the magazine of the accu

www.accu.org

{cvu}

Volume 29 • Issue 3 • July 2017 • £3

Features

Regulars

JOIN THE ACCU!

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

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

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

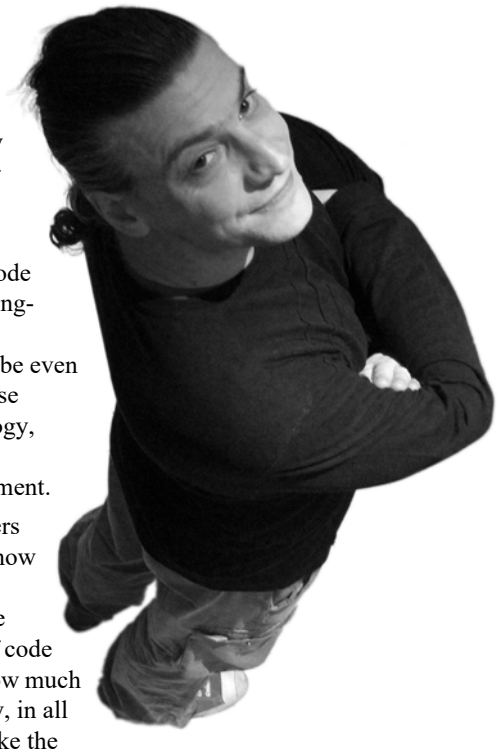
PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

{cvu}
Volume 29 Issue 3
July 2017
ISSN 1354-3164
www.accu.org

accu

It's written, that's why



Computer programs take many forms, from the very small, probably single person hobby projects, to the mind-bogglingly large enterprise multi-tiered distributed applications. It's rare for any program to be a write-once, deploy, forget affair, but it's probably fair to say that the larger applications tend to have a longer lifetime, if only because they represent so much investment and effort.

We've not yet reached the stage where source code no longer requires human understanding, so a long-lived program needs frequent intervention by programmers: adding features, fixing bugs, maybe even improving the code. All these things require those programmers to understand not just the technology, programming languages and so on, but also the conventions used during the program's development.

There are a whole slew of things we programmers do 'by convention'. C++ compilers do not care how many types you define in a given file, but it's conventional (although not universal) for it to be one. C# compilers don't care how many lines of code per method. Javascript is famously tolerant of how much code you can fit on a single line. Conventionally, in all these cases, it is considered good practice to make the code understandable.

Some conventions can end up being more of a hindrance than a help, however. I've written before about the 'I' prefix for interface types. Others that I find unhelpful are things like namespace hierarchies encoding corporate structure – especially if the directory structure on disk has to match – and always pairing getters with setters. These conventions all, I suspect, began with good intentions, but add nothing genuinely useful, and have become so common that they take on the nature of 'rules' rather than 'guidelines'.

If we take a default position of not flying in the face of convention, we run the risk of introducing unnecessary complexity almost *by accident*. Instead we should look critically at the things we do 'by convention', and only do them 'by intention'.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 29.3: 1st August 2017
C Vu 29.4: 1st October 2017

Overload 140: 1st September 2017
Overload 141: 1st November 2017

WRITE FOR C VU

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

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

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Living Within Constraints

Pete Goodliffe constrains what's possible in your code.

We have already considered defensive programming techniques (in *CVu* 29.1) to help make your code better. These techniques force you to consider what might go wrong – to assume the worst. So how can we physically incorporate these assumptions into our software so they're not elusive problems waiting to emerge? Clearly, we can simply write a little extra code to check for each condition. In doing so, we're codifying the *constraints* on program functionality and behaviour.

Constraints

The most obvious implementation of constraint checking in C and C++ is the humble **`assert`**. We'll look at it in detail later, but it's simple to use:

```
assert(itemIndex < maxNumItems);
```

When running a program with assertions enabled (usually this has to be within a 'debug' configuration build) a failure to satisfy the assertion will force the program to *unceremoniously* stop. Immediately. The program will dump out some diagnostic information about the assertion failure. But it will do so in an ugly way the programmer will understand. It's definitely not a nice user experience!

`assert` is simple, but can sometimes be a sledgehammer used to crack a walnut. There's no subtly; you cannot control what it does (apart from switch it on or off).

In a mature system, what *do* we want the program to do if a constraint is broken?

Since this kind of constraint will likely be more than a simple detectable and correctable run-time error, it must be a flaw in the program logic. There are few possibilities for the program's reaction:

- Turn a blind eye to the problem, and hope that nothing will go wrong as a consequence.
- Give it an on-the-spot fine and allow the program to continue (e.g., print a diagnostic warning or log the error).
- Go directly to jail; do not pass go (e.g., abort the program immediately, in a controlled or uncontrolled manner – either allowing it to neatly clean up if possible, or just exiting like assert).

Indeed, you may have different types of constraint condition that require different ways of handing.

For example, it is invalid to call C's **`strlen`** function with a string pointer set to zero, because the pointer will be immediately dereferenced, so the latter two options are the most plausible. It's probably most appropriate to abort the program immediately, since dereferencing a null pointer can lead to all sorts of catastrophes on unprotected operating systems.

Checking a value is 'sane' before displaying it on the screen might not need such a brute force approach.

Many debug/logging libraries provide constraint checking services. Generally they are configurable and flexible. Constraint checking is often bound in logging libraries because often you want to log a broken constraint, but allow the application to continue nevertheless.

