the magazine of the accu

www.accu.org

{cvu}

Volume 30 • Issue 6 • January 2019 • £4.50

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe

{cvu}

Volume 30 Issue 6
January 2019
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Silas S. Brown, Pete Cordell, Francis Glassborow, Pete Goodliffe, Chris Oldwood, Roger Orr, Alison Peck, Emyr Williams

ACCU Chair
Bob Schmidt
chair@accu.org

ACCU Secretary
Patrick Martin
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

Usefulness versus complexity

Computers, and the software that drives them, are all becoming increasingly sophisticated. They have come a long way from the days of punch-cards and room-sized machines that could perform – admittedly quite complex – mathematical functions *almost* as quickly as their human operators. Today we have pocket-sized computing power that far out-strips the abilities of early computing machinery, that has access to a vast array of information and computing power over the Internet.

Modern computers and computer programs perform tasks that no human could achieve (even relatively simple tasks like rendering graphics in a game, if done by a real person, would make the game un-playably slow!). This sophistication comes with a price, of course. The hardware and software used to do those things requires deep, specialized knowledge in a multitude of disciplines, probably beyond the abilities of any one person. An extreme example of this is the use of 'AI' techniques in almost every area of technology. Yes, the quotes are deliberate, because most of those techniques don't really represent 'intelligence'. Nevertheless, it's an area where real understanding of what is going on 'under the hood' is held by a very small number of people.

Most of the areas for which AI has been lauded is in fields that *people* are instinctively good at: identifying faces (and other objects) in photos, recognizing and parsing natural speech in real-time. It's undoubtedly true that computer automation (let's call it what it is) can indeed do some of the work that up until now has been done by human beings. It is to be hoped that this results in a positive outcome all-round, where everyone benefits. On the other hand, I don't think we're under any immediate threat from as widespread a take-over of all human affairs that some sections of the media seem to want us to believe.

It being the year of Blade Runner, it's not flying cars that are conspicuous by their absence, but the intelligent robots that are nearly indistinguishable from people.

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.

FEATURES

DIALOGUE

REGULARS

SUBMISSION DATES

C Vu 31.1: 1st February 2019
C Vu 31.2: 1st April 2019

Overload 149: 1st March 2019
Overload 150: 1st May 2019

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from C Vu without written permission from the copyright holder.

Effective Software Testing

Pete Goodliffe describes a healthy software test regimen.

Fights would not last if only one side was wrong.
~ François de la Rochefoucauld

The mid-twentieth-century philosophers and purveyors of jaunty, tuneful hair, The Beatles, told us *all you need is love*. They emphasised the point: *love is all you need*. Love; that's it. Literally. Nothing else.

It's incredible how long a career they had given that they didn't need to eat or drink.

In our working relationships with other inhabitants of the software factory, we would definitely benefit from more of that sentiment. A little more love might lead to a lot better code! Programming in the real world is an interpersonal endeavour, and so is inevitably bound up in relationship issues, politics, and friction from our development processes.

We work closely with many people. Sometimes in stressful scenarios.

It is not healthy for our working relationships, nor for the consequent quality of our software, if our teams are not working smoothly together. But many teams suffer these kinds of problem.

As a tribe of developers, one of our rockier relationships is with the QA enclave; largely because we interact with them very closely, often at the most stressful points in the development process. In the rush to ship software before a deadline, we try to kick the software soccer ball past the testing goalkeepers.

So let's look at that relationship now. We'll see why it's fraught, and why it must not be.

What is QA good for?

To some it's obvious what they do. To others it's a mystery. The 'QA' department (that is, *Quality Assurance*) exists to ensure that your project ships a software product of sufficient quality. They are a necessary and vital part of the construction process.

What does this entail? The most obvious and practical answer is that they have to test the living daylights out of whatever the developers create in order to ensure:

- That it matches the specification and requirements – that every feature that should be implemented has been implemented.
- That the software works correctly on all platforms – that is, it works on all OSes, on all versions of those OSes, on all hardware platforms, and on all supported configurations (e.g., meeting minimum memory requirements, minimum processor speeds, network bandwidth, etc.).
- That no faults have been introduced in the latest build – the new features don't break any other behaviour, and no regressions (the reintroduction of previous bad behaviour) have been introduced.

Their name is 'QA,' not just 'the testing department,' and for a reason. Their role is not just pushing buttons like robots; it's baking quality into the product.

To do this, QA must be deeply involved throughout, not just a final adjunct to the development process.

- They have a hand in the specification of the software, to understand – and shape – what will be built.
- They contribute to design and construction, to ensure that what's built will be testable.
- They are involved heavily in the testing phase, naturally.
- And also in the final physical release: they ensure that what was tested is what is actually released and deployed.

Software development: shovelling manure

In unenlightened workplaces, the development process is modelled as a huge pipe: conveying raw materials pumped in the top, through various processes, until perfectly formed software gushes (well, perhaps dribbles) out the end. The process goes something like this:

- Someone (perhaps a *business analyst* or *product manager*) pours some requirements into the mouth of the pipe.
- They flow through architects and designers, where they turn into specifications and pretty diagrams (or good intentions, and smoke and mirrors).
- This flows through the programmers (where the *real* work gets done, naturally), and turns into executable code.
- Then it flows into QA. Where it hits a blockage as the 'perfectly formed' software magically turns into a non-functioning disaster. These people *break* the code!
- *Eventually* the developers push hard enough down the pipe to break this blockage, and the software finally flows out of the far end of the pipe.

In the fouler development environments, this pipe more closely resembles a sewer. QA feels like the developers are pumping raw sewage down to them, rather than handing them a thoughtfully gift-wrapped present. They feel they are being dumped on, rather than worked with.

Is software development really this linear? Do our processes really work like this simple pipeline (regardless of how pure the contents)?

No. They don't.

The pipe is an interesting first approximation (after all, you can't test code that hasn't been written yet), but far too simplistic a model for real development. The linear pipeline view is a logical corollary of our industry's long fascination with the flawed *waterfall* development methodology. (This model is often attributed to Winston Royce. Although he wrote about it in the 1970s, it was as an illustration of a *flawed* development process, not a laudable one. [1])

It is wrong to view software development as a linear process.

However, this view of the development process does explain why the development team's interaction with the QA team isn't as smooth as it should be. Our processes and models of interaction are too often shaped by the flawed sewage-based development metaphor. We should be in constant communication, rather that just throwing them software towards the end of the development effort.

A false dichotomy

Our inter-team interactions are hindered because they are just that: interactions between *separate* teams. The QA people are considered a

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

different tribe, distinct from the 'important' developers. This bogus, partitioned vision of the development organisation inevitably leads to problems.

When QA and development are seen as separate steps, as separate activities, and therefore as very separate teams, an artificial rivalry and disconnect can too easily grow. This is reinforced physically by our building artificial silos between testers and developers. For example:

- The two teams have different managers and different reporting lines of responsibility.
- The teams are not colocated, and have very different desk locations (I've seen QA in separate desk clusters, on different floors, in different buildings, and even – in an extremely silly case – on another continent).
- There are different team structures, recruiting policies, and expected turnover of staff. Developers are valued resources, whereas testers are seen as replaceable cheap mercenaries.
- And most pernicious: the teams have very different incentives to complete tasks. For example, the developers are working with the promise of bonus pay if they complete a job quickly, but the testers are not. In this case, the developers rush to write the code (probably *badly*, because they're hurrying). They then get very cross when the QA guys won't sanction it for a timely release.

We reinforce this chasm with stereotypes: developers *create*, testers *break*.

There *is* an element of truth there. There are different activities and different skills required in both camps. But they are not logically separate, silo-ed activities. Testers don't find software faults to just break things and cause merry hell for developers. They do it to improve the final product.

They are there to bake in quality. That's the Q in QA. And they can only do this effectively in tandem with developers.

Beware of fostering an artificial separation between development and testing efforts.

By separating these activities, we breed rivalry and discord. Often, development processes pit QA as the *bad guy* against the developer *hero*. Testers are pictured standing in the doorway, blocking a plucky software release getting out. It's as if they are *unreasonably* finding faults in our software. They are nit-picking over minutiae.

It's almost as if they're running into the forests, catching wild bugs, and then injecting them into our otherwise perfect code.

Does that sound silly?

Of course it does when you read it here, but it's easy to start thinking that way: 'The code is fine; those guys just don't know how to use it.' Or: 'They've been working far too long to find such a basic bug; they really don't know what they're doing.' Software development is not a battle. (Well, it shouldn't be.) We're all on the same side.

Fix the team to fix the code

Conway's famous law [2] describes how an organisation's structure – and specifically the lines of communication between its teams – dictates software structure. It is popularly paraphrased as: "If you have four teams writing a compiler, it'll be a four-pass compiler." Experience shows this to be pretty accurate. In the same way that team *structure* affects the code, so does the *health* of the interactions within the software.

Unhealthy team interactions result in unhealthy code.

We can improve the quality of our software, and the likelihood of producing a great release, by addressing these health issues: by improving the relationship between developers and QA. Working *together* rather than waging war. *Love*, remember, *is all you need.*

This is a general principle and applies far more broadly than just between developers and QA. Cross-functional teams in all respects are very helpful.

This comes down to the way we interact and work with QA. We should not treat them as puppets whose strings we pull, or to whom we throw ropey software to test. Instead, we treat them as coworkers. Developers *must* have good rapport with QA: a friendship and camaraderie.

Let's look at the practical ways we can work better with these inhabitants of the QA kingdom. We'll do this by looking at the major places that developers interact with QA.

But we have unit tests!

We are conscientious coders. We want to make rock-solid software. We want to craft great lines of code with a coherent design, that contribute to an awesome product.

That's what we do.

So we employ development practices that ensure our code is as good as possible. We review, we pair, we inspect. And we *test*. We write automated unit tests.

We have tests! And they pass! The software must be good. Mustn't it?

Even with unit tests coming out of our ears, we still can't guarantee that our software is perfect. The code might operate as the developers intended, with green test lights all the way. But that may not reflect what the software is *supposed* to do.

The tests may show that all input the developers envisioned was handled correctly. But that may not be what the user will actually do. Not all of the use cases (and *abuse cases*) of the software have been considered up front. It is hard to consider all such cases – software is a mightily complex thing. Thinking about this is exactly what the QA people are great at.

Because of this, a rigorous testing and QA process is still a vital part of a software development process, even if we have comprehensive unit tests in place. Unit tests act as our responsible actions to prove that the code is good enough before we hand it over to the testers to work on.

Releasing a build to QA

We know that the development process isn't a straight line; it's not a simple pipeline. We develop iteratively and release incremental improvements: either a new feature that needs validation or a fixed bug that should be validated. It's a cycle that we go around many times. Over the course of the construction process, we will create numerous builds that will be sent to QA.

So we need a smooth build and hand-off process.

This is vital: the hand-off of our code must be flawless – the code must be responsibly created and thoughtfully distributed. Anything less is an insult to our QA colleagues.

We must build with the right attitude: giving something to QA is not the act of throwing some dog-eared code, built on any old machine, over the fence for them. It's not a slapdash or slipshod act.

Also, remember that this is not a battle: we don't aim to *slip* a release past QA, deftly avoiding their defence. Our work must be high quality, and our fixes correct. Don't cover over the *symptoms* of the more obvious bugs and hope they'll not have time to notice the underlying evils in the software.

Rather, we must do everything we can to ensure that we provide QA with something worthy of their time and effort. We must avoid any silly errors or frustrating sidetracks. Not to do so shows a lack of respect to them.

Not creating a QA build thoughtfully and carefully shows a lack of respect to the testers.

This means that you should follow the guidelines covered in the following sections.

Test your work first

Prior to creating a release build, the developers should have done as good a job as possible to prove that it is correct. They should have tested the work they've done beforehand. Naturally, this is best achieved with a comprehensive suite of regularly run unit tests. This helps catch any behavioural *regressions* (reoccurrences of previous errors). Automated tests can weed out silly mistakes and embarrassing errors that would waste the tester's time and prevent them from finding more important issues.

With or without unit tests, the developers *must* have actually tried the new functionality, and satisfied themselves that it works as well as is required. This sounds obvious, but all too often changes or fixes that should 'just work' get released, and cause embarrassing problems. Or a developer sees the code working in a simple case, considers it adequate for release, and doesn't even think about the myriad ways it could fail or be mis-used.

Of course, running a suite of unit tests is only as effective as the quality of those tests. Developers take full responsibility for this. The test set should be thorough and representative. Whenever a fault is reported from QA, demonstrative unit tests should be added to ensure that those faults don't reappear after repair.

Release with intent

When a new version is being released to QA, the developer must make it clear exactly *how* it is expected to work. Don't produce a build and just say, "See how this one works."

Describe clearly exactly what new functionality is and isn't implemented: exactly what is known to work and what is not. Without this information you cannot direct what testing is required. You will waste the testers' time. You communicate this in *release notes*.

It's important to draw up a set of good, clear *release notes*. Bundle them with the build in an unambiguous way (e.g., in the deployment file, or with a filename that matches the installer). The build must be given a (unique) version number (perhaps with an incrementing *build number* for each release). The release notes should be versioned with this same number.

For each release, the release notes should clearly state what has changed and what areas require greater testing effort.

More haste, less speed

Never rush out a build, no matter how compelling it seems. The pressure to do this is greatest as a deadline looms, but it's also tempting to sneak a build out before leaving the office for the evening. Rushing work like this encourages you to cut corners, not check everything thoroughly, or pay careful attention to what you're doing. It's just too easy. And it's not the right way to give a release to QA. Don't do it.

