purchase now appears in the opposite order. In either case, the array is not sorted in alphabetical order of their strings. The sort function needs modification.

As indicated above, the sort criteria is incorrect. Instead of sorting by the values of the pointers in the array, we need to sort according to the alphabetical order of the strings pointed to be the pointers. The **sort** function can be modified by adding a predicate in the form of a lambda expression.

```
std::sort(first, ptr, [](const char * const a,
  const char * const b){
  return strcmp(a, b) < 0;
});
```

The lambda makes use of **strcmp** to compare the two strings and returns **true** if the first string comes alphabetically before the second string. With this modification, the program produces the desired result.

There are a few ways in which the program can be improved. When it is discovered that a required book is already in the library, there is no need to continue searching the library for the same book. A **break** statement can, therefore, be added after decrementing the **ptr** pointer.

Although it does not make any difference when using built-in types, it is conventional to use the pre-decrement operator in preference to the post-decrement operator when decrementing a variable. With user-defined types, the pre-decrement operator is usually more efficient. In keeping with convention, the statement to decrement the **ptr** pointer should be **--pre;**. Also, there is no need to flush **std::cout** when printing the books to purchase. A simple **'\n'** will do instead of **std::endl**.

The parameters of **to_buy()** could be declares **const** (as well as to what they point). This will help, to some extent, make the code self documenting. The pointer **first** could also be declared **const** for the same reason. If we want to be thorough about this, the pointer **q** in the **for**-loop that prints the result could be declared as a pointer that points to a **const** pointer, i.e. **const char * const * q**. The pointer itself cannot be declared **const** as it will be incremented within the loop.

The names of the template parameters of **to_buy()** could be a little more descriptive. Perhaps **A** and **B** could be renamed **to_read_length** and **own_length** respectively, or something similar.

The supplied program has been written using very low-level constructs. There are pointers and even pointers to pointers. The student has chosen raw pointers to be responsible for deleting objects created on the heap. A better choice would be to use **std::unique_ptr** because the arrays would be automatically deleted **when std::unique_ptr** goes out of scope, even when an exception is thrown. Better still would be to use **std::vector** when relatively large arrays are required as they will delete themselves when they go out of scope. Low-level features may be required occasionally but they make the program difficult to write, understand and maintain.

Despite this, there are some interesting features to the program. One particularly intriguing feature is the use of a template to define the types of the parameters passed to **to_buy()**. Without a feature like this the array type would decay to a pointer to a **char** (in this case) and the size of the array would be lost. It is sometimes the case that 'clever features' are a sign that the program is too elaborate and complicated and that there is a simpler way to construct a program that performs the same function.

As mentioned above, the student's software is written at the very low level. If it could be determined what the software is required to achieve in fundamental terms, then it might be possible to use some high-level constructs (preferably provided by a library) to write a program that is shorter and simpler to understand. After a little thought, it is realised that what is required is for the program to print the elements of one set that are not in another set. That is all! This is such a fundamental and useful idea that there must be existing code to do the job. Well, there is and it's in the C++ standard library by the name of **set_difference()**. This function takes two sets, calculates the difference (the elements that are in one set and not in the other) and then copies them, in sorted order, to an output stream. This is exactly what is required. My version of the program makes use of **set_difference()** and consists of just three C++ statements. By making appropriate use of an existing implementation of an algorithm, the program has become considerably simpler.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <set>
#include <string.h>
int main()
{
  const std::set<std::string> top_100 {
    //...
  };
  const std::set<std::string> my_library {
    //...
  };
  std::set_difference(
    top_100.cbegin(), top_100.cend(),
    my_library.cbegin(), my_library.cend(),
    std::ostream_iterator<std::string>(
      std::cout, "\n"));
}
```

But what about the speed of execution? The student's code is specially written to solve a specific problem, so surely a general purpose library function must be slower. The best way to find out is to perform some tests. I arranged for **my_library** and **top_100** to have the same number, **n**, of random titles. I then added a further 50 randomly generated titles to

top_100. I then ran the two programs and timed their execution. The results for various values of **n** are shown in Table 1.