How to use constraints

There are a number of different scenarios in which constraints are used:

- **Preconditions** These are conditions that must hold true *before* a section of code is entered. If a precondition fails, it's due to a fault in the client code.
- **Postconditions** These must hold true *after* a code block is left. If a postcondition fails, it's due to a fault in the supplier code.
- **Invariants** These are conditions that hold true every time the program's execution reaches a particular point: between loop passes, across method calls, and so on. Failure of an invariant implies a fault in the program logic.
- **Assertions** Any other statement about a program's state at a given point in time.

The first two are frustrating to implement without language support – if a function has multiple exit points, then inserting a postcondition gets messy. Eiffel supports pre- and postconditions in the core language and can also ensure that constraint checks don't have any side effects.

Good constraints expressed in code make your program clearer and more maintainable. This technique is also known as *design by contract*, since constraints form an immutable contract between sections of code.

Without built-in language-level constraint checking, the code mechanism used to check each of type of constraint is usually identical, and the type of constraint is just implicit from its location in the code.

What to constrain

There are a number of different problems you can guard against with constraints. For example, you can:

- Check all array accesses are within bounds.
- Assert that pointers are not zero before dereferencing them.
- Ensure that function parameters are valid.
- Sanity check function results before returning them.
- Prove that an object's state is consistent before operating on it.
- Guard any place in the code where you'd write the comment *We should never get here*.

The first two of these examples are particularly C/C++ focused. Other languages have their own ways of avoiding some of these pitfalls.

Just how much constraint checking should you do? Placing a check on every other line is a bit extreme. As with many things, the correct balance becomes clear as the programmer gets more mature. Is it better to have too much or too little? It is possible for too many constraint checks to obscure the code's logic. Readability is the best single criterion of program quality: If a program is easy to read, it is probably a good program; if it is hard to read, it probably isn't good.

Realistically, putting pre- and postconditions in major functions plus invariants in the key loops is sufficient.

Removing constraints

This kind of constraint checking is usually only required during the development and debugging stages of program construction. Once we have used the constraints to convince ourselves (rightly or wrongly) that the program logic is correct, we would ideally remove them so as not to incur an unnecessary runtime overhead.

Thanks to the wonders of modern technology, all of this is perfectly possible. As we've already seen, the C and C++ standard libraries provide a common mechanism to implement constraints – **`assert`**. **`assert`** acts

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

In Java I Wish I Could...

Paul Grenyer wishes for features of one language in another.

I started programming in BBC Basic on an Acorn Electron in 1985. I then went on to learn and use commercially C, C++ (there's no such language as C/C++), C# and Java. When I was a C++ programmer, I looked down on Java with its virtual machine, just in time compiling and garbage collector. When I became a Java programmer, I completely fell in love with it and its tool chain. Not so with Ruby, especially its tool chain, a lack of a static type system and lack of interfaces.

However, there are some fantastic features in the language and a few of them I wish I could use in Java. For example, in Ruby, you can put conditional statements after expressions, for example:

```
return '1' if a == 1
return '2' if a == 2
```

whereas in Java you'd have to write:

```
if (a == 1)
 return "1";
if (a == 2)
  return "2";
```

which is more verbose and less expressive.

Ruby also has the **unless** keyword, which Java lacks, so in Ruby you can do this:

```
return @colour unless @colour.nil?
```

The example shows off another feature in Ruby. To test for **nil** in Ruby you can call **.nil?** on any object, whereas the equivalent null check in Java is more verbose:

```
if (colour != null)
  return colour;
```

I could go on, but I'll leave that for a later piece in the series. These features of Ruby may only be, in the main, syntactic sugar, but they are the ones I miss most when I'm developing in Java. ■

PAUL GRENYER

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com

Living Within Constraints (continued)

as a procedural firewall, testing the logic of its argument. It is provided as an alarm for the developer to show incorrect program behaviour and should not be allowed to trigger in customer-facing code. If the assertion's constraint is satisfied execution continues. Otherwise, the program aborts, producing an error message looking something like this:

```
bugged.cpp:10: int main(): Assertion "1 == 0"
failed.
```

assert is implemented as a preprocessor macro, which means it sits more naturally in C than in C++. There are a number of more C++-sympathetic assertion libraries available.

To use **assert** you must **#include <assert.h>**. You can then write something like **assert(ptr != 0);** in your function. Preprocessor magic allows us to strip out assertions in a production build by specifying the **NDEBUG** flag to the compiler. All **assert**s will be removed, and their arguments will not be evaluated. This means that in production builds **assert**s have no overhead at all.

Whether or not assertions *should* be completely removed, as opposed to just being made non fatal, is a debatable issue. There is a school of thought that says after you remove them, you are testing a *completely different* piece of code. Others say that the overhead of assertions is not acceptable in a release build, so they must be eliminated. (But how often do people profile execution to prove this?)

Either way, our assertions must not have any side effects. What would happen, for example, if you mistakenly wrote:

```
int i = pullNumberFromThinAir();
assert(i = 6); // hmm - should type more
               // carefully!
printf("i is %d\n", i);
```

The assertion will never trigger in a debug build; its value is 6 (near enough *true* for C). However, in a release build, the **assert** line will be removed completely and the **printf** will produce different output. This can be the cause of subtle problems late in product development. It's quite hard to guard against bugs in the bug-checking code!

It's not difficult to envision situations where assertions might have more subtle side effects. For example, if you **assert(invariants());**, yet the **invariants()** function has a side effect, it's not easy to spot.