If you feel like a school kid desperately trying to rush your homework and get 'something' in on time, in the full knowledge that the teacher will be annoyed and make you do it again, then something is wrong! Stop. And think.

Never rush the creation of a build. You will make mistakes.

Some products have more complex testing requirements than others. Only kick off an expensive test run across platforms or OSes if you think it's worthwhile, when an agreed number of features and fixes have been implemented.

Automate

Automation of manual steps always removes the potential for human error. So automate your build or release process as much as possible. If you can create a single script that automatically checks out the code, builds it, runs all unit tests, creates installers or deploys on a testing server, and uploads the build with its release notes, then you remove the potential for human error for a number of steps. Avoiding human error with automation helps to create releases that install properly each time and do not contain any regressions. The QA guys will love you for that.

Respect

The delivery of code into QA is the act of producing something stable and worthy of potential release, not the act of chucking the latest untested build at QA. Don't throw a code grenade over the fence, or pump raw software sewage at them.

On getting a fault report

We give the test guys a build. It's our best effort yet, and we're proud of it. They play with it. Then they find faults. Don't act surprised. You knew it was going to happen.

Testing will only reveal problems that software developers added to the system (by omission or commission). If they find a fault, it was *your fault!*

On finding a bug, they lodge a *fault report*: a trackable report of the problem. This report can be prioritised, managed, and once fixed, checked for later regression.

It is their responsibility to provide accurate, reliable fault reports, and to send them through in a structured and orderly way

It is *their* responsibility to provide accurate, reliable fault reports, and to send them through in a structured and orderly way – using a good bug tracking system, for example. But faults can be maintained in a spreadsheet, or even by placing stories in a work backlog. (I've seen all these work.) As long as there's a clear system in place that records and announces changes to the state of a fault report.

So how do we respond to a fault report?

First, remember that QA *isn't* there to prove that you're an idiot and make you look bad. The fault report isn't a personal slight. So don't take it personally.

Don't take fault reports personally. They are not a personal insult!

Our 'professional' response is, "Thanks, I'll look into it." Just be glad it was QA who they found it, and not a customer. You *are* allowed to feel disappointed that a bug slipped through your net.

You should be worried if you are swamped by so many fault reports that you don't know where to start – this is a sign that something very fundamental is wrong and needs addressing. If you're in this kind of situation, it's easy to resent each new report that comes in.

Of course, we don't leap onto every single fault as soon as it is reported. Unless it is a trivial problem with a super-simple fix, there are almost certainly more important problems to address first. We must work in collaboration with all the project stakeholders (managers, product specialists, customers, etc.) to agree which are the most pressing issues to spend our time on.

Perhaps the fault report is ambiguous, unclear, or needs more information. If this is the case, work *with* the reporter to clarify the issues so you can both understand the problem fully, can reproduce it reliably, and know when it has been closed.

QA can only uncover bugs from development, even if it's not a fault that *you* were the direct author of. Perhaps it stems from a design decision that you had no control over. Or perhaps it lurks in a section of code that you didn't write. But it is a healthy and professional attitude to take responsibility for the *whole product*, not just your little part of the codebase.

Our differences make us stronger

Whenever you're in conflict with someone, there is one factor that can make the difference between damaging your relationship and deepening it. That factor is attitude.
~ William James (philosopher and psychologist)

Effective working relationships stem from the right developer attitudes.

When we're working with QA engineers, we must understand and exploit our differences:

- Testers are very different from developers. Developers often lack the correct mind-set to test effectively. It requires a particular way of looking at software, particular skills and peccadilloes to do well. We must respect the QA team for these skills – skills that are essential if we want to produce high-quality software.
- Testers are inclined to think more like a user than a computer; they can give valuable feedback on perceived product quality, not just on correctness. Listen to their opinions and value them.
- When a developer works on a feature, the natural instinct is to focus on the *happy path* – on how the code works when everything goes well (when all input is valid, when the system is working fully with maximum CPU, no memory or disk space issues, and every system call works perfectly).

  It's easy to overlook the many ways that software can be used incorrectly, or to overlook whole classes of invalid input. We are wired to consider our code through these natural cognitive biases. Testers tend not to be straightjacketed by such biases.
- Never succumb to the fallacy that QA testers are just 'failed devs'. There is a common misconception that they are somehow less intelligent, or less able. This is a damaging point of view and must be avoided.

Cultivate a healthy respect for the QA team. Enjoy working with them to craft excellent software.

Pieces of the puzzle

We need to see testing *not* as the 'last activity' in a classic waterfall model; development just doesn't work like that. Once you get 90% of the way through a waterfall development process into testing, you will likely discover that *another* 90% of effort is required to complete the project. You cannot predict how long testing will take, especially when you start it far too late in the process.

Just as code benefits from a test-first approach, so does the entire development process. Work with the QA department and get their input early on to help make your specifications verifiable, ensure their expertise feeds into product design, and that they will agree that the software is maximally testable before you even write a line of code.

The QA team is not the sole owner of, nor the gatekeeper of 'quality.' It is everyone's responsibility.

To build quality into our software and to ensure that we work well together, all developers should understand the QA process and appreciate its intricate details.

Remember that QA is part of the *same* team; they are not a rival faction. We need to foster a holistic approach, and maintain healthy relationships to grow healthy software. *All we need is love*. ■

Questions

- How close do you think your working relationship with your QA colleagues is? Should it be better? If so, what steps can *you* take to improve it?
- What is the biggest impediment to software quality in your development organisation? What is required to fix this?
- Ask the QA team what would help them most.
- Who is responsible for the 'quality' of your software? Who gets the 'blame' when things go wrong? How healthy is this?
- How good do you think your testing skills are? How methodically do you test a piece of code you're working on before you check in or hand off?

References

[1] Winston Royce (1970) 'Managing the Development of Large Software Systems', *Proceedings of IEEE WESCON* 26:1–9.
[2] Melvin E. Conway (1968) 'How Do Committees Invent?' *Datamation* 14:5 : 28–31.

JOIN ACCU

You've read the magazine. Now join the association dedicated to improving your coding skills.

ACCU is a worldwide non-profit organisation run by programmers for programmers.

Join ACCU to receive our bi-monthly publications *C Vu* and *Overload*. You'll also get massive discounts at the ACCU developers' conference, access to mentored developers projects, discussion forums, and the chance to participate in the organisation.

What are you waiting for?

How to join
Go to www.accu.org and click on Join ACCU

Membership types
Basic personal membership
Full personal membership
Corporate membership
Student membership

professionalism in programming
www.accu.org

C++ Tagged Reference Types

Pete Cordell proposes an extension to C++ move syntax.

I'm a big fan of C++ move semantics. They are a significant development that not only improves C++, but also moves computer programming language concepts forward. But a question raised recently on the ACCU General mailing list makes me wonder if they are still a 'work in progress'.

The question asked was, when is it best to use l-value refs, r-value refs or pass-by-value to pass non-trivial parameters to functions? Even amongst the experts of ACCU, there didn't seem much consensus. And there seemed no resolution on how to pass potentially moveable types through a derived class constructor on to a base class constructor.

To me, if there is no clear consensus among ACCU members on the best way to do this, then it seems too complicated for the average C++ programmer like myself. This is a bad situation for C++. This led me to idly musing whether something better could be achieved.

A key observation is that passing parameters as l-value refs, r-value refs or values is essentially an optimisation problem, and conventional wisdom is to let the compiler do low level optimisation. Therefore, would it be possible for the compiler to handle this situation too?

The core of the problem is that a function (or class) wants to end up with its own copy of a value, such as a string or more complex data structure, passed in as a parameter. If the input parameter is an l-value ref, then the function has no choice but to do a copy operation. But if the parameter is an r-value ref, then the most efficient operation is typically a move.

Outside of templates, a programmer would have to implement two functions to cater for both these situations. Both of which are likely to have almost identical code. This is not desirable.

I could wish for a syntax that allowed a function to declare that 'this parameter can be either a l-value ref or an r-value ref' and the compiler would be responsible for generating functions for both the l-value ref and an r-value ref variants. So, using a triple ampersand (`&&&`) to denote such a function parameter, given a function definition of:

```
int func( Foo const &&& f ) {...}
```

the compiler would generate code for:

```
int func( Foo const & f ) {...}
int func( Foo && f ) {...}
```

where the `...` would be identical C++ code in both cases.

This is certainly an option, but it doesn't readily cater for the more general case where a function needs to take a number of such parameters, such as:

```
int func( Foo const &&& f, Bar const &&& b,
  Dee const &&& d ) {...}
```

A definition like this would require 8 functions to be auto written. Feasible, but not ideal.

An alternative in such as case is to move away from having a compile-time solution and adopt a run-time one instead. This would require the generated reference to include whether it is an l-value ref or an r-value ref. In C++ terms it might look something like:

```
template< class T >
struct tagged_ref {
  T & ref;
  enum { lvalue, rvalue } form;
}
```

But rather than be an STL type, it would be built into the compiler, and the compiler would be able to optimise its format and usage as necessary. For example, if found to be more efficient, the reference part could be passed to a function in a register and the 'form' part on the stack. (Or even, for simple functions that don't have multiple parameters, the compiler could use the approach of auto-generating both l-value and r-value reference forms of the function, as previously mentioned.)

I'll call the type a 'tagged reference'.

Converting a tagged reference to a const l-value reference would be trivial. This could happen when using the tagged reference in an expression or passing it to a function parameter that has an l-value reference signature.

Assigning a tagged reference to another variable becomes more interesting. The generated code would have to look at the tagged reference form and decide whether a copy or move should be done. While the program code might look like:

```
void func( Foo const &&& f ) {
  a = f;
}
```

The code generated by the compiler might look more like:

```
void func( Foo const &&& f ) {
  if( f.form == tagged_type<Foo>::lvalue )
    a = f;
  else
    a = std::move( f );
}
```

Note that once the tagged reference has been assigned to another variable, its value may have been destroyed by a move operation. To avoid problems, the tagged reference variable shouldn't be used again after that point. It needs to effectively go out of scope:

```
void func( Foo const &&& f ) {
  a = f; // Fine, but could be a copy or a move
  b = f; // Error: f is no longer in scope.
}        // Use b = a
```

The same applies to passing it to another function as a parameter that is a tagged reference or r-value ref. (For this reason, assigning a tagged reference to another variable, or passing it to a tagged reference or r-value ref function parameter can't be done inside a loop. Another little thing for a compiler to look out for!)

Other than that, compiler support for a tagged reference type looks like a reasonably simple feature to support. Programmers would still be able to use the separate forms if they needed utmost efficiency.

In conclusion, tagged reference types could significantly simplify the burden on programmers of making the most of move semantics in a consistent, correct and efficient way. The compiler would automagically by able to 'do the right thing'. Source code size would be reduced by de-duplication, easier to understand and gain all the other benefits of DRY code. All of which is in line with many of the other improvements made to C++ in recent years. Move semantics could then become 'magic move semantics'. ■

(In my next article: The **`const?`** keyword, wherein the compiler generates two versions of a function, one with the **`const?`** keyword replaced with **`const`** and the other with the empty string.)

PETE CORDELL

Pete Cordell started with V = IR many decades ago and has been slowly working up the stack ever since. Pete runs his own company, selling tools to make using XML in C++ easier. Pete can be reached at accu@codalogic.com.

When Will Python 2 End?

Silas Brown explains why the days of Python 2, while numbered, may be longer than you think.

The last major release of Python 2 (i.e. Python 2.7) has been with us since 2010, and since then it has been a considerably stable language—more so than Python 3—so I have preferred Python 2 for production use, and not just because I occasionally want to run code on an old device for which there does not exist a Python 3 runtime (although there's that as well).

As Python includes rather a lot of libraries that you might be using to handle untrusted data (such as data you receive from the Internet), it would be good if any security issues yet to be discovered in the runtime or libraries are fixed when they are found, before the associated exploit code ends up in 'script kiddie' arsenal. While it seems increasingly unlikely that a stable library that has been in the public eye for years with no new features added would still have undiscovered security bugs of any serious consequence, it would certainly be more reassuring to think that people who Know What They're Doing™ are still looking after it just in case, so let's review the dates at which various organizations plan to stop support for Python 2.

The Python Software Foundation has announced end of support on 1st January 2020. But the Debian GNU/Linux distribution (which has a good security team) have already decided to ship Python 2 with Debian 10, which is currently aiming for a mid-2019 release, meaning the Debian security team will most likely support it until 2022 and the Debian long-term support team (which is not the security team but does include people who know about security) until 2024. The more-commercial Ubuntu distribution (which is based on Debian) has included Python 2 in their 18.04 'long-term support' release, which, they announced in November 2018, will be supported for 10 years (double the usual 5 years of previous long-term support releases), and have therefore committed to support Python 2 until 2028. Also Red Hat Enterprise Linux 7 includes Python 2 and has committed to support until 2024. (Both companies have decided to drop Python 2 in their following long-term versions: Ubuntu 20.04 and RHEL 8.) Therefore the timetable is currently looking like this:

- 2020: upstream support ends on January 1
- 2022: likely end of support from Debian's main security team
- 2024: end of support from Red Hat + likely end of Debian's long-term support
- 2028: end of support from Ubuntu

But if you want to run Python 2 through to 2028, you had better have control over which Linux distribution is on your servers. Just because Ubuntu 18.04 has support planned until 2028, won't necessarily stop your local sysadmin from updating to Ubuntu 20.04 in mid-2020 and saying 'sorry that means our Python 2 interpreter has gone' and you'll have to persuade them to install a package which Canonical's update tool labels as 'unsupported' even though it is probably the exact same package that is still supported by 18.04 (plus is likely to have support from Debian till 2024: Canonical seems to refer to Debian support as 'community support' and implies it's somehow less reliable than Canonical's commercial support, although this is not necessarily the case as Debian has very high standards). On the plus side, Ubuntu's do-release-upgrade script currently defaults to keeping these 'no longer supported' packages, so if the local sysadmin doesn't do something different then you might still get Python 2.