Table 1: Program Performance Test Results

Number of titles (n)	Execution time / seconds		Performance factor
	Student's program	Using set_difference	
1	0.00029	0.000239	1.21
10	0.000249	0.000245	0.98
100	0.000358	0.000292	1.22
1000	0.00214	0.000203	10.55
10000	0.17435	0.000737	236.56
100000	35.194	0.01183	2975
1000000	> 1800	0.135	> 13333

After waiting half and hour for the student's program to work its way through a million books, I gave up and put it out of its misery. For values of up to about 100 there is not much difference in the speed of execution between the student's program and the one using **set_difference** as the execution time is dominated by the time taken to print the result. As the number of books increase, it can be seen that the program using **set_difference** gets significantly faster than the student's code. The execution time of the student's code increases exponentially with **n** whereas the **set_difference** program increases linearly with **n**. The way **set_difference** achieves this performance is by relying on the data sets being sorted. This is ensured because **std::set** stores its data (in this case, the film titles) in sorted order. By spending a little time sorting the data at the beginning of the program allows **set_difference** to have very good performance, especially with large data sets.

In summary, it is always worth thinking about what a program is required to do in abstract terms and then seeing whether there is an existing algorithm that can be used to perform the task. A great deal of thought by some clever people has gone into the design of library functions and it is often advantageous to make use of their labours. The advice is to familiarise oneself with the C++ standard library (and other libraries) to see what algorithms and functions are available and attempt to program at a higher level of abstraction.

Commentary

First, a comment about `string.h` which, as James noted, was in the version emailed to accu-general but not in that printed in the magazine… This is because needing this header depends on which compiler/library combination you are using. In C++ it's unspecified which other things may be included when you use a standard library header and on one of the environments I tested the header was implicitly included. While there are some good reasons for this flexibility, it does make writing portable code is little harder (and even code that works with different versions of the *same* compiler). This thing to bear in mind is that you should list in your source the include files the code requires. So since the code in this critique uses **strcmp** it *ought* to include **<cstring>**.

This critique is an example of what Pal describes as 'a single-use tool to be thrown away'. The context can matter: in this case in particular, the list of 100 top books from the BBC Good Read is a given, so having it in source code for a program of this kind may be perfectly fine; however, I suspect the list of books in 'my library' ought to be held separately as, presumably, I am going to be purchasing some of the books that I'm missing and so my library will change! However, unless I have very deep pockets, my library is not going to be more than a few thousand volumes, so worrying too much about the performance of larger collections is likely to be out of scope in this particular case. (Of course, programs (and fragments of programs) can get re-used in very different environments, but I think it is the person re-using the code that needs to consider the performance issues since only they know the actual constraints in the new use case.)

As several people noted, the code is a C++ program but with a strong flavour of C. There seems no reason for the use of C features (such as raw pointers and use of **strcmp**) and I think it makes sense to try to provide the programmer with a more idiomatic C++ program. This includes making using other algorithms, such as **set_difference**. Looking at James's solution, for example, just above this commentary, the program consists of two statements for initialisation and one to actually do all the work of the program. The result, in my mind, is a program that is obviously correct. To quote Tony Hoare's famous dictum:

> There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies, and the other way is to make it so complicated that there are no *obvious* deficiencies.

We should encourage newcomers to C++ to use the algorithms! We have quite a few, and teaching people to know them, and how to recognise when they can be used, is very useful.

The winner of CC 122

All the critiques correctly identified that the sort order was broken because of sorting the pointer values of the string literals. Moving to a value-based type, such as **std::string** (or, in this case, **std::string_view**), provides an easy way to solve that problem.

It was good to see that Ovidiu, for one, reached for a tool (in this case address sanitizer) to help diagnose the crash. Tooling can be an easy win for finding problems like these (and even latent ones that are currently without symptoms.)

Francis noted the program has no protection against bad typing or spurious spaces and both Ovidiu and Hans noted that a duplicate title in one of the lists would cause problems, especially with the original code. Francis's separation of the **get_title** function as a place to add logic to do something cleverer with the title list was a good idea.

I liked Pal's attempt to fill in a possible back-story to the critique, and thought this was a creative way to put across some of the points in his critique. I also liked his use of CTAD (class template argument deduction) for **std::array** - as he says, this removes the pain point of having to provide the array bound explicitly.