Since assertions can be removed in production code, it is vital that only constraint testing is done with **assert**. Real error condition testing, like memory allocation failure or file system problems, should be dealt with in ordinary code. You wouldn't want to compile that out of your program! Justifiable run-time errors (no matter how undesirable) should be detected with defensive code that can never be removed.

Java has a similar **assert** mechanism, which throws an exception (**java.lang.AssertionError**) instead of causing a program abort. .NET provides an assertion mechanism in the framework's **Debug** class.

When you discover and fix a fault, it is good practice to slip in an assertion where the fault was fixed. Then you can ensure that you won't be bitten twice. If nothing else, this would act as a warning sign to people maintaining the code in the future.

A common C++/Java technique for writing class constraints is to add a single member function called **bool invariant()** to each class. (Naturally this function should have no side effects.) Now an **assert** can be put at the beginning and end of each member function calling this invariant. (There should be no assertion at the beginning of a constructor or at the end of the destructor, for obvious reasons.) For example, a **circle** class invariant may check that **radius != 0**; that would be invalid object state and could cause later calculations to fail (perhaps with a divide by zero error). ■

Questions

- How much constraint checking do you employ in your codebase?
- Are there some functions that benefit more from pre/post condition checking? Why?
- How can you ensure that the logic in a constraint expression will have no observable affect on the program's behaviour?

Rip It Up and Start Again

Jez Higgins shares a tale of re-implementing a software system.

I've just wrapped up a hectic couple of weeks for a client and realised I'm now in a position to write one of those 'We reimplemented a piece of software originally written in language X using language Y and you won't believe what happened next' articles. You might not believe it but you can probably guess it, because articles like that are 10-a-penny and the reported results are always that some problem with the original code was solved in the most wonderful way. That problem could be speed, maintainability, memory use, lack of suitable job applicants, almost anything, but you can be sure it was resolved by the simple expedient of chucking out a load of code and writing some new code. The new code is invariably shorter and more expressive, somehow lacking the cruftiness of the old. The author invariably had lots of fun writing the new code, and often declares their chosen language to be their favourite. Frequently the article conclude with a little coda encouraging others to follow suit, or roundly insulting those so blinkered they choose not to.

This is not that article.

It is the case that I replaced a piece of production code developed by several people over a number of months in nine days, by myself. The new code is visibly faster, in the way that people notice without having to run benchmarks and draw graphs. It's leaner too – the VM it deployed to has half the CPU and a quarter of the memory of the previous version. It doesn't time out, exceed its memory quota, or crash.

I must be ace on the old keyboard, right?

Well, perhaps. But if there's any lesson to be learned here, it's that reimplementing a system is generally substantially easier than building it the first time round. In this case, a data access layer for a website, the existing code told me exactly which endpoints I needed to support, and which parameters they took. The productions logs told me which of those endpoints were actually being used, and with which parameter combinations. Almost straightaway, therefore, I knew where to concentrate my efforts, and what I could ignore completely. The existing code had a reasonable set of unit tests, which saved me the bother of having to come up with my own. I was able to look at the existing database queries, and translate them across into my new code. I could examine how the existing code had a addressed a particular issue and evaluate whether I wanted to bring that into the new code. I had the ridiculous luxury of being able to run up the old code and my new code side by side, prod them with the same requests, and compare the results, one with the other.

In short then, I benefited hugely from the fact that the system already existed. There was a just a ton of stuff I had to think a whole lot less about, and quite a lot I didn't have to think about at all. Consequently, I was able spend more of my effort thinking about the things that did matter, and the result, unsurprisingly, was less code.

While I rarely suggest an all-out rewrite, I did one here because I was asked to. The language I worked in wasn't my choice, it was the client's. The new code is quicker, not because of any special property of that language, but because I was able to look at the old code, see the inefficiencies, and *was given time and space to do something about it*. If they'd chosen another language, the results would have been similar. Hell, if they'd chosen the **same** language as before, the results would have been similar. If I'd been left to work on the original code for the same amount of time, I probably could have produced something similar [1].

Even when you're throwing it out, there's a lot of value in old code. Be grateful for it. ■

Note

[1] Given their lack of faith in the existing code, I suspect that if I'd suggested just letting me have at it, I may well have been turned down. Had that suggestion been accepted, I probably would have only been given half as much time.

JEZ HIGGINS

Jez is the 2017 Player of the Season for Kings Heath Hockey Club Mens IIIs – the high point of a moderate sporting career. When not playing hockey, he writes software for money and for fun. Contact him at jez@jezuk.co.uk or @jezhiggins.

Learning Other Languages

Francis Glassborow considers natural and computer languages as tools of communication.

I have this theory (actually it is a bit more than just a theory) that learning other languages improves my ability to use the language of my choice. This does not just apply to natural languages but to computer languages as well.

To get the greatest benefit one needs languages as different from ones first choice as possible. As a native English speaker learning French is of relatively little benefit. But learning Arabic, Chinese or a version of Sign has considerable impact. You do not need to be fluent in another language to benefit from studying it.

Let me take my three exemplars.

Arabic is fascinating in that its grammatical structure is radically different to English. It has only two tenses (completed and ongoing action) but seven moods including the very emphatic mood. In Spoken English we can use tone to emphasise and in the printed form we can use typeface but before printed writing became almost universal we had a problem that was solved by the use of 'particles' such as the classic biblical verses that start 'verily, verily' which is an attempt to translate the very emphatic mood that Hebrew shares with Arabic.

There is a great deal more to Arabic such as its use of consonants to convey root meaning and vowel structures to create nouns, verbs, adjectives, etc. As context often defines the part of speech written Arabic often omits many of the vowels. That makes it hard for the novice to read the local newspaper and much easier to practice on religious works where vowels are never omitted.

