the magazine of the accu

www.accu.org

{cvu}

Volume 30 • Issue 3 • July 2018 • £4.50

Features

Write Less Code!
Pete Goodliffe

Don't Assume Any Non-Zero Exit() Will Fail!
Silas S. Brown

The Half-Domain/Half-Primitive Proxy
Chris Oldwood

Everyday Coding Habits For Safety and Simplicity
Arun Saha

ACCU18 Trip Report
Balog Pal

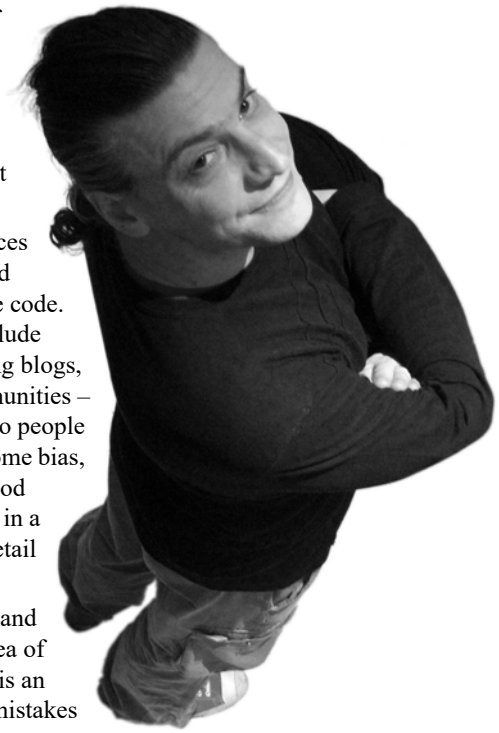
Regulars

Program Challenge
Code Critique
Members' Info

{cvu}
Volume 30 Issue 3
July 2018
ISSN 1354-3164
www.accu.org

accu

Into Gold



Software developers perform a kind of magic. There is quite a lot that goes into the act of creating working software. We pull in information and experience from a multitude of sources, filter out the bits that aren't useful to the problem in hand, mix up the bits that are (or at least, might be), and a program pops out at the end of the process. Right?

Well, almost. We certainly use a variety of sources of information, filter that through experience and (occasionally) experimentation, in order to write code. Those information sources vary greatly, and include things like formal education and training, reading blogs, magazines, books, taking part in different communities – online, and in the flesh – and even just chatting to people in the coffee shop or the pub. I suppose I have some bias, but I still find magazines like this one to be a good source because it often exposes me to new ideas in a succinct way, which I can find more elaborate detail on elsewhere.

Previous experience of writing software – good and bad – is also a source of information, and the idea of filtering raw data through a sieve of experience is an analogy that appeals to me. Learning from our mistakes is only one side of this, because hopefully there are positive things we remember from previous jobs, projects or teams that can inform us on the job in hand today.

But it's when you get a team of diverse people together in one place that the sparks of alchemy really start to fly. No two people have the *same* experience, or have read the *same* articles and books, or if they have, they will likely have not formed the *same* connections and opinions. Even disagreement can be the cause of innovation. Programming is a social activity, whatever image of the lone hacker people have in their minds when they hear the term 'computer programmer'.

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

17 Code Critique Competition
Code Critique 112 and the answers to 111.

22 Program Challenge Report 3 and Challenge 4
Francis Glassborow comments on his last challenge and presents a new one.

REGULARS

24 Members
Information from the Chair on ACCU's activities.

FEATURES

3 Write Less Code!
Pete Goodliffe helps us avoid unnecessary lines of code.

6 ACCU18 Trip Report
Balog Pal reports his experiences from the 2018 ACCU Conference.

8 Don't Assume Any Non-zero exit() Will Fail!
Silas S. Brown shares his finding on process exit codes.

9 Everyday Coding Habits for Safety and Simplicity
Arun Saha has some simple advice for forming good habits in C++.

12 The Half-Domain/Half-Primitive Proxy
Chris Oldwood presents a pattern for abstracting client-side proxies for testing.

SUBMISSION DATES

C Vu 30.4: 1st August 2018
C Vu 30.5: 1st October 2018

Overload 146: 1st September 2018
Overload 147: 1st November 2018

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

Write Less Code!

Pete Goodliffe helps us avoid unnecessary lines of code.

A well-used minimum suffices for everything.
~ Jules Verne, *Around the World in Eighty Days*

It's sad, but it's true: in our modern world there's just too much code.

I can cope with the fact that my car engine is controlled by a computer. There's obviously software cooking the food in my microwave. And it wouldn't surprise me if my genetically modified cucumbers had an embedded microcontroller in them. That's all fine; it's not what I'm obsessing about. I'm worried about all of the *unnecessary* code out there.

There's simply too much unnecessary code kicking around. Like weeds, these evil lines of code clog up our precious bytes of storage, obfuscate our revision control histories, stubbornly get in the way of our development, and use up precious code space, choking the good code around them.

Why is there so much unnecessary code?

Some people like the sound of their own voice. You've met them; you just can't shut them up. They're the kind of people you don't want to get stuck with at parties. *Yada yada yada.* Other people like their own code too much. They like it so much they write reams of it:

```
{ yada->yada.yada(); }
```

Or perhaps they're the programmers with misguided managers who judge progress by how many thousands of lines of code have been written a day.

Writing lots of code does *not* mean that you've written lots of software. Indeed, some code can actually negatively affect the amount of software you have – it gets in the way, causes faults, and reduces the quality of the user experience. The programming equivalent of antimatter.