The measurements of the performance costs of the algorithms over a range of input sizes from James were very interesting: they show how much difference the right algorithm can make at scale.

Overall there were some good solutions (and very little left uncovered) so thanks all for your entries… and I am awarding the prize for this critique to Pal, by a short head.

Code Critique 123

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

A shorter critique than usual this time, and in C, to give some variety.

> Please find a bug in my code based on finding whether a positive integer can be expressed as pow(a, b) where a and b are non negative integers and b>1. My code outputs for 1000 (10^3), 0 where it should be 1. Please find the bug…

Thanks to Francis Glassborow for passing this example on to me, originally from alt.comp.lang.learn.c-c++. He also points out that this competition is now in its 21st year (he originated it). That means it has been running for longer than most software/developer magazines.

The code is in Listing 2 (`pow.c`) and Listing 3 (`test.c`), both overleaf. Can you help?

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.

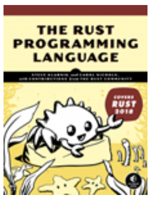
Reviews

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.

The Rust Programming Language



By Steve Klabnik and Carol Nichols with contributions from the Rust community, published by No Starch Press (2018), 511 pages, ISBN: 978-1-59327-828-1

Reviewed by Daniel James

This is the official book on Rust. It's available free online (doc.rust-lang.org/book) and also as a paperback book. I read the dead-tree version but referred to the online version while playing with code, so this review is based on both. The online version has the advantage of being more up-to-date, but I didn't find many differences between the two. This was the first book on Rust I'd read, so it was mostly new material, though I had known of Rust's goal of eliminating certain classes of bugs by catching common errors at compile time.

The book opens with instructions for installation of the compiler and its associated tools, and illustrates the use of cargo – the package-manager-cum-build-system – by taking the reader through the process of writing a simple command-line number-guessing game. Cargo is a powerful part of the Rust environment, so the treatment here is useful. The programming exercise gives a flavour of the Rust language, but in doing so it uses (without fully explaining) some of the key language elements that are not described until later in the book, which I found unhelpful.

There follows some description of Rust's basic data types and flow control mechanisms before the book moves on to one of the supposedly 'difficult' areas of Rust: ownership. The concept of ownership itself is neither new nor particularly difficult, but Rust's tracking of ownership of memory is a key part of its correctness checking and the language imposes rules preventing, in particular, the sharing of mutable state. The book attempts to explain these rules and the concepts behind them, but to my mind makes fairly heavy weather of it. The reader is led through a series of examples of code that doesn't compile before the – actually quite simple – rules for taking and for borrowing resources are eventually spelt out. The authors take this opportunity to explain the sorts of error messages that the compiler may generate when the rules are broken, which hardly seems necessary as the Rust compiler gives quite clear, concise, errors; I think it's something the reader could have been left discover through trial and error.

Code Critique Competition 122 (continued)

Listing 2

```
/**
  * @input A : Integer
  *  * @Output Integer
  */
int isPower(int A) {
  if(A==1) return 1;
  else
  {
    int i;
    if(A%2==0)
    {
      for(i=2; i<=sqrt(A); i=i+2)
      if(pow(i, (int)(log(A)/log(i)))==A)
      return 1;
    }
    else
    {
      for(i=3; i<=sqrt(A); i=i+2)
      if(pow(i, (int)(log(A)/log(i)))==A)
      return 1;
    }
    return 0;
  }
}
```

Listing 3

```
#include <stdio.h>
int main(void)
{
  int A = 1000;
  printf("isPower(1000) => %i\n", isPower(A));
  return 0;
}
```

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

Moving on through a description of Rust's approach to struct and tuple datatypes – which could probably, with benefit, have been lumped together with the earlier section on basic data types – we come to a short chapter on Rust's enum types and an introduction to matching. Rust's enums are not simple numbers, like those in (say) C, but are sophisticated discriminated union types. Enums are everywhere in Rust and they deserve to be covered earlier and in more detail. Matching, in Rust, is similar to that of a functional language like Haskell. I felt that this, too, deserved a chapter of its own, early on, with some real-world examples rather than the trivial ones presented here; instead there is an introduction to matching here, and a further chapter near the end of the book.