Because of the way vowels are used, rhymes are very easy in Arabic, so classical Arabic poetry usually requires very extensive triple rhyme schemes.

Written Chinese exhibits one of the great advantages of an ideographic language: you do not need to be able to speak it to read it. However, all the various spoken languages that come under the heading 'Chinese' share a common facet: they are very deficient in phonemes. This makes them rich languages for puns. That means that not being able to speak Mandarin, Cantonese, etc. detracts from your ability to appreciate Chinese poetry that relies heavily on puns.

I sometimes hear people opining that ideographic languages cannot have dictionaries. Well, I have a perfectly good Chinese dictionary somewhere on my book shelves. To use a dictionary for an alphabetic language you need to know the order of the alphabetic symbols (including how to deal with accents, etc.) For example, a German dictionary needs to have rules to deal with various idiosyncrasies of the German symbols.

In the case of Chinese, you need to be able to cope with two concepts: 'The master stroke' and the count of strokes (which if memory serves me correctly, can be anything from 1 to 17). The master stroke tells you which part of the dictionary to look in and the number of strokes takes you to the correct subsection. You may then have to look a little to find the one you want.

The group of languages under the heading 'Sign' Because of the extra dimensions brought about by the use of space, facial expression, etc., a Sign language is capable of shades of expression that spoken languages lack. However, it is almost impossible to represent them in writing. In the modern era, video has opened up their potential.

How does one produce poetry in Sign? I have no idea but I can hazard a guess that the visual representation would play a key part. I am reminded of Chinese calligraphy, so perhaps there is or should be an equivalent in Sign.

I wish that every school taught a version of Sign to every child. It is useful for communication across crowded rooms, noisy factories and other places where speech is difficult or frowned upon.

I could write a great deal more on natural languages and why I find the work of Noam Chomsky unsatisfying but let me move on.

Computer Languages

I can remember one Saturday when I had been commissioned by the local School Sailing Warden to write some race control software to run on his BBC Micro. It took me a couple of hours (the job was not that complicated). It had to be written in BBC Basic because that is what he had available, not a language with which I was enamoured. There were far better dialects of Basic. At the end the client commented that the code did not look like Basic. He was absolutely correct. The internal design and structure was Forth, the language in which I was most fluent in the 1980s. Had I tried the task by thinking in Basic it would certainly have taken me much longer and it would have been a much larger program. Yes, I was pretty fluent in Basic but it was the synergy of Forth with Basic that enabled me to work quickly and effectively.

This is just one example of the benefits of being multi-lingual. However the great benefit comes from having some familiarity with seriously different languages. It is to my eternal regret that I never managed to master any dialect of Lisp. I simply failed to make that fundamental step from procedural/functional type thinking.

What I am interested in is the fundamental design and construction of computer languages. I can use ideas from languages such as Prolog or Haskell, to name but two, to extend my skills in other languages in which I may be superficially more fluent.

The current crop of popular languages are, in my mind, too similar to each other to create any great benefit from studying them. Is Java better than C++? Is Ruby better than Python? I simply do not care. What I want to know is what do each of those languages teach my about the wider art of programming that makes it worth my while to invest time in studying them even if I do not actually master them or use them with a vengeance.

Studying Haskell has extended my understanding of C++ template metaprogramming. And actually it is much easier to do the design thought in Haskell and then implement it with templates.

My challenge to the readers of *CVu* is to write an article (as long as necessary but no longer) that tells me what your language(s) of choice have to reveal about the art of programming. I do not want to know why they are better but I want to know why they are different. I want to understand what they offer. For example, why are pure functions beneficial in problem solving? What is wrong with globals? Can you actually write real, useful code that is free of side-effects?

I do not want a better X where X is any language you like to name. I want a better understanding of the wider art of programming and how X can contribute to my understanding.

Over to you, reader. I am certain that you have insights worth sharing. ■

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++*.

A Magical New World?

Samathy Barratt shares her experience as a first time ACCU Conference attendee.

I went to my first ACCU Conference last week. It was great. I'd heard about ACCU from Russel Winder several months ago. He recommended I check out the conference (for which he's on the programme board) since I'm a fan and user of the C and C++ languages.

Russ Miles opens ACCU 2017

I arrived in Bristol on Tuesday excited for what the week held. We started the conference proper with a fantastically explosive keynote delivered by Russ Miles, who jumped on stage to deliver a programming parody of Highway to Hell accompanied by his own guitar playing. His keynote was all about modern development and how most of a programmer's tools currently just shout information at the programmer, rather than actually helping.

Later on the Wednesday, I headed into a talk from Kevlin Henny that totally re-jigged how I think about concurrency. Thinking outside the Synchronisation Quadrant was wonderfully entertaining, with Kevlin excitedly bouncing across the floor.

Wednesday's talks continued with several other good talks and a number of great lightning talks too. Finalising with the welcome reception where delegates gathered in the hotel for drinks, food and conversation. It was here that I really got the chance to socialise with a good few people, including Anna-Jayne and Beth, who I'd been excited about meeting since I found out they were going to be there!

Thursday began with an interesting keynote about the Chapel parallel programming language. The talk has encouraged me to try the language out and I'll certainly be having a good play with that soon.

Thursday's stand out talks included Documentation for Developers workshop by Peter Hilton. I really enjoyed the workshoppy style that Peter used to deliver the talk. He got the audience working in groups, talking to each other and essentially complaining about documentation. He finished with suggesting a method of writing docs called Readme Driven Development as well as other suggestions.