It's probably a good idea to call 'python2' explicitly rather than rely on the distribution mapping the 'python' command to 'python2', as some distributions are already discussing making the 'python' command mean Python 3, and/or removing it altogether so scripts must specify a version.

Compiling Python 2 in your home directory is doable: it's

```
./configure --prefix=/path/goes/here && make &&
make install
```

and it takes over 120 megabytes by default (although 95M of this is libraries you might be able to thin down if you don't use all of them). Or you could use a home server (the Raspberry Pi's 'Raspbian' distribution is based on Debian and should hopefully continue to contain Python 2 as long as Debian does, plus if you do need to start installing things in your home directory then you are likely to have far more space to do so on a home server) but bandwidth might be an issue for some applications.

The most compatible alternative implementation of Python 2 appears to be PyPy, and it seems likely that PyPy's support for Python 2 syntax will continue beyond 2020 although I have not found any clear statement about this. But C-based extension library modules like 'pycurl' tend not to work on PyPy (at least not without extensive changes) although 'ctypes' is supported. So whether PyPy is a feasible way forward depends on your project. There is also Google's apparently short-lived effort to write a transpiler from Python 2 to Go (the most active fork appears to be at https://github.com/grumpyhome) but it's anyone's guess whether this will be production-ready by 2020, and it is unlikely to support C extension modules at all (even ctypes).

Then there is Cython (you could try 'cython -2') but its support for Python 2 syntax is not complete, for example you can't currently unpack tuples in function parameter lists and **`unichr`** is missing. If you do get it to work, you will end up with an automatically-generated C program that calls into Python's mechanism for C extension modules: you'll need a correctly set-up installation of Python's developer packages to compile this, but it can be from Python 3 even if your original source file was Python 2. However, once again this is a method that will not work with all projects. Continuing to run the stock implementation of Python 2 is the only method that's 'sure' to work.

And then there is the option of trying to port your code to Python 3. Python's '2to3' tool is supposed to help with this, but it doesn't do everything: urllib-related libraries, for example, have changed in Python 3 in more ways than the 2to3 tool's simple renaming will handle, so you can't just run 2to3 and expect your code to work: you need to have full test coverage and/or inspect it all very carefully! This is the reason why many codebases have been slow to migrate off Python 2 in the first place. I suspect that developers coerced into going through an entire codebase with a fine-toothed comb to get it out of Python 2 might decide to use that effort in 'escaping' from the Python ecosystem altogether and port it to a more stable programming language of their choice.

Meanwhile it is possible to continue using Python 2.7 (in your home directory if necessary) and be reasonably confident of third-party security support until mid-2028, after which you'll be relying on 18 years of peer review to have done their job, and/or look out for vulnerability announcements yourself and take whatever action needed to make them irrelevant to your use-case. That might be hard to explain to corporate management though. ■

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

An Interview Icebreaker

Chris Oldwood tries a novel approach to open up the interview conversation.

Kicking off an interview can be hard. We all know what it's like to be on the other end of the phone line or the other side of the desk waiting to be fed questions that might be designed to trip us up or check out how much we know about the esoteric and dark corners of our technology stack. Being in fear of this is not a good way to get a candidate to be comfortable and therefore to stand a good chance of performing somewhat near how they would on the job.

Telephone interviews are a common early screening technique, and in my experience it's an all too common scenario where interviewers cut-to-the-chase and launch into a series of gnarly questions which often do little to either prepare you for the job in question, or they hope to filter you out as quickly as possible so they can get back to their day job.

This kind of approach has always bugged me and so over the last few years I have been trying to find better ways to ease potential candidates into the interview process so that hopefully they're more comfortable and eager to chat about what makes them tick. One such icebreaker that I've developed has turned out to be quite useful, not least because what started out purely as a bit of fun has on occasion provided some interesting insights that became useful with candidates later in the overall process.

The technique

The basic premise of the icebreaker is to present the candidate with a number of somewhat silly and seemingly pointless comparisons and ask them to (arbitrarily) choose one. Even in just a few minutes it's quite easy to rattle through a couple of dozen, which, if chosen reasonably well should bring a smile to candidate's face and maybe even a 'knowing glance' or a slightly quizzical 'hmmm'.

The outcome is largely intended to get the candidate talking comfortably and therefore I usually take a fair bit of effort to convince them that this is all just a bit of harmless fun, and that the answers are by-and-large utterly meaningless. Some candidates are a little more suspicious than others and so I always make sure I start with the more trivial questions to make sure they quickly get the hang of it and realise I'm not going to suddenly ask them to estimate the number of lampposts in Liverpool or the inner workings of a modern garbage collector. (I also steer clear of the obvious choices such as whitespace characters or text editors.)

Hence my opening few questions generally go like this:

- Google or Bing?
- Android or iPhone?
- Laptop or desktop?
- Twitter or Facebook?
- Book or blog?
- Chrome or Firefox?
- Time or space?

As you can see there is clearly no right or wrong answer, they are purely a matter of opinion (unless it involves Internet Explorer or Visual SourceSafe). Some candidates might initially try to give you some justification, e.g. why Bing is so much better these days, and therefore you may have to remind them that it's really just a bit of fun. Consequently I have a number of very silly comparisons that I like to throw in if I feel something even more light-hearted would help with the defrosting process, for example:

- Daily stand-up or daily sit-down?
- No SQL or no transactions?
- Jira or Post-It notes?
- BDD or TDD?

My original intention was to throw some less random, but perhaps also still apparently arbitrary choices in just to see what the reaction was. I was a little concerned that if I picked a specific technology or process that was in use on the actual project then candidates might feel obliged to answer with that choice or they might feel I'm trying to catch them out in some way. Hence these were some of the slightly less arbitrary choices that I initially began to mix in:

- Whiteboard or pen & paper?
- Scrum or Kanban?
- NUnit or MSTest?
- Mock or stub?
- SOAP or REST?

While this was still mostly a bit of fun, I wanted to see if I could come up with some more coding specific choices that I could weave in later on that might give a suitable candidate something a little more meaty to chew on. I realised that this was potentially moving away from my original goal but given the way candidates had been responding to the exercise it felt like it was worth a punt. These were some of the choices I began to add (mostly for C# senior developer roles):

- Interface or abstract base class?
- Tuple or struct?
- Unit test or acceptance test?
- Inheritance or composition?
- Zero element array or Enumerable.Empty?
- Null string reference or empty string value?
- ToArray() or ToList()?
- Test-before or test-after?
- Build script or build service?
- Feature branch or feature toggle?
- Lambda or closure?

These were all pretty well received, although naturally some candidates had to be reassured occasionally that the answer to every single question was really 'it depends'.

I always liked to finish off with a couple of silly ones just to make sure we ended the exercise on the right note. I would also check their 'temperature' by asking them if they enjoyed the exercise and whether they felt 'warmed-up' and ready for something a little more 'deep'. (The rest of the telephone interviews were longer questions aimed at exploring how they solved particular kinds of problems, e.g. parallel data loads, cache construction, monitoring, etc.)

CHRIS OLDWOOD

Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise-grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

Retrospective

When I started out this was intended to just be an icebreaker, I never for a moment believed that anyone's preference for Bing over Google, or watching tutorial videos instead of reading books was going to be an indication of a programmer who would be unsuitable for a role. And I still don't. However, perhaps unsurprisingly, as the set of questions grew some minor tell-tale signs did appear, which, by themselves, were not something you'd make a hiring decision on but maybe did indicate areas that you might want to focus on in the rest of the interview if there were no other pressing concerns.

For example, terminology was one aspect that came up a few times. While someone starting out their career in programming might not have been introduced to terms like 'tuples' (however you choose to pronounce it) or 'feature toggles' I am more surprised when a senior-level programmer is unaware of them. And, while they should be able to pick up new concepts relatively quickly, if it forms a significant part of the role then that is something I'd like to double-check sooner rather than later. On the plus side, having a candidate admit up-front they don't know something is a good sign which should be appreciated. Hence, in response to this I started throwing in the 'lambda or closure?' near the end just to see what the response was.

Some of the more subtle clues I managed to pick-up on were around the combination of choices. In a number of cases, the follow-up interview was a programming exercise – just a simple kata. Solving the kata in the short interview timeframe was never really the goal, but seeing how they approached it and responded to questions about their approach, was. For example in one case a candidate who had chosen 'abstract base class' over 'interface' and 'inheritance' over 'composition' showed the hallmarks of a C# programmer still thinking like a 90's C++ programmer, in the sense that the design was classic Gang of Four rather than more idiomatic C#. After that discovery, I made a note to try out biasing my follow-up questions slightly to explore class design in preference to other topics. (This was not the only characteristic which made that particular candidate unsuitable for that specific role but it did highlight how important writing code can be in an interview process.)

On a similar note, I began to use a candidate's preference for **`ToList()`**, the object initializer syntax, etc. as a cue to discuss mutability concerns. In the intervening years, I've started to throw in a few other classic trade-offs such as returns codes and exceptions, null references and options, and primitive types versus domain types, partly for variety but also to see what reactions it provokes or smells it might give off.

Conclusion

Since I started using this icebreaker a few years ago, I've never dismissed a candidate purely because I didn't like the choices they made. On the contrary, I'm well aware that 'one size never fits all' and diversity is to be embraced and therefore it's only been used as a line of conversation to explore the forces that drive their choice and, more importantly, to see whether they are conscious ones or not.

It's almost certainly an unconventional technique, at least by normal enterprise recruiting standards, and therefore it has the potential to have the opposite effect and make some candidates more nervous, not less. Not everyone is happy in an interview to just say the first thing that comes into their head and so I have been careful to gauge their opening responses to see if it's working against me. Apart from one candidate that proceeded to launch into a lengthy explanation on the very first question to justify their choice, I'm not aware of anyone else who's not got the hang of it pretty quickly.

Feedback from candidates appears to be favourable, mostly from the novelty angle. The fact that recruitment agents have mentioned it seems to back that up. The reason I find that feedback pleasing is that we have to remember interviews are a two-way street – the candidate is interviewing us as much as we are interviewing them. Asking the same boilerplate questions does little to help sell your position to them, whereas putting a little more effort into the process might help compensate for a less stellar job spec. In my experience working with interesting people rates pretty highly on the job satisfaction scorecard and so I'll try whatever tricks I can to try and lure those candidates in. I wouldn't want them to turn the job down prematurely because the interview process was uninspiring.

Interviewing for a new job is always a daunting prospect, especially when you're a bit rusty. Acknowledging that and factoring it into your interview process should help you make the most of those precious few minutes you get with each candidate. This icebreaker is intended to inject a little light-hearted humour into a traditionally stale process and, maybe, at the same also provide a rapid-fire technique for triangulating on further points of discussion. If nothing else, you'll have a bit of fun coming up with a list of amusing choices. ■

Advertise in C Vu & Overload

80% of readers make purchasing decisions, or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for info.

Best Articles 2018

Vote for your favourite articles:

- Best in CVu
- Best in Overload

Voting open now at:

https://www.surveymonkey.co.uk/r/HNP773J

DIY Technical Authoring

Alison Peck shares some of the tricks of her trade.

As a freelance technical author, I develop manuals, online help, tooltips tutorials, videos, simulations and other forms of 'user assistance'. Throughout this article, I'm going to refer to user assistance (UA) to refer to this type of 'documentation' to distinguish it from 'developer' and 'marketing' documentation, both of which are very important in their own right but which have very different purposes.

I love learning new things, and that includes getting involved in how people are using the software you all develop. As you may imagine, I'm keen for a professional technical author to be involved in UA. However, I'm a pragmatist and I know that not all organizations share my view… and even when they do, there are times when something needs to be done and the technical author (for whatever reason) isn't available. With that in mind, I thought it would be helpful to offer the people given the task a few pointers to help them do a good job.

First things first. To do the job effectively, you need the answers to the following questions:

1. When and why do people want to use the different features the software offers?
2. What is the most frequent and most effective way of completing a task, and are there any times when an alternative route through the software will be followed?
3. What is the existing level of knowledge of the different people using the software – both in terms of general IT-literacy and in domain-based knowledge? And for the latter, do they know enough?

But you all know that, right? It's what your use cases are built on. People want to achieve X, so you develop feature Y that enables them to reach their goal. It's obvious – and usually the software does that very well. For some reason, though, the documentation doesn't always do the same.

People use software to do something

Technical authors worth their salt know that (most of the time), you want to give instructions on how to do something – task-based instructions, not feature descriptions. That hasn't always been the case – back in the mists of time (the 1980s), the user manual often described the interface. Section headings like 'The Properties dialog' and 'The New Order page' were common, and the content would often have mostly been along the lines of: "The five menu options are: File, Edit, View, Sort and Help. The File menu contains…". When that still happens today is often because some pre-sales material (or some developer documentation) has been 'repurposed' to form the user assistance. This can work, but only if either it's been written with that use in mind or you're prepared to substantially edit it.

A good starting point for UA is to list the things people want to do – remembering also that most people go to the UA when they are stuck.

What are the tasks?

You probably know these already. You have your use cases and (hopefully) some knowledge of the subject domain. Often, though, your use cases are too granular (or, at least, the ones I've seen tend to be).