Next we're led through Rust's module system and collection types – concentrating here on string types and the issues caused by the fact that Rust's strings are Unicode stored in UTF-8 – after which comes 'Error Handling'. Rust is, unsurprisingly, very big on handling of errors, and we've had sneak previews of some of the material here in earlier chapters. I would have preferred to have had a fuller description up-front.

Next comes coverage of Rust's type traits, generic types, and lifetimes – all in the same chapter. Lifetimes are another of the notoriously 'difficult' areas of the language: the idea is simple in concept – the compiler needs to know the lifetimes of references in order to enforce its checking for dangling pointers, and often in needs some help from the programmer. This means that the language includes a special syntax for describing lifetimes and the programmer has to understand it, and when to use it. This is pretty important stuff in Rust and, to my mind, deserves more prominent treatment, earlier in the book. Traits and generics are important too, of course, but they're not related to lifetimes and not fundamental to Rust's primary goal of catching errors at compile time, and could have been treated later.

After a section on Rust's automated testing facilities the reader is led through an exercise in writing a grep-like command-line tool before being introduced to closures and iterators. This is followed by additional coverage of the cargo tool and Rust's online package repository crates.io before the reader is treated to Rust's take on smart pointers. The discussion of smart pointers and how they fit into Rust's compile-time correctness checking scheme is important and deserves to have been covered earlier in the book (but after enums and lifetimes, upon which it depends).

There then follows a chapter on 'Fearless Concurrency', which explains why Rust's refusal to allow shared access to writable state leads naturally to safe threading, and one on Rust's Object-Oriented features (traits again) and the fuller treatment of patterns and matching that we've been looking forward to for the last 12 chapters.

Finally there is a section on advanced features, which includes a discussion of 'unsafe' Rust code, which is not guaranteed by the Rust compiler to be correct (this includes code written in C or another alien language), followed by one last worked example in which a multi-threaded web server is coded in Rust. The book ends with some appendices, the last of which contains a brief description of Rust's macro language.

I didn't hate this book – it was an interesting read and I learned a lot about Rust – but you've probably gathered that I didn't entirely like it, either. I found the order in which language features were presented to be illogical – the reader is forever being asked to take a feature from a future chapter on trust while working through an example – and I felt that there was a lack of meaningful examples. Although the introduction states that the reader is expected to know another language it goes on to explain a number of quite basic programming concepts while glossing over several things that are peculiar to Rust and should have been covered in more depth. There are diagrams to show the layout in memory of Rust's strings, for example, even though they're pretty unsurprising, but no diagrams showing the layout of the smart pointer types or enums with data fields about which I had wondered while reading their descriptions. At times the book left me with more questions than it answered. The index, by the way, is terrible. I referred to it several times, and never found an entry for the thing I was trying to look up.

I felt, at the end, that I could read Rust code, but I wasn't confident that I knew how to write it. I had no feeling for the idioms of the language – perhaps in part because of the book's habit of introducing each topic by showing code that doesn't work and then fixing it.

That said: the free online version of the book is certainly accessible and economical, and although it wasn't my cup of tea, but it may be yours.

Programming Rust

By Jim Blandy and Jason Orendorff, published by O'Reilly (2018), 598 pages, ISBN: 978-1-491-92728-1

Reviewed by Daniel James

Recommended.

I read this book (PR) right after *The Rust Programming Language* (TRPL). Please be aware that I wasn't approaching PR with the same fresh innocence as I had TRPL, and although I have tried not to allow this to colour my review, I am not sure that I have succeeded completely.

This book begins, appropriately enough with a short chapter entitled 'Why Rust?'. As expected, this stresses Rust's emphasis on type-safety. Less expected was the categorization of Rust as specifically a systems programming language – I'd view it as being just as suitable for applications work. The point being made is that efficient close-to-the-metal languages like C and C++ that are typically used for systems work are significantly less typesafe than 'softer' languages like Python. As a C++ programmer, I found the comparison rankled, but the authors do have a point.