Lightening talks on Wednesday

The other talk on Thursday which I really loved was 'The C++ Type System is your Friend'. Hubert Matthews was a great speaker with clear experience in explaining a complex topic in an easier to understand fashion. I can't say I understood everything, but I certainly liked listening to Hubert speak.

Peter Hilton's 'Documentation for Developers workshop'

Thursday evening I headed out for dinner with Anna-Jayne and Beth before heading back to my accommodation to write up a last minute talk for Friday.

My talk was covering Intel Software Guard Extensions – Russel announced that there was an open slot on Friday for a 15 minute topic and I took the chance to speak then.

Odin Holmes talks about Named Parameters

Friday began with a curious but thought-provoking talk from Fran Buontempo called 'AI: Actual Intelligence'. I'm not entirely sure what the take away from the talk was intended to be, but nonetheless it was interesting!

Friday morning was full of 15 minute talks. A format I think is wonderful. I really loved that amongst the 90 min talks throughout the rest of the week, there was time for these quick fire shorter talks too that were still serious technical talks (unlike the 5min lightning talks). The talks I went to see were:

- 'The missing piece of the continuous integration puzzle – what to do with all those test failures?' – Greg Law
- 'Named Parameters' – Odin Holmes. Using Template magic to impliment named params in C++. I loved this talk and fully intend to investigate the idea!
- 'Passwords. Are. Not. Hard!' – Dom Davis. A hilarious rant about how nonsensical password handling is.

At Friday lunch time I took part in a bit of an unplanned workshop on sketch noting with Michel Grootjans. It was essentially an hour of trying to make our notes prettier! It was a lot of fun.

Friday was the conference dinner – a rock themed night of fun and frivolities. This was by far the high point of the conference for me. It offered a great evening of meeting people and having a lot of fun. I loved how everyone loosened up and spoke to anyone else there.

I met a whole bunch of people, and got on super well with a few people who I would like to consider friends now.

ACCU made it easy to get to know people too by forcing everyone who isn't a speaker to move tables between each meal course. It's a great idea!

Saturday's talks started with a really fun talk from Arjan van Leeuwen about string handling in C++1x. Covering the differences between `char`

SAMATHY BARRATT

Samathy is a magical code fairy with a passion for tackling really hard problems, teaching others and supporting a diverse and friendly tech community. She enjoys C, C++, Python, Linux, Coffee and people. She tweets @samathy_barratt and can be contacted at samathy@sbarratt.co.uk.


Sketch Noting with Michel Grootjans

arrays and **std::string**s and how best to use them. As well as tantalising us with a C++17 feature called **std::string_view** (immutable views of a string).

Later I watched a talk from Anthony Williams and another from Odin, both of which went wildly over my head, but all the same I gained a few things from both of them.

Finishing off the conference was a brilliant keynote from renowned speaker and member of the ISO C++ standards committee, Herb Sutter.

Herb Sutter on Metaclasses at ACCU 2017

Herb introduced a new feature of C++ that he may be proposing to the standards committee.

He described a feature allowing one to create meta-classes.

Essentially, one could describe a template of a class with certain interfaces, data and operators. Then, one could implement an instance of that class defining all the functionality of the class. Its essentially a way to more cleanly describe something akin to inheritance with virtual functions.

I highly suggest you try to catch the talk, since it was so interesting that even an hour or so after the talk, there was still quite a crowd of people gathered around Herb asking him questions.

Herb Sutter on Metaclasses

The conference environment

As a first time ACCU attendee, I want to say a few words about the environment at the conference.

As most of the readers of my blog[1] know, I'm a young transwomen, so a safe and welcome environment is something that I very much appreciate and makes a huge difference to my experience of an event.

It's something that's super hard to achieve in a world like software development where the workforce is predominantly male.

I'm glad to say that ACCU did a great job of creating a safe and welcoming space. Despite being predominantly male as expected, everyone I encountered was not only friendly and helpful, but ever so willing to go out of their way to make me feel welcome and comfortable. Everyone I met simply accepted me for me and didn't treat me any way other than friendly.


Herb Sutter surrounded by curious programmers


Odin enjoys inflatable instruments

I would suggest that offering diversity tickets to ACCU would help make me feel even better there, since I'd feel better with a more diverse set of delegates.

I was especially comforted by Russel mentioning the code of conduct, without fail, every day of the conference. As well as one of the lightning talks, being delivered by a man, taking the form of a spoke word-ish piece praising the welcoming nature of ACCU and calling for the maintenance of the welcoming nature to all people in the community, not just people like himself.

I'd like to especially mention Julie and the Archer-Yates team for checking up on my happiness throughout the conference. They really helped me feel safe there.

I think there still could be work to do about making the conference a good place for younger adults – I was rather overwhelmed by the fact that everyone seemed older than me and clearly had a better idea of how to conduct themselves in the conference setting. However, I think the only real way of solving this problem would be to make the conference easier to access to younger people (cheaper tickets for students – it's still super expensive) which wouldn't always be possible. Additionally, the inclusion of some simpler, easier to understand talks would have been great. Lots of the talks were very complicated and easily got to a level that was way over my head.


A presentation is underway...

Thanks to everyone who helped me feel welcome at ACCU – including but not limited to Richard, Antonello, Anna-Jayne, Beth, Jackie, Fran, Russel and Odin.