Use cases list separately all of the different instances under which a task may want to be done – making sure all are considered – whereas from a UA perspective, these are nuances of the same task. For example, you may have a use case that covers an existing customer creating an order, a new customer creating an order and someone who isn't a customer trying (and sensibly failing) to create an order. In UA terms, you're simply 'creating an order' and if you can only do so with a customer account, your first instruction is either to login or to create an account (and reference elsewhere for instructions on how to do that).

Depending on the subject domain, tasks may combine into workflows – and the same task may fit into more than one. Personally, I find a spray diagram/mind map a useful tool – it helps me to categorise the tasks as I think of them and make sure I haven't missed any out.

What sort of information to include?

The next step is to split the information you want to share into three categories. These can be called a number of different things, but for now I'll pick the terms used by DITA (the Darwin Information Typing Architecture, an XML schema designed for technical authoring tasks) [1]:

- *Concept* – the background information we all need to make sense of what follows. You can't start a set of instructions with "Step 1: Click…". First, people need to know what you're explaining, under what circumstances they may need to do this task, is this the set of instructions they are looking for… you get the picture.
- *Task* – the nitty-gritty, step-by-step instructions, usually numbered, and often with far too many screenshots (more on this later).
- *Reference* – often lists or tables of settings, abbreviations, options, icons and so on.

  Just in case you thought from my comment earlier that I never describe interfaces, I put what I call 'orientation' information into this category. For example, if just about every screen has five buttons along the bottom that jump you from module to module, or if tool palettes are managed in a particular way, then that is useful to know. (The information also doesn't need to be repeated every time you get to the end of a task or open a palette. ☺)

If you follow the reference to the easyDITA website [1], you'll see these referred to as 'topics' in DITA, as one of the principles behind DITA is 'write once, reuse many times' – but even if you're writing a more 'traditional' manual (and you may be surprised how many organisations – and their customers – still want those), thinking about the type of information you have to share is a good strategy.

How much should I include?

As with everything in life, the answer is: "It depends." It depends mainly on the answers to the three questions I posed at the beginning of this article.

Concept information

If you are providing instructions for a very simple task in a subject domain familiar to the user, a sentence or two of concept information for each task may suffice. This may be a task and domain familiar to just about all of us (for example, ordering something online) or one familiar to the users of that particular software (for example, doctors who are recording medical information – they know what they need to record, just not how). In both those cases, the 'concept' simply needs to specify the circumstances (as understood by the people using the software) in which these particular instructions need to be followed. What we are not attempting to do in

ALISON PECK

Alison has been ACCU's Production Editor since 2005 and a technical author for much longer. She works in many sectors, developing UA for software and machinery. She can be contacted at ajp@clearly-stated.co.uk

either case is 'teach' the user about the domain. In this case, we can also use domain-specific terminology (which isn't always the same as application-specific terminology) without explanation.

On the other hand, if you are writing for people who are using the software to accomplish a task outside their normal range of expertise, you may need to teach the user about the domain as well. For example, you may be developing UA for an accounting package used by business startups, who not only need to know what *could* be done but also what *should* be done: for example, not only the mechanics of generating invoices but some guidance on best practice as well. The best UA for people in this category gives a lot of guidance on what to choose depending desired outcome or constraints using everyday terminology. One of my biggest frustrations with UA I'm using is when there are two (or three or more) options that clearly state it's important which I choose to follow but without any indication about which option you should choose when!

You – I trust – already understand who will be using your software. When I join a team, one of the frequent questions I have to ask is, "Will the users of the software know why they are doing this?"

As well as the little bit of concept that introduces a task, you may need some concept information to set the scene for sequences of tasks or workflows. Anything that provides a context for the rest.

Task information

This part of the documentation can be quite formulaic, and the basics are very straightforward. A few 'rules of thumb' are given below… but, as always, there are exceptions.

- Prefer the simple present tense over future or conditional tenses.

  The majority of people are 'doing' while reading/following the UA: things are happening *now*. Save the future tense for things that will happen later and the conditional for things that might.

- Talk to them directly, using the imperative ("Click…" or "Select…") or address them as "you". For example: "If you forget your password, you must…".

  This does, however, depend on who you are addressing and who you are talking about; for example, "If users submit the same order multiple times, they must… " can be confusing if the readers are the users ("Does that mean me…?"), but not if the readers are customer support, trainers or administrators.

- Give instructions, don't word them as suggestions unless they *are* suggestions.

  You are giving instructions – trying to be too 'polite' can cause confusion. I once 'translated' the phrase "You may want to consider setting…" into "Optional: Set …". A colleague then deleted it as the remit was to simplify instructions, removing optional steps. Turns out it was an essential step in the process!

- One instruction, one step – and an instruction may be simply to review something. (*Very* small steps may be combined: "From the **File** menu, select **Properties** and then **Security**.")

  People may be looking for help when they are 'stuck' – something is not working as expected, or they don't know what to do. Putting multiple instructions in one step may mean some are missed.

- Don't number consequences (the windows that open, the messages that appear) – it helps separate the things the user is doing from the results of those actions.

  This particularly helps people who are just refreshing their memory – and not doing it can change a 4-step process into a 12-step one.

- Give all the information needed… but don't patronise!

  You don't want people to feel frustrated – either because some vital information is missing or because you've told them what they already know. Knowing what is 'needed' and what is 'patronising' comes from experience – but also from the three questions at the start of this article. You need to set a baseline for IT literacy in your target audience/readers. Can you assume they know the difference between left- and right-clicking? What a scroll bar is for? How 'obvious' are the labels and on-screen prompts?

  When I started developing UA, I was told by my boss (who wasn't a technical author herself) that I had to provide instructions for every field on the screen. This resulted in some very patronising statements, such as "In **Surname**, type the person's surname".

Do this…	**Not this…**
Enter **Start Time** and **End Time** using the 24-hour clock.	In **Start Time**, enter the time the consultation starts. In **End Time**, enter the time the consultation ends.
Type the **Surname** as it should be displayed. To enter accented characters, see…	In **Surname**, type the person's surname.
The form contains 20 questions – you may need to scroll to see them all.	Drag the scroll bar on the right of the window up or down to see the top or bottom of the page.

- If there are options or alternatives, state them before the associated instructions.

  There is no point in reading instructions when you didn't want to (or can't) do things that way anyway. Make it clear under which circumstances each instruction should be followed before giving the instructions (especially important when the instructions are long or complex).

Do this…	**Not this…**
To see a preview, click…	Click … to see a preview.
Windows only: Change user account permissions to…	Change user account permissions to… (Windows only).

What about screenshots?

Screenshots can be extremely useful to the users of the UA, and also extremely frustrating to both them and the person trying to capture them. Why should that be… surely a straightforward 'Print Screen' will do the job? And aren't they always helpful?

It's not usually the mechanics of getting the screenshot that's the problem – although some dedicated software such as Snag-It [2] or Greenshot [3] does simplify the task. It's more getting the screens populated with sensible information – that makes sense in sequence – before (or as) you capture them… and coping with late changes to the software. In general, populated screenshots are much better than empty ones as people tend to use them as a quick guide to what should be entered. The format of a telephone number, for example, can often be determined more quickly from a clear example than a convoluted explanation.

Many steps within a task don't require screenshots, and some tasks may not benefit from them at all. To begin, choose whether you want a non-populated screenshot at the beginning of the task, a populated one towards the end, both or neither. Only add intermediate screenshots if something significant happens that makes them worthwhile – perhaps a change to the options shown. And show partial screenshots if that makes things clearer.

Screenshot tips:

- Take a lot – they are quick and easy to capture, you don't have to use them all, and you have some source material for a bit of editing if necessary.
- Try to use sensible information – accepting defaults unless you want to illustrate something in particular.
- Save the screenshots in the capturing program's native format for easy editing, then resave in whatever format you need for the UA. For example, a .snag file [2] enables you to change the annotations at a later date. If you are saving to a standard image format, use .png – that way, you can edit field contents for consistency (you can often copy/paste appropriate field contents from the original screenshot over a new one).

China's New AI School Textbooks

Silas Brown is sceptical about an education initiative.

It's hard to write a review of a book I haven't seen, but I'm worried about the hype that currently seems to be doing the rounds in the Chinese news media about a series of 10 books aimed at children aged 7 to 15 called 'Future Smart Manufacturers on AI', which aims to bolster China's 'talent pool' by teaching children how to be 'AI' programmers. Titles in the series include *Magic Animals on AI*, *Smart Life on AI* (apparently a reference to the Internet of Things), *AI in the Shape-changing Workshop* (I'm not sure if that means 3D modelling or something else), *Cute Pet 'Little E' in AI*, *AI Super Engineer*, *AI's Backstage Hero: Python*, *AI in Future Towns*, *AI in a Wonderful World*, *AI Super Designer* and *AI's Uses and Explorations*. (The actual series name and titles are in Chinese only, with no official English translation, so all English translations here are my own.)

I always put 'artificial intelligence' in scare quotes because it's not really intelligence, it's just programming. People refer to computer-game characters as being controlled by 'AI' even when they're doing nothing more than a simple loop with perhaps a one-line piece of homing logic added. People refer to Chess engines as 'AI' even though they're mostly nothing more than optimised brute-force search algorithms. People refer to machine translation as 'AI' although it's usually just an application of statistics (albeit with rather a lot of data), with perhaps a precious few researchers still working on 'good old-fashioned' symbolic knowledge reasoning (the sort of thing that used to be written in Lisp or Prolog). Nowadays 'AI' usually implies some sort of attempt at writing a 'signal-to-symbol converter' (as in computer vision) by throwing a load of training data at TensorFlow or equivalent and not quite understanding how it works (which means you don't quite understand its limitations either, which is worrying), but if you're going to call a book *AI's Backstage Hero: Python*, that sounds to me like you simply want to write a general introduction to Python but your series editor is insisting that the word 'AI' must be in every title. This series might turn out to be something like the 1980s Usborne coding books that found their way into the children's sections of British public libraries, but I get bad feelings about the fact that the underlying message seems to be that the only type of computer programming that really matters is something called 'AI' programming.

Non-programmers sometimes ask me, "Can you do AI?" and I always answer "No," and then perhaps add "but the things I do might be called 'AI' by some people." It has become such a wishy-washy word, I don't know what it's supposed to mean anymore. Whatever happened to good old Computer Science? ■

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

DIY Technical Authoring (continued)

- Screenshots in steps don't need figure numbers or captions – the step number is the reference point and the instruction is the caption.
- If you have something like Snagit [2], capture a video. It makes a relatively large file, but you can extract individual screens from it – particularly useful if you're capturing screenshots from a mobile device or want to include steps that may fly past quite quickly.
- If your documentation is translated, limit your annotations to numbers – that way no-one needs to edit the image. Even if they are not translated, text in images is unlikely to be searchable.

What about QA?

You are a much braver (or foolhardy) person than I am if you are happy to release UA without some sort of QA process. As with software QA, there are a number of stages to go through:

- A review by the person who has developed it. Is it complete? Does it still contain 'placeholders', marking places for information you didn't have at the time? Assuming the UA has some sort of table of contents, is it in a logical order? Professional pride means you don't send anything for further review that you haven't reviewed yourself.
- A review by the subject matter expert (SME). Do the instructions match the reality, especially any concept information? (The SME will often know if something is wildly 'off' or if the step-by-step works but only in the specific scenario the author has followed.)
- A thorough review by the QA team. Nothing beats someone actually trying to use the UA to configure something or to complete a task. This review should cover:
  - Is the information complete and accurate?
  - Is it 'findable'?
  - Are there references to related information? And to prerequisites?

Everyone has an opinion

This is true about software development, but it's even more true about natural language. Others may have preferred ways of doing things that are different to your own – and many a healthy debate results. However, people in *other* disciplines are (I hope) more likely to tell you how something should work than how you should code it. In my field, it's different. After all, everyone can write, can't they…

People don't just tell you something isn't clear – they rewrite it for you. That can be the quickest and easiest option – otherwise someone says something isn't clear, you scratch you head over it wondering what is needed, only to discover you missed out a word. Ask the editors of the ACCU journals… if I spot something, I suggest alternatives. ☺

My advice? Be clear what aspects you want the review to cover. Follow the corporate style guide (or, even better, the one specifically for the UA). Treat 'rewrites' as 'suggestions' and decide each one individually. ■

References

[1] Stephani Clark (2017) 'When to use the Concept, Task, and Reference types in DITA' on easyDITA at https://easydita.com/when-to-use-the-concept-task-and-reference-types-in-dita, dated 14 November 2017

[2] Snag-It from Techsmith: screen capture and annotating software, which also captures video: https://www.techsmith.com/screen-capture.html

[3] Greenshot, open source screen capture and annotating software: http://getgreenshot.org/

ACCU Standards Report

Emyr Williams reports the latest from the C++ Standards meetings.

The ISO Committee for C++ met in San Diego California a few weeks ago, hosted under the auspices of Qualcomm. It was an exceptionally well attended meeting, with nigh on 180 people attending, which is the largest gathering for the C++ Committee meetings.

While the meeting was slated to be one of the last meetings for adding features in to C++ 20, the committee were eager to ensure that priority was given to proposals that might actually make it in to C++20. And this was no mean feat given that the pre-meeting mailing was 274 papers. Therefore, quite a number of papers to get through.

A good number of things made it through to C++20, and as ever some of the highlights are mentioned here. But before I go in to that, I would like to offer my sincere thanks to Roger Orr and Herb Sutter for being gracious enough to proof read what I've written and helping me to make sure that what I've written is an accurate reflection of what occurred in San Diego.