The second chapter provides a whistle-stop tour of Rust, touching on a great many language features without really explaining any. This is exactly what I complained about in TRPL, but that doesn't seem to matter here: the presentation makes it clear that the tour is merely a preview to whet the readers' appetites – like a trailer in the cinema. It's a flourish of special effects and we're not supposed to be able to deduce the whole plot from it.

Chapter 3 begins the description of Rust in earnest. The basic data types are discussed, with comments on their acceptable values and ranges. The detail here seems more complete and better arranged than in TRPL and it's all easy reading. Tuples, arrays, vectors, strings and slices are all introduced here, though not structs or (sadly) enums which are, as in TRPL, left until far too late.

The next couple of chapters discuss ownership (along with moving/taking and borrowing) and references. Lifetimes are introduced here, too, almost incidentally. These are the two 'difficult' topics, where Rust's strictness causes most problems for newcomers, and the treatment here is fairly painless but perhaps underplays their importance. Rust's smart pointer types `Box` and `Rc` are also discussed here.

Chapter 6 is entitled 'Expressions', but is actually an introduction to most of the statement types that are still to be introduced. This is not unreasonable as Rust is an expression-oriented language and most statements can be expressions. Functions are introduced, here, in passing, and there's enough of an overview of matching to allow `if` and `while` to be covered sensibly.

Chapter 7 deals with error handling. As Rust handles errors using `Result` values that are actually enums, it seems odd to talk about errors before explaining in detail what an enum is. The description here is clear enough, I think, but could have been clearer had the authors simply been able to say '`Result` is an `enum` type' and had the reader already known what an enum was.

Chapter 8 discusses modules in Rust, and introduces Rust's notion of a crate (or external library). Crates are expected to contain their own tests and documentation, so unit tests and document comments are covered here, too, as are dependencies and the role played by the cargo tool in managing them.

Chapter 9 discusses structs and introduces methods and derived traits. Traits were introduced briefly in the tour in chapter 2, but haven't been covered in any detail, so the discussion of derived traits really demands a little more explanation than is here, but what is here is clear enough. We also touch here on Rust's generic types.

Chapter 10 brings the long-awaited discussion of enums and of matching. The discussion of enums is thorough and includes a description of how the compiler arranges enum objects, whose fields may be of different sizes, in memory. The discussion of matching might have given more examples of matching of patterns against enum types … but by now we've become so familiar with seeing matches against `Result` and `Option` types, introduced ahead of time, that there isn't much new to tell.

Chapter 11 brings a discussion of Rust's traits, and how they are used to add functionality to types, and to implement generic behaviour. This discussion is carried on in the next couple of chapters in which operator overloading (which depends on traits) and utility traits are discussed. Utility traits include things like `Copy` and `Clone` (among others) that are used to add system-defined behaviours to user-defined types so that the compiler can handle them efficiently. Their treatment here is thorough and well-explained.

Chapters follow on closures, iterators, collections, strings and text, and input and output. The last touches on networking while noting that it is beyond the scope of the book. Then comes a chapter on concurrency – one of the longest in the book – which shows how Rust's type-safety prevents the sharing of mutable data between threads, and shows how communication between threads can nevertheless be safely achieved.

A chapter on macros gives a good overview of writing macros for Rust with a number of practical examples, and finally a chapter on unsafe code brings the book to a close.

PR has some 20% more pages than TRPL, and feels as though it contains about twice as much information with much clearer explanations – though of course it could just be that the material seemed clearer because the subject matter wasn't all new to me, in which case TRPL must take some of the credit! I was annoyed by a few snide digs at C++ for being less safe than Rust – yes, of course a Rust `enum` is more typesafe than a C `union` … but you don't use a C union in C++ you use something like `std::variant` – those things aside the book is readable and informative. I felt that the ordering of material wasn't optimal – in particular: Rust's enum is pervasive and needs to be explained early on – but for the most part the book doesn't use features that haven't yet been covered, so there are few surprises. The index isn't great, but I did find most things that I tried to look up, so it's better than many.

C++ Templates The Compete Guide, Second Edition

By David Vandevoorde, Nicolai M Josuttis, Douglas Gregor, published by Pearson Addison-Wesley (2018), 770 pages, http://www.tmplbook.com/

Reviewed by Paul Floyd

Recommended for advanced users. I would suggest that beginners start with something else.