In conclusion

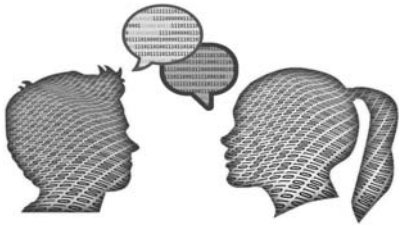
ACCU was a fantastic experience for me. I would highly recommend it to anyone interested in improving their C and C++ programming skills as well as general programming skills. I'll certainly be heading back next year if I can, and am happily a registered ACCU member now! ■

Reference

[1] My blog: https://medium.com/@samathy_barratt?source=post_header_lockup

Code Critique Competition 106

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



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

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

Last issue's code

I was writing a simple program to analyse the programming languages known by our team. In code review I was instructed to change `languages()` method in the programmer class to return a `const` reference for performance. However, when I tried this the code broke. I've simplified the code as much as I can: can you help me understand what's wrong? It compiles without warnings with both MSVC 2015 and gcc 6. I am expecting the same output from both 'Original way' and 'New way' but I get nothing printed when using the new way.

The listings are as follows:

- Listing 1 is `programmer.h`
- Listing 2 is `team.h`
- Listing 3 is `team_test.cpp`

Listing 1

```
#pragma once
class programmer
{
public:
  // Add a language
  void add_language(std::string s)
  { languages_.push_back(s); }
  // the programmer's languages - original
  std::vector<std::string>
  original() const
  {
    return {languages_.begin(),
            languages_.end()};
  }
  // new - avoid copying for performance
  std::list<std::string> const &
  languages() const
  { return languages_; }
  programmer() = default;
  programmer(programmer const& rhs)
  : languages_(rhs.languages_) {}
  programmer(programmer &&tmp)
  : languages_(std::move(tmp.languages_)) {}
  ~programmer() { languages_.clear(); }
private:
  std::list<std::string> languages_;
};
```

Listing 2

```
#pragma once
#include <map>
class team
{
public:
  // Add someone to the team
  void add(std::string const &name,
    programmer const & details)
  {
    team_.emplace(
      std::make_pair(name, details));
  }
  // Get a team member's details
  auto get(std::string const &name) const
  {
    auto it = team_.find(name);
    if (it == team_.end())
      throw std::runtime_error("not found");
    return it->second;
  }
private:
  std::map<std::string, programmer> team_;
};
```

Listing 3

```
#include <iostream>
#include <list>
#include <map>
#include <string>
#include <vector>
#include "programmer.h"
#include "team.h"

int main()
{
  team t;
  programmer p;
  p.add_language("C++");
  p.add_language("Java");
  p.add_language("C#");
  p.add_language("Cobol");
  t.add("Roger", std::move(p));

  p.add_language("javascript");
  t.add("John", std::move(p));

  std::cout << "Original way:\n";
  for (auto lang : t.get("Roger").original())
  {
    std::cout << lang << '\n';
  }
  std::cout << "\nNew way:\n";
  for (auto lang : t.get("Roger").languages())
  {
    std::cout << lang << '\n';
  }
}
```


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

Critique

James Holland < james.holland@babcockinternational.com>

The problem with the student's code lies in the 'New way' range-based `for` loop. Calling the `get()` member function on `t` results in a temporary object of type `programmer`. The temporary object's `languages()` function is then called which returns an `std::list` of `std::string`s. The range-based `for` loop then effectively calls `begin()` on the `std::list` thus returning an iterator to the list. The iterator will be then compared with the end of the list to determine whether there are any items in the list. Unfortunately, it is around this time (and I must admit I am not exactly sure when) that the temporary `programmer` object will go out of scope, its destructor called and will no longer be valid. We are now in the realms of undefined behaviour. Anything could happen and we should not be too surprised if nothing is printed.

Obtaining a reference to `t`'s `programmer` object, instead of making a copy, will result in the range-based `for` loop referring to a valid list of strings and ultimately in the expected printed output. This can be achieved by amending team's `get()` function to return a reference to a `programmer` object. As the `t` object will remain valid while the `for` loop is executing, there is no need to make a copy.

I think there are other oddities with the student's code. The `std::move()` within the second parameter of the `t.add()` has no effect as the parameter is being passed by (`const`) reference. This means object `p` contains the same programming languages after the call to `t.add()` as before. This results in John possessing all the programming languages of Roger plus Javascript. This is probably not what was required. If `std::move(p)` is used, it must be assumed that `p` is left in a valid but unknown state. It may contain values or it may not. It is best to modify the `programmer` class to have a public `clear()` function. This function should then be called before repopulating `p` with another member of staff's programming languages.

Commentary

The main problem with this critique, as James explained, is holding a reference to a temporary after its destruction. This is, unfortunately, quite easy to do in C++ and the new-style range based `for` provides a potential site.

In C++ the statement

```
for (auto lang : t.get("Roger").languages()){}
```

is equivalent to this rewritten block of statements:

```
{
  auto &&__range = t.get("Roger").languages();
  auto __begin = __range.begin();
  auto __end = __range.end();

  for ( ; __begin != __end; ++__begin ) {
    auto lang = *__begin;
    {}
 }
}
```

Since `languages()` returns a `const` reference to an `std::list` this means `__range` in turn is a `const` reference to an `std::list` – but the target of the reference is member data of a *temporary* that is destroyed at the end of the first statement in the re-written code.

James' solution in this case, of making the `get()` function return a reference not a copy, solves the problem as `__range` now refers to an object owned by `t`, whose lifetime lasts after the end of the `for` loop, and so there is no longer a 'dangling reference'.