Ranges (p0896r3)

Ranges was adopted for C++ 20. This was due to a herculean effort from the authors, indeed my gmail inbox bears testimony to this. It has an amazing tribute (if that's the right word) to J.R.R Tolkien on its front page.

Casey Carter's git repo for Ranges can be found here: https://github.com/CaseyCarter/cmcstl2

Concepts convenience notation for constrained templates (P1141r1)

(Or 'concepts with a terse syntax'.)

This paper proposed to shorten the syntax for the constrained declaration of function parameters, function return types and variables. This means that in most cases, we'll no longer need to use the word 'template' or use 'requires' which can lead to code that no longer looks like this:

```
template <typename T> requires Sortable<T>
void sort (T &target)
{
  …
}
```

But rather looks like this:

```
void sort(auto Sortable &target);
```

Now, C++ will let you write many generic functions without the `template` keyword or angle brackets. The proposal as adopted means you'll typically need to use `auto` (which indicates that you are writing a generic function), part 2 of the proposal, which was not adopted, would allow you to simply use a concept name.

Now Herb mentioned in his blog, that "C++ will now let you write lots of generic functions without 'template' or angle brackets, and that are concept-constrained and therefore much easier to use correctly than function templates have ever been before." Which led me to wonder, "What cases would it NOT let you write generic functions without the 'template' keyword and the angle brackets."

Herb was gracious enough to e-mail me an answer, so my deep gratitude for his answer.

There are still instances where you'll need to use both keywords.

The right way to think of it is that the terse syntax is to make common cases easy, not to be a fully general syntax. (We already have one of those.)

The terse syntax will never cover all concept cases, and it doesn't currently cover some that it could in the future, such as multi-type concepts. So if you wanted to write something like:

```
template<class In1, class In2, class Out>
   requires Mergeable(In1,In2,Out)
   Out merge(In1, In1, In2, In2, Out);
```

then there is currently no shorter way to say it. Templates like this one still have to be written with the full elaboration, but many common ones can be written without the `template` keyword.

C++ and Constexpr programming

There has been quite a surge in compile-time programming features to C++ 20. This has been coming for a while and it's important for the current evolution of C++ as it becomes important to enable us to make effective use of compile time reflection.

This has been on the cards since we were given:

- Simple constexpr functions in C++ 11, i.e. a single return statement.
- constexpr with loops in C++ 14
- constexpr lambdas and 'if constexpr' in C++ 17

Yet more compile time features have been added to C++20, the highlights include:

- **consteval immediate functions (p1073r2)** While the constexpr specifier applied to a function or member function can indicate that a call to that function might be valid in a context needing a constant expression, it does not require that every such call be a constant-expression. However, there are times that programmers would want to define a function that should always produce a constant when it's called, whether that be directly or indirectly, and that a non-constant result should produce some kind of error.

  The aim of the paper was to address this issue. This allows developers to write functions that are guaranteed to be run at compile time. This work is essential for the kind of code generation that Herb's Metaclasses proposal will later rely on, as well as for the proposed future direction of refection.
- **constexpr union (p1330r0)** Allows developers to change the active member of a union at compile time. What does this give us? Well, some containers like `std::string` and `std::optional` use unions inside their implementations. And at the present time, changing an active element of a union isn't currently allowed in a constant expression. The idea of this paper is to help pave the way towards `std::string` becoming constexpr-friendly.
- **constexpr try/catch block (p1002r0)** This allows developers to write `try`/`catch` blocks in constexpr code. The motives for this was to help with reflection and with metaprogramming. The paper also stated that the limitation for this was encountered while looking at `std::vector` in libc++ with the view of making it constexpr enabled, as `vector::insert` uses a `try-catch` block to provide

EMYR WILLIAMS

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

the strong exception guarantee. This paper should dovetail nicely with the paper Standard Containers and constexpr (P0784r1)

It should be noted however, that this doesn't actually allow you to throw exceptions at run time.

- **constexpr dynamic_cast and polymorphic typeid in constant expressions (p1327r0)** There are a number of proposals to improve `std::error_category` which added new members or new virtual functions to it. Changing a standard polymorphic type is an ABI (Application Binary Interface) break, and this is something library writers are seeking to avoid.

  This paper aims to solve this by adding the ability to use `dynamic_cast` in constant expressions.
- **std::is_constant_evaluated (p0595r1)** Allows for conditional code to be written that lets you write part of a function using code that is guaranteed to be executed at compile time.
- **constexpr std::pointer_traits (p1006r1)** This is essential for enabling compile time reflection and to enabled `std::vector` to become constexpr. (The latter is dependent on constexpr new, which is still under development at this time.)

Modules

Modules now has an approved unified design targeting C++ 20. As such there is more work to be done on wording; however, it is expected that the committee will consider modules at the next meeting in February.

Coroutines

There is still a lot of work being done with coroutines. EWG recommended merging the coroutines TS (n4774) in to C++, and the also planned to include features from Core Coroutines (p1063r1) proposal as well. However, the vote to merge the TS fell just short of the required numbers, so it wasn't adopted for C++ 20 at this meeting.

However, with collaboration from Facebook and other companies, there will be more work done over the winter to address the concerns that remain, as well as doing further merging work from the Core Coroutines paper in to the TS. It's expected that coroutines will be proposed again at the next meeting (Kona), with the new information, and it will give the national bodies time to get their heads around this new information.

Executors

Due to the progress between the previous ISO meetings, and some special meetings in between, it's starting to look like an initial executors design could make it for C++ 20. While the paper wasn't merged in San Diego, the design was approved, and again it's a case of working on the wording, which will be discussed at the next meeting.

Networking

For a while now, it has been known that this work is dependent on executors, and there has been discussions about decoupling the parts that are reliant on executors so that there is at least some networking functionality in C++ 20. However, it was decided at the meeting to target merging the Networks paper in to C++23; while this is disappointing, it also gives the committee time to work through whether there is integration work to be done with merging Networking with Coroutines.

New groups...

A number of study groups were formed at the meeting, as well as two incubator study groups.

- **SG13 – Human-Machine Interface Study Group** was reformed with ACCU's very own Roger Orr as chair, and will be working on figuring out what the next steps are for the 2D graphics proposal. They met at San Diego, and spent half a day or so scoping the work they were going to tackle. It's highly probable that an extra meeting specifically for 2D graphics in C++ will be held with the purpose of trying to get feedback from experts in the field.
- **SG19 – Machine Learning Working Group**, chaired by Michael Wong was formed to take advantage of C++'s strengths in generic programming, as well as code portability for the specific area of Machine Learning. The main aim of the SG19 group is to work on improving C++'s ability to support:
  - Fast iteration
  - Arrays
  - Matrices
  - Linear Algebra
  - In memory passing of data for computation
  - Optimisation for graph programming

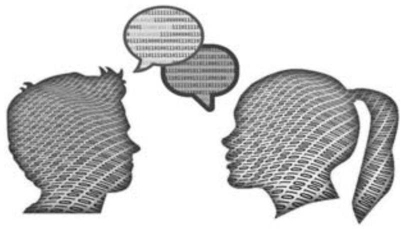
  to name a few highlights.
- **SG20 – Education**, chaired by JC van Winkel was formed to improve the quality of C++ education, and to enable developers to use the language, its features and ecosystem to write correct, maintainable and performant software. The group will aim to create a set of guidelines for a educational curriculum for various levels of expertise and application domains, as well as encourage WG21 paper authors to include guidance on how to teach the new feature they are proposing to add to the standard.

A pair of incubator groups were also formed. The intention is that these groups will be additional stages in language and library evolution pipelines. Their focus is to refine, merge and filter new and forward-looking proposals and to ensure that effort is focussed on delivering C++'s long term strategic goals. The groups are as follows:

- **SG17 – The Evolution Working Group Incubator**, chaired by JF Bastien. The idea behind this was that the EWGI would be able to look at papers that the EWG wouldn't have time to look at. They worked through about 30 papers at the meeting, and sent a few onwards to the EWG. These were:
  - `nodiscard` should have a reason (p1301r0) (This paper will also be sent to the LEWG, as they could want to add reasons to existing uses of `nodiscard` in the standard library.)
  - Array size deduction in new-expressions (1009r0)
  - Make `char16_t` and `char32_t` string literals be UTF-16/32 (p1041r1)
  - Named character escapes (p1097r1)
  - Using `enum` (p1099r2)
- **SG18 – The Library Evolution Working Group Incubator (LEWGI)**, chaired by Bryce Adelstein Lelbach. The main role of the LEWGI is to give directional input on proposals as well as early design reviews. During this meeting they were focussing on post C++ 20 content, however some things were sent for consideration for C++ 20 such as `std::unique_function` (p0288r1) which is similar to `std::function` except it doesn't have a copy constructor or copy assignment operator, which allows it to wrap function objects containing non-copyable resources.

Code Critique Competition 115

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 am trying to write a simple test for some C++17 code, but I am getting some unexpected output. I have simplified the code quite a bit, and in the resultant code below I was expecting to see the output "A,B" but on the latest version of both gcc and MSVC I get just "AB". Where has my comma gone?! Even more oddly, if I use an older compiler I get different output: "65,66", which gives me back the comma but loses the letters. Has C++17 broken something?

The code is in Listing 1.

Listing 1

```
#include <iostream>
#include <string>
#include <vector>
template <typename T>
using coll = std::vector<T>;
// Start the collection
template <typename T>
void start(coll<T> &up)
{
  up.clear();
}
// End the collection
template <typename T>
void end(coll<T> &up)
{
  up.push_back({});
}
// Extract the non-zero data as a string
template <typename T>
std::string data(coll<T> const &up)
{
  std::string result;
  for (auto v : up)
  {
    if (v)
    {
      result += std::to_string(v);
      result += ",";
    }
  }
  result.pop_back();
  return result;
}
void test_char()
{
  auto i2 = coll<char>();
  start(i2);
  i2.push_back('A');
  i2.push_back('B');
  end(i2);
  // actual test code elided
  std::cout << data(i2) << '\n';
}
int main()
{
  test_char();
}
```

Critiques

Balog Pal <pasa@lib.hu>

The code looks simple enough so I try a pure reading, no compiler solution. Let's scan the code first:

- **coll** is defined as an alias for **vector<>** for no visible reason, but that may be part of simplification
- a function called '**end**' is looking for trouble, clashing with **begin()** and **end()**. Also the name makes no sense for the semantics, might be **finish()**
- inside **end**, a more optimal form of insertion would be **emplace_back()**
- another nonsense function name: '**data**'
- implementation iterates by value instead of **auto const&**, that is okay with the **char** type, but suboptimal for a general **T**
- **T** must support **if(v)** – this works for **char** but might be bad, no information was provided on the intended **T** for the code
- I have to look up **std::to_string** on cppreference: it appears we can use it for numeric types and will produce string-ized version of the value. Like "65" for input of 'A'.
- if empty **v** is meant as a sentinel, I'd expect a break on **else**
- **pop_back** will be bugged if **up** was **empty** (but not executed this way by the test)
- the test starts with an **auto X =** ...; form that seems to be gaining popularity lately, I'm still undecided if it is good or bad. The traditional form is certainly a direct **init: coll<char> i2;**

And nothing else interesting, so I would expect "65,66" as result. And the description states it is "AB"? Hmm, that looks like we asked for **i2.data()**. Maybe we added another free function **data()** to the company of **begin**/**end**, **size** and friends? A look in cppreference shows that has been the case since C++17.

I guess telling that to the OP would result in the question: how come that is relevant to me, I never said **std::data**, not even used using namespace **std**. And we are in global scope. Shouldn't **::data** be called?

Well, for some interpretations of 'should', it should be alright. It seems we have stumbled on another case of computers work on commands instead of wishes. At first I thought of just passing over the issue stating that name lookup and overload resolution rules are not for ordinary humans

ROGER ORR

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

to explain, but this particular case is not THAT complicated, and even I could figure it out without extra reading (of course ONLY after the fact/accident ☺) The explanation is deliberately simplified for this case. Anyone who wants the full run-down can read it at https://en.cppreference.com/w/cpp/language/overload_resolution and the name lookup link at its beginning.

The first thing to explain is how **`std::data`** gets considered at all. That is a thing that ordinary programmers should be aware of really: we have names visible from the current scope plus any associated namespaces related by arguments. So the latter, argument-dependent lookup drags in **`std::`** because our argument is in reality **`std::vector`**. That we used it through an alias does not change its origin. If **`coll`** was a class template that just reproduced **`vector`** or subclassed it, we would not fall into this trap.

While we are at it, let's mention that this only applies to unqualified lookup, if spelled **`::data()`** in the call, again we would be calling our function.

The next step is more interesting: why is **`std::data`** better than **`::data`**? The compiler could still pick ours or declare it ambiguous. The rules state an ordering of candidates to find a best fit based on fewer conversions or more specialization. And at first glance ours looks winning, as **`coll<T>`** is more specialized than just **`<T>`**. But again the language drops another rule on us that has proven unintuitive and misleading to programmers in many other cases. **`std::data`** has not one but two overloads matching our case, one taking **`T&`** and one taking **`T const&`**. And our ammo, **`i2`**, is 'lvalue of type **`coll<char>`**'. What I usually just think of in **`decltype`** terms as '**`coll<char> &`**'. certainly can be bound to a **`const&`**, but that involves a qualification conversion. If all we had to select from were the **`std::data()`** functions, the non-const one is used. And with **`::data`**, there is no such version, so that will lose the race. We would not have the problem if **`i2`** was const.