I read the first edition of this book sometime back in 2003 (reviewed here https://accu.org/index.php?module=bookreviews&func=search&rid=506). A great deal has changed in the 15 years between the two editions – 3 new revisions of the C++ standard in particular. This shows up in the page count, and even visibly with the second edition being about 2cm thicker.

This being an 'upgrade' read, I thought that I would just be able to skim over the 'new bits' in just a few days. Boy, was I wide of the mark… several weeks more like it. Even though, when I look at the two chapter listings, clearly the two books have the same structure, my feeling was that the second edition was an entirely different experience. In part this is due to the large amount of extra material, and also in part a change in my perspective. Back then I thought that I knew a bit about template programming. Now I'm sure that I know little about it.

The book covers templates in depth for all C++ versions up to and including C++17, plus a bit on additions expected a bit in C++20, especially concepts. I won't list the chapters (there are 28 of them, plus 5 appendices). There are 3 parts to the book, 'Basics', 'Templates in Depth' and 'Templates and Design'. I did laugh (out loud even) at the start of part 2 when I read that part 1 ('Basics') "is sufficient to answer most questions that may arise in everyday C++". I doubt that your average C++ programmer's needs get much past chapter 2 and perhaps chapter 9 (covering instantiation and error messages).

As you would expect, everything regarding templates is covered. Function and Class Templates, Nontype Parameters, much about traits/policies and meta programming. There's also a lot on overloading and name lookup. There's considerable coverage of how templates interact with language features, which is very important if you want to understand which types will be used to instantiate your template functions (and to a lesser extent classes). There is some nitty-gritty, but this is no 'C++ Templates in 7 Days'. The last few chapters cover a few concrete examples such as Tuple and Variant template classes.

The book contains a lot of examples in the form of snippets, often in the form of a few definitions followed a list of 4 or 5 example uses with comments like "// OK T deduced as long int" or "// ERROR". I found that this bogged down the compiler in my head, and I would often spend 5 or 10 minutes trying to think through why a certain piece of code would resolve to which types. Perhaps I should have skimmed through quickly a first reading and then a more thorough second pass.

I don't know what sort of reader will benefit from this book. It is neither a tutorial/cookbook nor a straight reference book. I suppose that, as it says on the cover, 'guide' is the best way to describe it. Compiler writers (there are a few sections on parsing) and template library writers will want a copy. After that, if you already have copies of Stroustrup, Meyers and Josuttis then you will also want to get this one.

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.

View from the Chair

Bob Schmidt
chair@accu.org

ACCU's 2019–2020 term has ended, not with its usual bang (the conference), but with a whimper (courtesy of the novel coronavirus).

ACCU 2020 Conference

I know that many of you were looking forward to a week of education and socializing at ACCU 2020, as was I. As concerns about the virus grew, companies adopted no-travel policies, and the number of cancellations amongst the keynote speakers and presenters outpaced our ability to find substitutes. This led Archer-Yates Associates (AYA), in conjunction with the conference committee, to make the tough decision to cancel the conference. Given the rate at which the virus has spread, and the subsequent governmental reactions to the pandemic, cancelling the conference was absolutely the correct thing to do.

This was to be Russel Winder's final year as conference chair. It is a shame that he, and we, won't get to experience the culmination of his five years of dedication to the conference. Please join me in thanking Russel for his contribution to ACCU.

State of ACCU

2019 saw a slight decrease in paid membership, from 532 (non-corporate) members at the beginning of 2019 to 515 at the beginning of 2020. This is a lower rate of attrition than ACCU experienced in 2018. Membership numbers are flat for 2020 year-to-date. In order to try to boost membership, the committee has decided to offer a temporary, basic membership to conference attendees who were not already members. (AYA sent the offer to registrants via email.) Our hope is that a person who test drives a temporary membership will recognize the value and convert to a regular membership.

Our financial situation remains strong. ACCU finished 2019 with a £6,167 surplus. Once again, the revenues we received from AYA for the 2019 conference accounted for the surplus. This is important because we won't have a conference contribution in 2020; we expect our accounts for this year to be break-even.