The root cause of the problem is the declaration of the method:

```
auto get(std::string const &name) const;
```

Somehow it seems the presence of `auto` partially disables the programmer's critical mind. Without `auto` the programmer, in my experience, is more likely to notice the implied copy in:

```
programmer get(std::string const &name) const;
```

and to have written the method to return a reference:

```
programmer const
  &get(std::string const &name) const;
```

In other cases, though, this may not be a possible solution (for example, the data returned by `get()` might be generated during the call). We can make the code safe by binding the temporary to a named variable that outlasts the `for` loop:

```
auto temp = t.get("Roger");
for (auto lang : temp.languages()) {}
```

One drawback with this solution is that the variable introduced remains in scope until the end of the enclosing block; it would be nicer to ensure it gets deleted at the end of the `for` loop. By a bit of a co-incidence, two proposals published this year would provide alternative ways of making the code safe. (Note that neither proposal has, as yet, been approved by the standards committee.)

Firstly Zhihao Yuan's proposal P0577R0 ('Keep that temporary!') would allow using the proposed '`register`' expression to extend the lifetime of the temporary to the end of the loop, using this syntax:

```
for (auto lang :
  (register t.get("Roger")).languages()){}
```

Secondly Thomas Köppe's proposal P0614R0 ('Range-based `for` statements with initializer') would allow using an initializer in a range `for` loop, using this syntax:

```
for (auto tmp = t.get("Roger");
     auto lang : tmp.languages()){}
```

This elegantly ensures the scope of the introduced variable ends at the end of the `for` loop.

Even if either or both of these proposals are accepted, the underlying problem of *indirectly* binding a reference to a temporary object remains untouched, and can be hard to diagnose. The problem occurs when chaining method calls where the *final* call returns a reference but one or more of the intermediate calls return a temporary. This is something a static analysis tool could in principle detect; does anyone know of an existing tool that does so?

The second problem with the code is the re-use of the moved-from `p` – James correctly explains the problem. In order for the `std::move()` to have any effect, the argument to `add()` would need to be either a value or an r-value reference, for example:

```
void add(std::string const &name,
  programmer details)
{
  team_.emplace(
    std::make_pair(name,
      std::move(details)));
}
```

(Note you also need to add `std::move()` when passing the `details` object to `emplace()`.)

However, it is still unspecified what the state of the object will be after the call, so it cannot be assumed to be empty.

The Winner of CC 105

After a full post-bag of six entries for CC104 I was disappointed to only receive one entry for CC105 – but it was a clear and concise critique of the code and so I have no qualms in awarding James this issue's prize.

Code Critique 106

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

I am learning some C++ by writing a simple date class. The code compiles without warnings but I've made a mistake somewhere as the test program doesn't always produce what I expect.

```
> testdate
Enter start date (YYYY-MM-DD): 2017-06-01
Enter adjustment (YYYY-MM-DD): 0000-01-30
Adjusted Date: 2017-07-31
>testdate
Enter start date (YYYY-MM-DD): 2017-02-01
Enter adjustment (YYYY-MM-DD): 0001-01-09
Adjusted Date: 2018-03-10
>testdate
Enter start date (YYYY-MM-DD): 2017-03-04
Enter adjustment (YYYY-MM-DD): 0001-00-30
Adjusted Date: 2017-04-03
```

That last one ought to be 2018-04-04, but I can't see what I'm doing wrong.

Please can you help the programmer find his bug – and suggest some possible improvements to the program!

- Listing 4 contains `date.h`
- Listing 5 contains `date.cpp`
- Listing 6 contains `testdate.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 4

```
// A start of a basic date class in C++.
#pragma once

class Date
{
  int year;
  int month;
  int day;

public:
  void readDate();
  void printDate();
  void addDate(Date lhs, Date rhs);
  bool leapYear();
};
```

Listing 5

```
#include "date.h"
#include <iostream>
using namespace std;

// Read using YYYY-MM-DD format
void Date::readDate()
{
  cin >> year;
  cin.get();
  cin >> month;
  cin.get();
  cin >> day;
}
// Print using YYYY-MM-DD format
void Date::printDate()
{
  cout << "Date: " << year << '-' <<
    month/10 << month%10 << '-' <<
    day/10 << day %10;
}

void Date::addDate(Date lhs, Date rhs)
{
  year = lhs.year + rhs.year;
  month = lhs.month + rhs.month;
  day = lhs.day + rhs.day;
```

Listing 5 (cont'd)

```
  // first pass at the day -- no months
  // are over 31 days
  if (day > 31)
  {
    day -= 31;
    month = month + 1;
    if (month > 12)
    {
      year += 1;
      month -= 12;
    }
  }
  // normalise the month
  if (month > 12)
  {
    year += 1;
    month -= 12;
  }
  // now check for the shorter months
  int days_in_month = 31;
  switch (month)
  {
  default: return; // done 31 earlier
  case 2: // Feb
    days_in_month = 28 + leapYear()?1:0;
    break;
  case 4:  // Apr
  case 6:  // Jun
  case 9:  // Sep
  case 11: // Nov
    days_in_month = 30;
  }
  if (day > days_in_month)
  {
    day -= days_in_month;
    month += 1;
    if (month > 12)
    {
      month -= 12;
      year += 1;
    }
  }
}
bool Date::leapYear()
{
  // Every four years, or every four centuries
  if (year % 100 == 0) return year % 400 == 0;
  else return year % 4 == 0;
}
```

Listing 6

```
#include "date.h"
#include <iostream>

using std::cout;

int main()
{
  Date d1, d2, d3;
  cout << "Enter start date (YYYY-MM-DD): ";
  d1.readDate();
  cout << "Enter adjustment (YYYY-MM-DD): ";
  d2.readDate();

  // Add the two dates
  d3.addDate(d1, d2);
  cout << "Adjusted ";
  d3.printDate();
}
```

View from the Chair

Bob Schmidt
chair@accu.org

2017 ACCU Conference

Our annual conference once again was held in the Bristol Marriott City Centre from the 26th through to the 29th of April. Russell Winder and his merry band of programme committee members [1] did a wonderful job of making the conference an educational and social extravaganza. Please join me in expressing our thanks to Russell and the programme [2] committee for their outstanding work. (Russell was particularly outstanding in his bright orange sweater!)