So what we get is what we ordered. Did C++17 'break something'? That is up to interpretation. The language is not supposed to make subtle changes to well-written code without making some noise. And the presented code in not obviously badly written. Strictly speaking it was 1) open to ADL from the start and 2) the standard stated from the start that it can add names to **`std::`** at any time. And it does. Though I didn't find a promise, the names used by the standard are all-lowercase. For practice the industry relies on that, and expects no clash from mixed-case names.

This code used all-lowercase names and really poor ones too. I'd rather go with the other wisdom of "Bad names will come and bite you in the ass. If not today, then tomorrow." Here it took all the way to C++17, but it happened. (Let's just kindly overlook the multitude of elements required to create the perfect storm, even be careful to assemble a fine zero-terminated string in **`i2`** to avoid UB.... ☺

The one thing I could not figure out is why the OP expected to see letters "A" and "B" from this, but I'm happy to leave it as mystery.

Alastair Harrison <aharrison24@gmail.com>

Given the unexpected behaviour of the program, a good approach is to try stepping through the **`data`** function with a debugger. Perhaps surprisingly, this shows that the **`data`** function turns out to never be executed. Neither does the **`end`** function. In fact, you can totally remove them from the program and it will still compile and run without complaint on C++17.

The reason is Argument Dependent Lookup (ADL) [1]. ADL is a mechanism which allows the compiler to find free functions that live in other namespaces, without the calling code needing to specify that namespace. The example below shows why that's (usually) helpful. The **`accu::print`** function is effectively treated as part of the public interface of the **`Widget`** class. This can be very useful when writing generic template functions when we don't know up-front which namespace the arguments live in. But sometimes ADL can unexpectedly call the wrong function, as is the case in the original program.

```
namespace accu {
  class widget {};
  void print(widget const&);
}
void foo(accu::widget const& w) {
  ...
  // ADL allows compiler to find the
  // accu::print function, even though it's in
  // a different namespace.
  print(w);
  ...
}
```

In C++11, the **`std::end`** template function was introduced. You pass in a reference to an STL container, and it will return an iterator to the end of the element range. It's an unconstrained template (ie. accepts any argument type) and ADL is choosing it instead of the **`template <typename T> void end(coll<T> &up)`** template function that we've defined in the global namespace.

In the C++ Core Guidelines, rule T.47 [2] tells us to "Avoid highly visible unconstrained templates with common names" because often ADL will select them in preference to the function you might expect. The text points out that the standard library violates the rule widely. And that's exactly what is causing problems for us now.

Since **`std::end`** doesn't mutate the container, the effect of the **`end(i2)`** call in the **`test_char`** function is to do absolutely nothing. Thus the container will be left holding only the letters **`A`** and **`B`**, without a terminating character.

But it gets better, because the C++17 standard introduced another unconstrained template function called **`std::data`**. It accepts a reference to a container and returns a pointer to the block of memory where the container's elements are stored. So when the **`test_char`** function calls **`data(i2)`**, ADL is actually selecting the **`std::data`** template, which returns a **`char*`** pointer to the contents of the **`i2`** container. That's then being passed to the **`char const*`** overload of **`operator<<`**, which treats the input as if it's a null-terminated character array.

As we discussed above, our container doesn't actually contain a null terminator. But luckily for us, the vector... probably allocated more than two bytes of storage, and it's... probably filled with zeros that look like null terminators. So the program prints the string "AB" and quickly finishes before anyone notices that it's just caused some Undefined Behaviour.

If you're curious, you can call **`i2.reserve(2);`** immediately after instantiating the container, to limit the memory allocated. Then compile the code using Clang's address sanitizer [3]. The program will now blow up impressively when it tries to print to **`std::cout`**, whizzes right off the end of the memory allocated by the vector and plunges into the nether regions of the free store.

The obvious solution for the ADL problems is to rename the free functions to give them less common names. Another option is to create a strong type to implement the custom container, rather than simply making it a typedef for **`std::vector`**.

So what's happening on older compilers? Under C++14, the **`std::data`** template doesn't exist, so the **`data`** template in the program gets used as expected, and prints some comma-separated values. Unfortunately **`std::to_string`** doesn't have an overload for **`char`**, so it ends up performing an implicit conversion to some other integral type, then returns a textual representation of the integer value as a **`std::string`**.

Sadly we're not done quite yet, as there's another bug lurking in the **`data`** function. When the input container is empty or contains only null elements the **`result`** string won't have any characters appended to it. The **`result.pop_back()`** call will then fail (with more of the dreaded Undefined Behaviour) as we've violated its pre-condition that the string is non-empty. This is the sort of thing that unit-tests should be exercising.

Other improvements

Because of the code author's simplification efforts, it's not clear how the original code is intended to be used (especially given the alarmingly terse variable names). If you delimit multiple sequences by calling **`end`** after each one then the **`data`** function won't format them properly; it'll just remove the comma between subsets. If instead the aim is to have only a single sequence then the **start** and **end** calls are superfluous. Worse, there's nothing to stop someone calling **end** multiple times and no mechanism to prevent elements from being inserted after a call to **end**.

We also need to talk about the use of a default-constructed element as the end-delimiter. Though this seems to work tolerably well when the element type is **char**, it could be a total disaster for **int**, because the caller could legitimately want to place a zero value into their container. A more robust design would ensure that the **only** way to insert an end delimiter is via an explicit API call. It shouldn't be possible to do it accidentally.

For the multiple-sequence case a better design could involve using another container as the element type, as shown below. It requires no delimiters, so the question of delimiter representations is completely avoided, as is the need for calls to **start** and **end**.

```
template <typename T>
using subset = std::vector<T>;

template <typename T>
using coll = std::vector<subset<T>>;

// Used like this
auto i2 = coll<char>();
i2.push_back({'A', 'B'});
i2.push_back({'C', 'D', 'E'});
```

Finally, the **data** function in the original code uses a fairly awkward loop to print a comma-delimited list of elements. It could also be quite inefficient for non-trivial element types, as the **range-for** loop copies every element. An alternative would be to use an **ostream_iterator**, as shown below. If the thought of a trailing comma horrifies you, then you can use a custom iterator such as Jerry Coffin's **infix_iterator** [4]. Or wait for **std::ostream_joiner** to be standardised.

```
template <typename InputIter>
std::string comma_delimit(InputIter first,
  InputIter last) {
  std::stringstream ss;
  std::copy(first, last,
    std::ostream_iterator<char>(ss, ","));
  return ss.str();
}
```

References

[1] Argument-dependent lookup in cppreference.com: https://en.cppreference.com/w/cpp/language/adl

[2] C++ Core Guidelines: http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t47-avoid-highly-visible-unconstrained-templates-with-common-names

[3] AddressSanitizer in the Clang 8 documentation: https://clang.llvm.org/docs/AddressSanitizer.html)

[4] c++ infix_iterator code on Code Review Stack Exchange: https://codereview.stackexchange.com/questions/13176/infix-iterator-code

James Holland <James.Holland@babcockinternational.com>

At first sight, the behaviour of the running code, as reported by the student, seems impossible. Fortunately, a little investigation reveals what is really going on. The solution to the puzzle centres on the call to **data()** in the output statement of **test_char()**. It would be reasonable to assume that the function **data()** defined in the student's code is being called but this is not the case. In fact it is **std::vector::data()** that is being called.

One of the mechanisms that contribute to determine which function is invoked (when there is more than one candidate) is called Argument Dependent Look-up (ADL). The argument of **data()**, **i2**, is of type **std::vector<char>** and by means of a fairly complicated set of ADL rules, the compiler determines that **std::vector::data()** should be called in preference to version of **data()** defined in the student's code sample.

The function **std::vector::data()** returns a pointer to the data contained within the vector **i2** which, in our case, is an array of **char**s. The array is then displayed by **std::cout**. It is fortunate that **end()** is called before being printed to the screen. The function **end()** adds an element to the end of the vector, **i2**, that has a value of zero. Without calling **end()**, the array of characters printed would not be null-terminated and so random characters would be printed after "AB" until the character with a value of zero was chanced upon. This also explains why there is no comma in the output, **std::vector::data()** simply prints the characters in the vector without any intervening characters.

The student noted that with older compilers, the printed result is "65,66". This is because **std::vector::data()** was only introduced in C++11. When using a pre-C++11 compiler, ADL would not have found a function named **std::vector::data()** and so the student's version of **data()** would have been called.

Unfortunately, there is a 'feature' of the student's **data()** function that converts the characters to be printed to the ASCII representation. The call to **std::to_string()** performs this transformation. The ASCII representation of 'A' and 'B' is "65" and "66" respectively. The statement containing **std::to_string()** should be replaced by **result += v**, or something equivalent.

There are several ways to ensure the student's version of **data()** is called. One way is to qualify the call of the function **data()** with the scope resolution operator, **::**. Doing this will disable the ADL mechanism and force the compiler to look in the global namespace for the function where it will find the correct version of the function. Another way is to call the required function with an explicit template parameter, i.e. **data<char>()**. Perhaps the safest way is to rename the function to something that is not in the **std::vector** namespace and is more specific to the application at hand.

The problems described above illustrates that an addition to the C++ standard can break existing code. In this case, the addition of **std::vector::data()** in C++11 was the cause of the trouble. It is important that when code is compiled with a new or different compiler, the software is retested before release.

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

There are two aspects to the code for Code Critique 114. One is the behaviour on compilers that do not support C++ 17, and the other is the behaviour under the current standard. Finally, there are some general comments on the code.

Behaviour pre-C++ 17

Type **char** is actually an integer type. Therefore, when **result += std::to_string(v);** is executed, the number in **v**, which is the ASCII code of the character, is turned into a string and added to the result. Hence, the result is 65,66. Leaving out **std::to_string** might work, i.e. **result += v;**. Alternatively, using **std::string::append** certainly should work with type **char**: **result.append(v);**. Of course, the comma works as expected between the output of the ASCII codes of the characters in the vector.

Behaviour under C++ 17

The problem here is a combination of factors 'conspiring' to hide the user's **data()** function for the call made at the end of **test_char()**.

The new standard provides a function template **std::data()** with a very similar signature to that the user has defined. Due to argument dependent lookup, this is in scope for the call in question. Argument dependent lookup means that because the collection **i2** is in fact a **std::vector**, functions in the containing namespace, **std**, are valid candidates.

Since **std::data()** also offers a template for non-const parameters – **template <class C> constexpr auto data(C& c) -> decltype(c.data());**, see https://en.cppreference.com/w/cpp/iterator/data, overload (1) – this is a better fit than the user's own definition of **data()**.

The `std::data()` function invokes the container's own `data()`, which in the case of a `std::vector<char>` returns "AB" as a single string.

This problem may also hold for `end()`; while there is no `std::start()` I am aware of, a non-member `std::end()` exists to get the iterator one past the end of a collection. In this example, the code works even if `end()` does not add a default-constructed item and the iterator returned by `std::end()` is ignored, but it does constitute a 'code smell'.

To avoid such problems, it is a good idea to follow the advice of the (forthcoming) Standard Library Compatibility Guidelines (see https://isocpp.org/std/standing-documents/sd-8-standard-library-compatibility). At CppCon 2018, Titus Winters introduced these in his talk 'Standard Library Compatibility Guidelines (SD-8)' (see https://www.youtube.com/watch?v=BWvSSsKCiAw). Pertinent to this code is the fact that the standard library reserves the right to add new functions, so if you have any functions that can potentially take data types from the std-namespace as parameters, these should be put into a namespace and the namespace specified explicitly in the call to avoid problems.

General comments on the code

In its current form, `coll`, `start()` and `end()` are not a proper abstraction, because to add an item, the user has to use the underlying container's support for adding items. Furthermore, it is assumed that `operator bool() const` of type `T` is equivalent to a function that would best be called `isPrintable` or `is_printable` (although snake case is discouraged, see again the aforementioned talk by Titus Winters).

This could lead to bugs, such as if a type has an `operator bool() const` that does not yield false for a default constructed object, so the assumed end marker (which is not required anyway, because the range-based for-loop prevents overstepping the vector bounds anyway) would be printed, adding an item that was added by the user's `end()` function, rather than an explicitly added item.

The other way around, there might be types where some values yield true, but `isPrintable` should really be false for them. And, in fact, if `char`s were not treated as numbers, `char` is such a type with many of the ASCII-values below 32 representing non-printable characters like bell and end of text.

Jim Segrave <jes@j-e-s.net>

This one was interesting – I tried it with g++ 4.85 and clang++ 3.4.2 (the default Centos 7 distributions) and got the `65,66` output as described in the problem statement. I then switched to c++17 compilers (g++ 7.31 and clang++ 5.01) and got the `AB` output.

The `65,66` output is a normal result, there's no reason to use `to_string()` if you want the ASCII character. Changing the line

```
result += std::to_string(v);
```

to

```
result += v;
```

fixed the `65,66` problem when compiled with the option `-std=c++11`.

To identify the cause of the `AB` output when using the `-std=c++17` option, I tried placing some output statements within the `data()` function, because I could not see why it wouldn't at least output values separated by commas. I discovered that, when using c++17 as a standard, the function was never called. Checking with the nm utility showed it was in the compiled executable with the expected return type and parameter list. Two quick experiments told me it was time to dig deeper into the changes from c++11 to c++17.

Changing the line:

```
std::cout << data(i2) << '\n';
```

to

```
std::cout << xdata(i2) << '\n';
```

caused the function to be invoked. It also gets invoked if that line is changed to:

```
std::cout << data<char>(i2) << '\n';
```

This tells me that the compiler isn't seeing `data(i2)` as a function call, but something else entirely. I thought it was time to look deeper still into the c++17 changes and discovered this (described in http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4280.pdf) which details changes to allow getting the size of a container, testing if the container is empty and getting a pointer of the appropriate type to the container's data all as `constexpr`'s rather than requiring a member function call on the container.