Fran Buontempo and Steve Love continue to produce quality issues of *Overload* and *C Vu* (respectively), assisted by their teams of highly-capable peer reviewers. Getting content from authors continues to be a challenge.

Although we have filled several positions recently (Publicity, Study Groups, Reviews; see below), we continue to have chronically vacant positions on the committee. The Social Media and Web Editor positions have been vacant for over three years. Advertising isn't chronically vacant yet, but it is heading that way. Last year the committee adopted an incentive to fill chronically vacant positions, offering up to a year of membership in return for a year of service.

Our local groups continue to be a strong part of ACCU's outreach. We added a new local group in York this past year. Phil Nash has plans to expand the reach of our affiliated groups program. Thank you to all our local group and affiliated group coordinators.

ACCU Committee

The 2020 Annual General Meeting (AGM) was held online, via Zoom, on 28 March 2020. Seventeen members attended the meeting. The agenda and AGM pack are available to members online. As of this writing the agenda has not yet been posted.

We need volunteers to run for Chair and Secretary. Both of those positions remain vacant after the AGM. ACCU has been placed into caretaker mode, where only day-to-day business can be transacted.

We had some changes to the makeup of the ACCU committee over the past year:

- Seb Rose stepped down from the Advertising role last May, after many years in that position.
- Ian Bruntlett has been doing a wonderful job as Reviews Editor since last summer.
- Phil Nash succeeded Nigel Lester as Local Groups Coordinator.
- Guy Davidson took over as Standards Officer from Emyr Williams.

The results of elections were as follows:

- Matt Jones will continue as Membership Secretary.
- Patrick Martin is moving from Secretary to Treasurer.
- After many years of service, Rob Pauer is stepping down as Treasurer. Rob will take on the ceremonial role of Treasurer Emeritus, and remain active with the committee, facilitating a smooth transition to Patrick as the new treasurer.
- Roger Orr will continue in Publications; Ralph McArdell will continue as the At-Large member; Jim Hague will continue as Webmaster.
- Ian, Phil, and Guy were re-elected to their respective positions.

We have two members who have been co-opted to fill chronically vacant positions on the committee:

- Ricardo Rodriguez will take over as Study Groups Coordinator. Ricardo is a teacher in Miama, Florida, USA, and is studying for a career change as a software professional. Ricardo's teaching experience should serve him well as Study Groups Coordinator.
- Adeel Nadeem has volunteered to handle Publicity for ACCU. Adeel is a Software Engineering recruiter from Bristol. He studied Economics at Reading University in 2008. For the last 10 years he has dedicated his career to helping the C++ community, building software engineering teams across the UK and developing some fantastic relationships along the way. When not working, he enjoys making the most if his National Trust membership and breathing in some fresh air outside the city, or experimenting in the kitchen where he claims to make a mean curry.

Our auditors for the past year were Alan Griffiths and Deitmar Kühl. Alan filled Guy Davidson's second year, when Guy became Standards Officer. (Auditors are separate from, and independent of, the committee. Guy couldn't be an auditor and serve on the committee at the same time.) Deitmar has just completed the first year of his two-year term. During the AGM, Alan volunteered to start a new two year term.

Daniel James has produced the ePub version of our magazine every month for two years, and continues in that role. I will be continuing to participate on the committee as co-opted web editor.

Felix Petriconi will be taking over as Conference Chair starting with the 2021 conference. He has been shadowing Russel for the past year in order to make the hand over as seamless as possible.