Thanks also to our Conference sponsors [1] for their support; and to Julie Archer and her team at Archer Yates Associates Ltd. [3]

Videos of most of the sessions are now available on ACCU's YouTube channel. Head on over to YouTube and check out the sessions you weren't able to attend. [4]

Annual General Meeting

ACCU held its Annual General Meeting on Saturday, the 29th of April, 2017, in conjunction with the 2017 ACCU Conference. On the agenda were approving the minutes from the 2016 AGM; annual reports from the officers; approval of the accounts for fiscal year 2016; election of auditors; and election of Officers and the Committee.

The minutes and the accounts were approved, and the results of the election were announced. All persons running for office were elected. The meeting pack has copies of the officers' annual reports and a list of persons who ran for election. [5] There were several positions for which nominations were not received prior to the deadline. The incumbents for each of these positions were re-elected by acclamation. [6] Thank you to all of you who took the time to vote.

Committee Spotlight

ACCU starts its new operating year with a fully staffed executive committee, and a mostly staffed non-executive committee. Current committee members are listed on our web site, so there is no need to duplicate that list here. [6][7]

The committee remains mostly unchanged from the previous year, which has its advantages and disadvantages. An advantage is the continuity that results from low or slow turnover. One large disadvantage is the absence of new people and their new ideas. We had people running for election for each of the executive positions, but we only had one person running for each position.

You don't have to be a committee member to work with the committee. If you have an idea that will benefit ACCU and your fellow members, please let one of us know. If you want to find out what being on the committee is like, any Member can attend committee meetings, where you can join in the conversation.

Call for Volunteers

Currently we have four vacancies in our roster of committee members. Our most immediate need is for one or more people to take over the role of web editor. As I mentioned in the last issue of *CVu*, long-time web editor Martin Moene has stepped down from the position as of 1 July, 2017. Prior to announcing his retirement Martin wrote a detailed description of the role. [8]

That description is quite daunting – I'm not sure any of us quite realized all that Martin has done for us during his years as web editor; thanks again, Martin! – and it may be more than any one person might want to commit to. We have discussed the idea of splitting the role amongst more than one person. If you want to contribute, but don't want to commit to everything in the description, please let me know. All help is welcome.

- The ACCU web site uses Xaraya, a PHP framework that has been moribund for the last 4 years at least, and a replacement is overdue. [9]
- The Publicity, Study Groups, and Social Media positions have been vacant for some time. [10]

Being on the Committee is an amazing opportunity to try doing something new. You are very welcome to start and just focus on making one thing happen, and in a way the fits with your family, professional and social activities. Just getting one thing done however small is better than the post being left vacant. Committee members will support you in any way they can.

Please contact me if you are interested.

Member News

Submitting a short piece of member news is your opportunity to share your successes with your fellow members, or make a call for volunteers to help with a project you may have. I will continue to send out a reminder on accu-members shortly before the *CVu* deadline.

On a personal note – with this column I have the honour and privilege of starting my second year as Chair of ACCU. Thank you for your votes. Now, what am I going to write about for the next 5 issues of *CVu*?

References

[1] ACCU Conference Programme Committee Members and Sponsors https://conference.accu.org/site/index.html

[2] Microsoft Word really does not like it when I spell 'programme' the British way, with the extra 'me'. I get the red squiggly line of bad-spelling shame. It's time to add it to the dictionary.

[3] Archer Yates Associates Ltd. https://conference.accu.org/site/index.html

[4] ACCU YouTube Channel https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw

[5] 2017 AGM Pack http://accu.org/content/agm/AGM-2017-Pack.pdf

[6] ACCU Committee for 2017-2018 https://accu.org/index.php/members/committee

[7] Listing all of the committee members here would be one way of padding the word count of this column.

[8] Role of Website Editor https://accu.org/index.php/members/committee/posts_and_roles#website_editor

[9] Ardent followers of this column (and that is all of you, hmm?) will notice that this item has been hanging around for some months now. Anyone out there know Xaraya and/or PHP and can lend a hand? Contact Jim Hague.

[10] ACCU Social Media https://plus.google.com/+AccuOrganisation https://www.facebook.com/accuorg/ https://twitter.com/accuorg

Member news

Silas Brown and co-authors published an open-access paper in 'Biology Methods & Protocols' (Oxford University Press), thereby giving biologically-qualified peer review to his free cancer-research tool ('Delivering Bad News From QA', *CVu*, 28(5):4–5, November 2016). He hopes other labs can now take the tool seriously.

Frances Buontempo has been talking and writing about how to program your way out of a paper bag, often using machine learning, for a while now.

She is pleased to announce she has now signed a deal to publish a book to pull together some of these ideas and add some new recipes. She would like to thank ACCU members for encouraging her and helping to proof read the pitch and sample chapter. Watch this space for more news.

"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