From the above document, the standard now adds:

```
template <class C> constexpr auto
data(const C& c) -> decltype(c.data());
```

Returns: `c.data()`.

In other words, the line in `test_char`:

```
std::cout << data(i2) << '\n';
```

does not call the function `data()`, it simply replaces `data(i2)` with a pointer to the data in the vector `i2` (equivalent to `i2.data()`). This is a pointer to `char`, pointing to `i2[0]`, `i2`'s values are stored contiguously in memory and, luckily, the sample program pushes an empty value onto the vector (a NUL char), so `cout` outputs the contents of `i2` in order as a C-string. It's not clear to me that in the absence of the call to `end()` which appends a NUL to `i2` that you might not get a core dump or a collection of output for locations not a part of `i2`'s data at all.

As a side note, perhaps it's just my own personal prejudice, but the method of building a string of values separated by commas is better done by always outputting a separator and the next value. At the start, the separator is set to a null string, after outputting a value, the separator is changed to the actual string to separate the outputs:

```
const char *sep = "";
std::string res;
for (auto v: i2) {
  res += sep;
  res += v;
  sep = [whatever you want as a separator]
}
```

The advantages of this method are that the string being built is always ready for use, it contains values separated as desired with no separator before the first value or after the last one. If you need to use the string while it's still being built, it's ready for that. It also means that if you decide to change the separator from a single character `','` for example, to include a space as well: `", "`, you only change one line of code, you don't need to add a second `pop_back()` call. The separator could even be passed as a pointer to `const char`.

My submitted code makes this change, it will call `test_char()` once with the default separator, giving output `A,B`, then call it again with a separator of `"-->"`, giving an output of `A-->B`.

```
#include <iostream>
#include <string>
#include <vector>
template <typename T>
using coll = std::vector<T>;
// Start the collection
template <typename T>
void start(coll<T> &up) {
  up.clear();
}
// End the collection
template <typename T>
void end(coll<T> &up) {
  up.push_back({});
}
// Extract the non-zero data as a string
template <typename T>
std::string data(coll<T> const &up,
  const char *separator) {
  std::string result;
  const char *sep = "";
  for (auto v : up) {
    if (v) {
      result += sep;
      result += v;
      sep = separator;
    }
  }
  return result;
}
void test_char(const char *separator = ",") {
  auto i2 = coll<char>();
  start(i2);
  i2.push_back('A');
  i2.push_back('B');
  end(i2);
  // actual test code elided
  std::cout << data<char>(i2, separator)
    << '\n';
}
int main() {
  test_char();
  test_char("-->");
}
```

Commentary

This issue's critique was inspired by recent discussions and example code demonstrating how code could be broken by the addition of new functions into namespace **`std`**, including cases where many programmers would not be expecting trouble.

The second angle in this critique is the dual role of **`char`** in C++ to represent characters and as an arithmetic type. When a **`char`** argument is passed to **`std::to_string()`**, the best overload is selected using the *integral promotion* rules. This will typically select the **`int`** overload of **`std::to_string()`** (although it could select the **`unsigned`** overload on some platforms.)

Note, too, the final subtlety of the rules for overload resolution and ADL in that the call to **`end()`** picks the locally defined function template while the call to **`data()`** picks the one found through ADL – this is because the of the different **`const`** qualification of the argument in the two functions.

The winner of Code Critique 114

The critiques between them seemed to give good coverage of the issues in the code. There were a variety of ways used to explain the name lookup issue that resulted in finding an 'unexpected' overload of **`data()`** – this is the sort of explanation that is much easier face-to-face as you can quickly adjust the explanation to the degree of understanding of the recipient!

There are several ways to avoid the unfortunate name lookup – it's good to know more than one technique as the best one to use depends on the circumstances of the case and James provided three. Alastair pointed out that avoiding the use of a simple alias for **`std::vector`** side-steps the problem here, Marcel and Jan's critique suggested putting the function into its own namespace, and multiple critiques mentioned renaming the function and using the scope resolution operator.

Alastair reported that **`end()`** was not called – which puzzled me since it *should* be called in all versions of C++. However, he was correct that it can be removed as this then allows overload resolution to pick the less specialized *end()* function from namespace **`std`**. (Adding a suitable **`static_assert`** to the **`end()`** function template is one way of demonstrating whether or not the template is used.) This led him slightly astray – but into an interesting discussion about whether or not the 'one past the end' character would be zero.

Pal pointed out a couple of details in the implementation where there might be optimisation opportunities: one of the challenges with writing templates in C++ is to try and consider the possible types that might be used in instantiations and ensure that good choices are made for the general case, such as avoiding unnecessary copies.

Jim gave a useful side-note about delimited string, or alternatively you might like one of the iterator solutions provided by Alastair.

Both Alastair's and Marcel and Jan's critiques also gave some useful discussion about the design issues with the code as presented (this is always a bit of a challenge given the simplified nature of the critiques presented in this column).

Taking all the points into consideration, I decided to award the prize for this issue to Marcel and Jan's critique.

Postscript to Code Critique 113

Balog Pal writes:

I almost wrote my entry at first sight – it was so perfect a bait – just one of my solutions didn't compile and I held back… 2 months is plenty of time, so obviously I realized what I forgot 2 days after the deadline. I then thought still to post it as a late entry, but expected plenty of submissions with similar points, so thought it better to wait and only post if something was missed.

And there is such a point: there was much talk about undefined behavior and what to expect under specific ABIs, yet the first critique never mentioned it. Some may even believe that if an implementation specifies how arguments are passed and they are placed as we would like them, the example would be correct. It is not.

The expression **`&first + 1 + sizeof...(rest)`** has defined behavior only when the last part is 0. Let me quote the standard at [epr.add] on p4:

> When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the expression P points to element x[i] of an array object x with n elements[86], the expressions P + J and J + P (where J has the value j) point to the (possibly-hypothetical) element x[i + j] if 0 _ i + j _ n; otherwise, the behavior is undefined. Likewise, the expression P - J points to the (possibly-hypothetical) element x[i - j] if 0 _ i - j _ n; otherwise, the behavior is undefined.
>
> [86]) An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1. A pointer past the last element of an array x of n elements is considered to be equivalent to a pointer to a hypothetical element x[n] for this purpose; see 6.7.2.

Now, **`first`** is a solo object, so the pointer **`&first`** is good for the addition of 0 and 1. Anything beyond that is undefined behavior.

IME, it's a point too often overlooked. Probably because it happens to 'work' on most current flat-memory platforms. Those who remember the old DOS or Windows 16-bit time where you had **`__huge`** pointers and such a pointer had a 16-bit paragraph or selector part and a 16-bit offset part. And using a wild + (or especially -) that looked outside an 'object' could easily produce broken result or even a direct crash.

Code Critique 115

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

> I am trying to write a generic 'field' that holds a typed value, optionally with a 'dimension' such as 'kg'. One use case is to be able to hold attributes of various different types in a standard collection. I want to be able to compare two attributes to see if they're the same (and maybe later to sort them?) – two simple fields are only comparable if they are the same underlying type and value and two 'dimensioned' fields need to be using the same dimension too. I've put something together, but I'm not quite sure why in my little test program I get true for comparing house number and height:

```
Does weight == height? false
Does weight == house number? false
Does house number == height? true
Re-reading attributes
Unchanged
Reading new attributes
some values were updated
```

Can you help with the problem reported, and point out some potential flaws in the direction being taken and/or alternative ways to approach this problem?

The code is in Listing 2 (`field.h`) and Listing 3 (`field test.cpp`).

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://accu.org/index.php/journal).

This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
#pragma once
#include <memory>
#include <string>
// Base class of the 'field' hierarchy
class field
{
protected:
  virtual bool
  equality(field const &rhs) const = 0;
public:
  // convert any field value to a string
  virtual std::string to_string() const = 0;

  // try to compare any pair of fields
  friend bool operator==(field const &lhs,
    field const &rhs)
  {
    return lhs.equality(rhs);
  }
};
// Holds a polymorphic field value
struct field_holder : std::unique_ptr<field>
{
  using std::unique_ptr<field>::unique_ptr;
  bool
  operator==(field_holder const &rhs) const
  {
    return **this == *rhs;
  }
};
// Implementation in terms of an underlying
// representation
template <typename Rep>
class field_t : public field
{
public:
  field_t(Rep const &t) : value_(t) {}
  std::string to_string() const override
  {
    return std::to_string(value_);
  }
protected:
  bool equality(field const &f) const override
  {
    // compare iff of the same type
    auto p =
      dynamic_cast<field_t const *>(&f);
    if (p)
      return value_ == p->value_;
    else
      return false;
  }
private:
  Rep value_;
};
// Add a dimension
#include <typeinfo>
```

Listing 2 (cont'd)

```
template <typename dimension,
  typename Rep = int>
class typed_field : public field_t<Rep>
{
public:
  using field_t<Rep>::field_t;
  std::string to_string() const override
  {
    return field_t<Rep>::to_string()
      + dimension::name();
  }
protected:
  bool equality(field const &f) const override
  {
    // compare iff the _same_ dimension
    return typeid(*this) == typeid(f) &&
      field_t<Rep>::equality(f);
  }
};
```

Listing 3

```
#include "field.h"
#include <iostream>
#include <map>
// Example use
struct tag_kg{ static const char *name()
  { return "kg"; } };
struct tag_m{ static const char *name()
  { return "m"; } };
using kg = typed_field<tag_kg>;
using metre = typed_field<tag_m>;
using attribute_map = std::map<std::string,
  field_holder>;
// Dummy method for this test program
attribute_map read_attributes(int value)
{
  attribute_map result;
  result["weight"]
    = std::make_unique<kg>(value);
  result["height"]
    = std::make_unique<metre>(value);
  result["house number"]
    = std::make_unique<field_t<int>>(value);
  return result;
}
int main()
{
  auto old_attributes = read_attributes(130);
  // height and weight aren't comparable
  std::cout << "Does weight == height? "
    << std::boolalpha <<
     (old_attributes["weight"] ==
      old_attributes["height"])
    << '\n';
  // weight and house number aren't comparable
  std::cout << "Does weight == house number? "
    << std::boolalpha <<
     (old_attributes["weight"] ==
      old_attributes["house number"])
    << '\n';
  // house number and height aren't comparable
  std::cout << "Does house number == height? "
    << std::boolalpha <<
     (old_attributes["house number"] ==
      old_attributes["height"])
    << '\n';
  std::cout << "Re-reading attributes\n";
  auto new_attributes = read_attributes(130);
  if (old_attributes == new_attributes)
  {
```

Report on Challenge 5

Francis Glassborow comments on his previous challenge, and provides a new puzzle.

Challenge 4 produced a large post bag. Unfortunately this one produced almost nothing. Well, I had one partial submission. I gave the author of that an extra hint but that resulted in effective silence.

As usual, I tried to give some pointers in the introduction to the challenge but no one seemed to pick up on it. You need some lateral thinking and you, the readers, are usually pretty good at that. As I suggested by relating the competition to a language of seven words, seven is actually surplus to requirements once you realise that words can have multiple meanings that can be selected by a suitable mechanism. Let me elaborate a little more on that. Even without fancy use of binary coded selectors, each of your seven words could have a position dependency in three word sentences. For example, given a sentence structure 'x y z' and a word abc: abc could mean 'put' when replacing x, 'it' when replacing y and 'there' when replacing z. So the sentence 'abc abc abc' translates as 'put it there'.

That is a simplistic example to illustrate the idea of positional meanings (actually, we have such uses in English, 'fast row' is not the same as 'row fast' and 'eight pieces' is very different from 'pieces of eight').

C++ is burdened with numerous contextual meanings inherited from a time when we struggled to avoid introducing new keywords. The outstanding example is **static** but even our operators suffer from multiple meanings (e.g the * in ***x** is not the same as the * in **x*y**).

Now the surprising thing (to some) is that we do not actually need many built-in operators, though doing without them would be tedious in the extreme.

Let me hit the metal. Down deep, our programs turn into operations using logic gates. Those who have studied mathematical logic, or the basics of hardware design, know that all we need is a single type of logic gate. We can choose a 'nand gate' which takes two inputs and outputs 'false' only if both inputs are true or a 'nor gate' which outputs true only if both inputs are true. I am going to focus on a nand gate.

We can create the software equivalent with the following function:

```
bool nand (bool b1, bool b2) {
  if(b1) {
    if(b2) return false;
  }
  return true;
}
```

We can build all the other six standard logic gates using just nand gates.