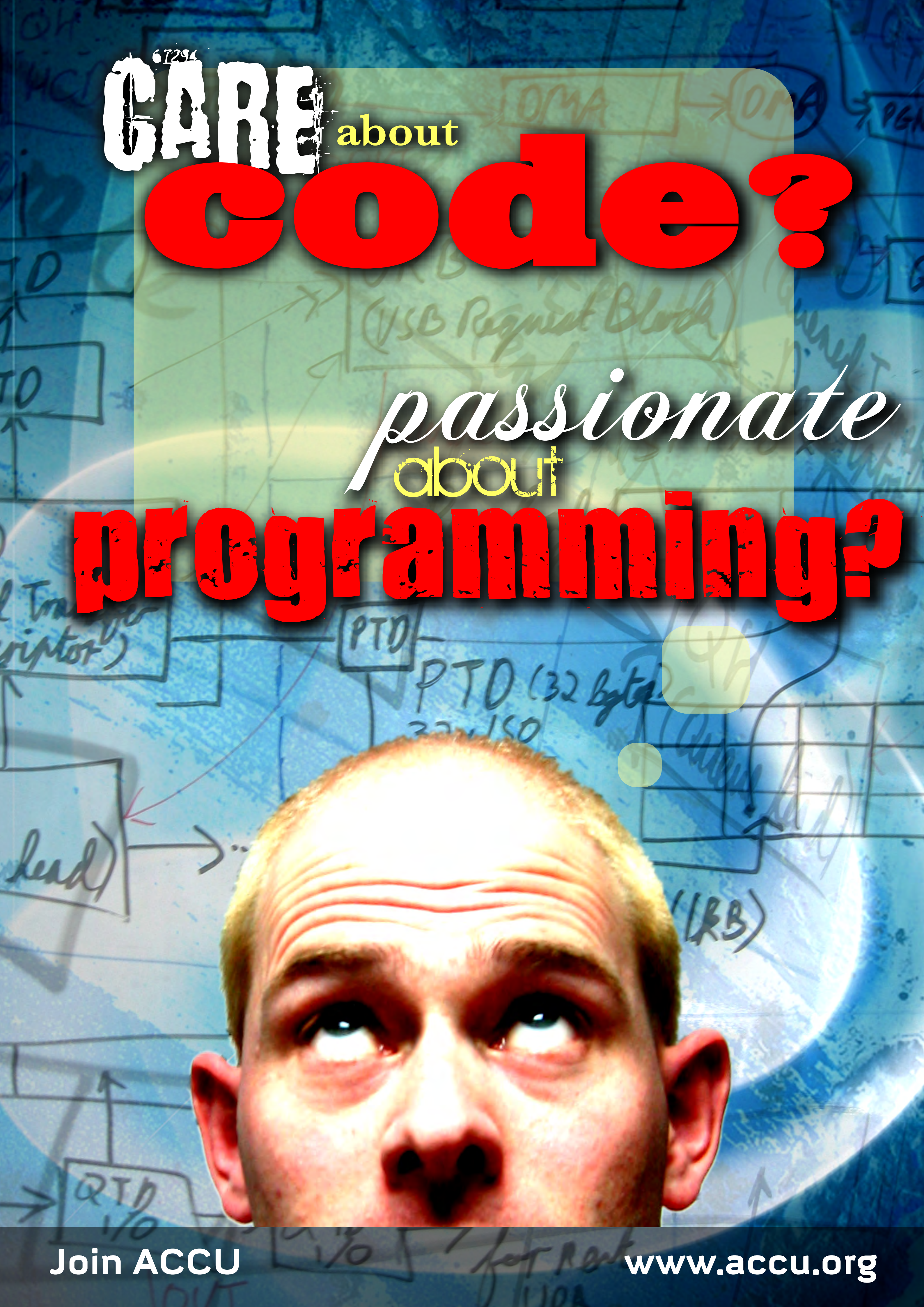
Finally, Thank You

As always my thanks go out to all of ACCU's volunteers. (I hope I have named everyone in the paragraphs above.) You have made the past four years a wonderful experience for me.

A special thank you goes out to Rob Pauer, our Treasurer for many years. Rob has been retired from his 'real' job for several years, and now is able to go into full retirement. Rob has the distinction of having been an active member of ACCU for the longest period of time (having joined a year before Francis Glassborow). Rob assures us that he will continue to be active in the organization, for which we should all be thankful.

This is my 24th, and final, *View as Chair*. I hope you have found my ramblings informative, if not entertaining. Thanks for reading.

Portions of this article appeared in the ACCU 2020 AGM Pack.


CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org

intel Software

PURE CODE ADRENALINE

Accelerate applications for enterprise, cloud, HPC, and AI.

Develop high-performance parallel code and accelerate workloads across enterprise, cloud, high-performance computing (HPC), and AI applications.

Amp up your code: **www.qbssoftware.com**