```
bool not (bool b) { return nand (b, b); }
bool and (bool b1, bool b2) {
  return not(nand (b1, b2));}
bool or (bool b1, bool b2){
  return nand( not(b1), not(b2));}
bool nor (bool b1, bool b2) {
  return not(or(b1, b2));}
bool xor (bool b1, bool b2) {
  return and(or(b1, b2), or(not (b1),
  not (b2)));}
bool notxor(bool b1, bool b2){ not(xor(b1, b2));}
```

I know that I can define those six functions more simply but I am demonstrating that all the simple logic operators can be defined in terms of just one of them (you can use 'nor' instead of 'nand' as your elementary logic operator if you wish).

Now let me construct a half-adder in software (it adds two bits and returns the result along with a carry bit):

```
struct bits{
  bool s;
  bool c;
};
bits half-adder(bit b1, bit b2){
  bits r{xor(b1, b2), and(b1, b2)};
  return r;
}
```

From that we can make a full adder which can cope with three inputs, two bits and a carry bit:

```
bits adder(bit b1, bit b2, bit carry){
  bits r1{half-adder(b1,b2)};
  bits r2{half-adder(r1.s, carry)};
  bits r3{r2.s, or(r.c, r2.c)};
    // only one of the calls to half-adder can
    // set the carry flag
  return r3;
}
```

FRANCIS GLASSBOROW

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

Code Critique Competition 115 (continued)

Listing 3 (cont'd)

```
    std::cout << "Unchanged\n";
  }
  else
  {
    // update ...
    std::cout << "Some values were updated\n";
  }

  std::cout << "Reading new attributes\n";
  new_attributes = read_attributes(140);
```

```
  if (old_attributes == new_attributes)
  {
    std::cout << "Unchanged\n";
  }
  else
  {
    // update ...
    std::cout << "some values were updated\n";
  }
}
```

Listing 3 (cont'd)

Bookcase

The latest book review.

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

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

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

Astrid Byro (astrid.byro@gmail.com)

Clean Architecture: A Craftsman's Guide to Software Structure and Design

By R.C. Martin with contributions by James Grenning and Simon Brown, foreward by Kevlin Henney and afterword by Jason Gorman. Published by Pearson, ISBN-13 978-0-13-449416-6.

Review by Ian Bruntlett.

The backbone of this book is the design principles/guidelines that are organised into the acronym SOLID. They are:

- **S**RP: The Single Responsibility Principle.
- **O**CP: The Open-Closed Principle.
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

Overview

This book is split into 7 parts, spanning 35 chapters.

Part 1 is the introduction.

Part 2 covers programming paradigms and takes the position that, with Structured Programming, Object-Oriented Programming and Functional Programming all having been discovered, we have no more paradigms to discover.

Report on Challenge 5 (continued)

So far I have managed to avoid all the C++ operators. I have sidestepped assignment by using initialisation to capture the result of a function call but it gets increasingly difficult to work round not being able to modify an existing variable. With pure functional programming, I might be able to avoid assignment but that will not meet the needs of C++. So it seems that I must allow `operator=` as a primitive.

I also need to have some way to extract individual bits from a variable.

In hardware, the assignment operator is handled by `store` and the individual bits by the wiring of the cpu. So I am going to allow myself two operators as primitives. `=` will store a value on its right-hand side into the storage designated by the left-hand side. The index operator (`[]`) will allow extraction of individual elements of an array.

We are almost there. In order to use our adder to do arithmetic, we need a way to decompose a value into its constant bits and a way to recompose a value from its constituent bits. We can decompose an `unsigned int` by using a suitable array containing the values of the individual bits (1, 2, 4, 6, etc.) together with a bitwise and operator. The boolean value of:

```
value bitand bitvalues[n]
```

will give us the nth bit of `value`.

Similarly a bitwise or operator will allow us to set bit n of value.

The ability to extract a bit coupled with being able to set a bit allows us to create shift operators by extracting a bit and inserting it in a result value either one place up or one place down.

Given an adder, bitand, bitor, assignment and an index operator we can, albeit tediously, create all the logic and arithmetic operators. Creating the comparison operators can be managed (compare equal and compare not-equal are relatively straightforward, less than and greater than require rather more work).

I thought that that was about it and that four operators (assignment, index, bitor and bitand) would be sufficient. However, there is one murky corner where we need something more. I leave it to the reader to determine what that is and how it might be provided.

Challenge 6

I suppose I need to provide a more down to earth challenge that many of you will feel able to tackle.

Back in the 1970s – and even the early 1980s – most attempts at providing a perpetual calendar were flawed, even those written by professional programmers. A perpetual calendar is a program that will determine which day of the week any specific date falls on. For example, what was the day of the week for 25/12/1884 (or in ISO format, 1884-12-25 as used by the Japanese, which has the great advantage that dates in that format are simple to sort)?

Numerous flaws trapped the unwary programmer, such as the definition of a leap year. A student of mine in 1979 wrote a Basic program that worked correctly for all dates given four inputs, day, month, year and Julian or Gregorian. The month could be given as either a number from 1 to 12 or as a name. It tolerated names spelt with mixed case. If you mis-spelled the name it would guess based on the first three letters and ask for confirmation. If the day was impossible it would issue an error message (e.g. 'February does not have 29 days in 1900 using a Gregorian Calendar.' or 'There are never 31 days in April')

Nothing that hard but he did it in 33 statements in Basic. At the time it was a phenomenal effort though the local moderators marked it down because it wasn't, in their opinion, long enough. Fortunately the chief examiner understood quality as opposed to length.

So your challenge is to write the shortest program that you can that takes as input a date and a choice as to whether to use a Julian or a Gregorian calendar. You are not allowed to use any library facilities for time.

View from the Chair

Bob Schmidt
chair@accu.org

ACCU 31st Annual General Meeting

ACCU's 2019 Annual General Meeting will be held Saturday 13th April, 2019 at the Bristol Marriott City Centre Hotel, in conjunction with ACCU's annual conference [1]. Although the agenda has not yet been set, the critical dates leading up to the AGM are shown in the table on the right.

These dates have been posted to the web site, which will be updated with the agenda as it becomes available. The time of the meeting will be announced as soon as it has been set.

Important dates for the ACCU AGM 2019

Event	Date	Countdown
Announce date	13 January 2019	AGM - 90
Nomination/proposal deadline	12 February 2019	AGM - 60
Draft agenda	2 March 2019	AGM - 42
Agenda freeze	16 March 2019	AGM - 28
Voting opens	23 March 2019	AGM - 21
AGM	13 April 2019	

BSI Standards Awards

The committee was notified by BSI that our very own Francis Glassborow has been nominated for a 2018 BSI International Standards Maker Award [2]. Here's the text of the letter:

> To whom it may concern:
>
> I am not certain if you are the best person / this is the best e-mail at your organization to contact about this news but I hope that, if you are not, you would be so kind as to forward my e-mail to the appropriate person. One of your members and / or colleagues has asked us to let you know that they were nominated for a 2018 BSI Standards Award. Francis Glassborow was nominated for an International Standards Maker Award.
>
> The BSI Standards Awards are the most prestigious awards BSI gives to standards makers and were created to recognize outstanding contributions by emerging standards-makers, industry leaders, consumer champions, international standards-makers and committee chairs and secretaries. In addition, with the special Wolfe Barry Medal, BSI recognizes exceptional commitment and contribution to standards-making by an individual. Through their services to standards-making, BSI Standards Awards nominees are helping their sectors to develop and grow and helping to safeguard consumers. Their nomination for an award is a form of recognition and appreciation in and of itself and so we thought you would be interested to learn of it.
>
> […]
>
> Kind regards,
>
> Stephanie Eynon

Please join me in congratulating Francis for his nomination.

The awards ceremony was held 29 November, 2018. I sent an email to Ms. Eynon requesting she provide us with a list of the winners, but she had not responded as of the deadline for this issue of *CVu*. I will provide the results in the next *CVu*.

Late edit: Unfortunately, Francis did not win his category. I'll try to have more information in the next *CVu*.

ACCU Committee

As listed above, nominations for ACCU committee positions are due by 12 February, 2019. (This will be the last issue of *CVu* prior to the nomination deadline.) Recall from the last issue that we have two executive committee positions that we know need to be filled – Treasurer and Chair – and all other positions will be open for new candidates. Please contact me or any other committee member if you are willing to stand for any of the positions.

Best ACCU Magazine Articles 2018

Voting is now open for the selection of the Best ACCU Magazine Articles from 2018. Go to Survey Monkey UK [3] to cast your vote for up to three articles from each magazine. The winner from each magazine gets an additional 15 minutes of fame in an upcoming issue, and bragging rights for a year. Vote now!

References

[1] ACCU 2019: https://conference.accu.org/
[2] BSI Standards Awards: https://www.bsigroup.com/en-GB/about-bsi/uk-national-standards-body/BSI-Standards-Awards/
[3] Survey Monkey: https://www.surveymonkey.co.uk/r/HNP773J

Books (continued)

Part 3 goes into detail about SOLID design principles and this covers things I am more familiar with.

Part 4 is about design principles for dealing with components, something I am unable to comment on.

Part 5 is all about architecture and will bear reading and re-reading.

Part 6 takes the view that the database, the web, and frameworks are all low-level details which you should isolate yourself from, using abstractions.

Part 7 – the Appendix – is a collection of interesting anecdotes relating situations that reinforces the conclusions of this book.

I do have some alternative SOLID acronyms – wide eyed Software Engineers might think of it as **S**ystem **O**f **L**everaging **I**nteresting **D**esigns and hard-bitten consultants might think of it as **S**ystem **O**f **L**everaging **I**mmense **D**osh. Other people have discussed SOLID – help from accu-general prompted me to look at Chris Oldwood's article 'KISSing SOLID Goodbye' in *Overload* 122 and to watch Kevlin Henney's 'SOLID Deconstruction' presentation and this helped me adopt a less prescriptive approach to SOLID.

Things I gained from this book:

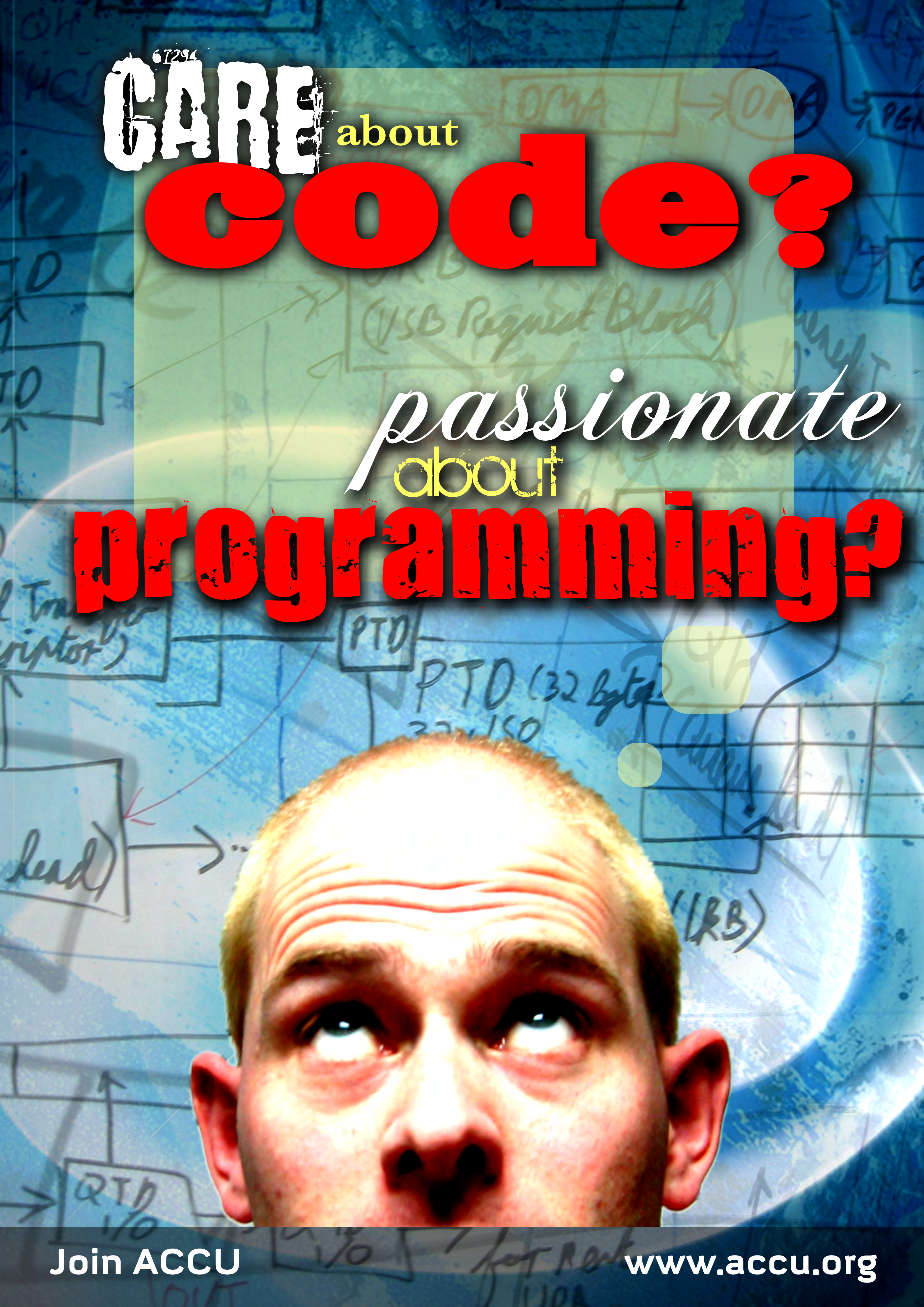
- Better understanding of the SOLID design principles / guidelines.
- Introduction to the Clean Architecture.

Downsides:

- Wedded to component architecture.
- Some of the structured diagrams are a bit too abstract for me – I would have appreciated something in addition – perhaps an explanation and a CRC card or two.

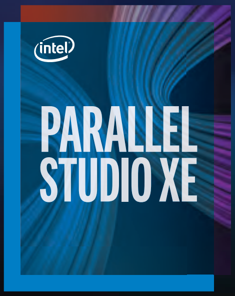
Conclusion

I'm going to have to read this book again, after a suitable period of doing other things. I particularly liked the contributions made by other authors. This is a thought provoking book that discusses the importance of coding, design, and architecture. Well worth reading – repeatedly.


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

intel Software

CODE MAXIMIZED



from £510

#HighPerformance

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

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.
© Intel Corporation

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

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

qbs SOFTWARE