> Less code *can* mean more software.

Some of my best software improvement work has been by removing code. I fondly remember one time when I lopped thousands of lines of code out of a sprawling system, and replaced it with a mere 10 lines of code. What a wonderfully smug feeling of satisfaction. I suggest you try it some time.

Why should we care?

So why is this phenomenon *bad*, rather than merely annoying?

There are many reasons why unnecessary code is the root of all evil. Here are a few bullet points:

- Writing a fresh line of code is the birth of a little life form. It will need to be lovingly nurtured into a useful and profitable member of software society before you can release a product using it.

  Over the life of your software system, that line of code needs maintenance. Each line of code costs a little. The more code you write, the higher the cost. The longer a line of code lives, the higher its cost. Clearly, unnecessary code needs to meet a timely demise before it bankrupts us.
- More code means there is more to read and more to understand – it makes our programs harder to comprehend. Unnecessary code can mask the purpose of a function, or hide small but important differences in otherwise similar code.
- The more code there is, the more work is required to make modifications – the program is harder to modify.
- Code harbours bugs. The more code you have, the more places there are for bugs to hide.
- Duplicated code is particularly pernicious; you can fix a bug in one copy of the code and, unbeknown to you, still have another 32 identical little bugs kicking around elsewhere.

Unnecessary code is nefarious. It comes in many guises: unused components, dead code, pointless comments, unnecessary verbosity, and so on. Let's look at some of these in detail.

Flappy logic

A simple and common class of pointless code is the unnecessary use of conditional statements and tautological logic constructs. Flappy logic is the sign of a flappy mind. Or, at least, of a poor understanding of logic constructs. For example:

```
if (expression)
  return true;
else
  return false;
```

can more simply, and directly, be written:

```
return expression;
```

This is not only more compact, it is easier to read, and therefore easier to understand. It looks more like an English sentence, which greatly aids human readers. And do you know what? The compiler doesn't mind one bit.

Similarly, the verbose expression:

```
if (something == true)
{
  // ...
}
```

would read much better as:

```
if (something)
```

Now, these examples are clearly simplistic. In the wild we see much more elaborate constructs created; never underestimate the ability of a programmer to complicate the simple. Real-world code is riddled with things like Listing 1, which reduces neatly to the one-liner:

```
return gorilla_is_hungry() && bananas_are_ripe();
```

Listing 1

```
bool should_we_pick_bananas()
{
  if (gorilla_is_hungry())
  {
    if (bananas_are_ripe())
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  else
  {
    return false;
  }
}
```

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

Cut through the waffle and say things clearly, but succinctly. Don't feel ashamed to know how your language works. It's not dirty, and you won't grow hairy palms. Knowing, and exploiting, the order in which expressions are evaluated saves a lot of unnecessary logic in conditional expressions. For example:

```
if ( a
  || (!a && b) )
{
  // what a complicated expression!
}
```

can simply be written:

```
if (a || b)
{
  // isn't that better? didn't hurt, did it?
}
```

> Express code clearly and succinctly. Avoid unnecessarily long-winded statements.

Duplication

Unnecessary code duplication is evil. We mostly see this crime perpetrated through the application of *cut-and-paste* programming: when a lazy programmer chooses not to factor repeated code sections into a common function, but physically copies it from one place to another in their editor. Sloppy. The sin is compounded when the code is pasted with minor changes.

When you duplicate code, you hide the repeated structure, and you copy all of the existing bugs. Even if you repair one instance of the code, there will be a queue of identical bugs ready to bite you another day. Refactor duplicated code sections into a single function. If there are similar code sections with slight differences, capture the differences in one function with a configuration parameter.

> Do not copy code sections. Factor them into a common function. Use parameters to express any differences.

This is commonly known as the *DRY* principle: *Don't Repeat Yourself*! We aim for 'DRY' code, without unnecessary redundancy. However, be aware that factoring similar code into a shared function introduces tight coupling between those sections of code. They both now rely on a shared interface; if you change that interface, both sections of code must be adjusted. In many situations this is perfectly appropriate; however, it's not always a desirable outcome, and can cause more problems in the long run than the duplication – so DRY your code responsibly!

Not all code duplication is malicious or the fault of lazy programmers. Duplication can happen by accident too, by someone reinventing a wheel that they didn't know existed. Or it can happen by constructing a new function when a perfectly acceptable third-party library already exists. This is bad because the existent library is far more likely to be correct and debugged already. Using common libraries saves you effort, and shields you from a world of potential faults.

There are also microcode-level duplication patterns. For example:

```
if (foo) do_something();
if (foo) do_something_else()
if (foo) do_more();
```

could all be neatly wrapped in a single `if` statement. Multiple loops can usually be reduced to a single loop. For example, the code in Listing 2 probably boils down to:

```
for (int a = 0; a < MAX; ++a)
{
  // do something
  // do something else
}
// make hot buttered toast
```

Listing 2

```
for (int a = 0; a < MAX; ++a)
{
  // do something
}
// make hot buttered toast
for (int a = 0; a < MAX; ++a)
{
  // do something else
}
```

if the making of hot buttered toast doesn't depend on either loop. Not only is this simpler to read and understand, it's likely to perform better, too, because only one loop needs to be run. Also consider redundant duplicated conditionals:

```
if (foo)
{
  if (foo && some_other_reason)
  {
    // the 2nd check for foo was redundant
  }
}
```

You probably wouldn't write that on purpose, but after a bit of maintenance work a lot of code ends up with sloppy structure like that.

> If you spot duplication, remove it.

I was recently trying to debug a device driver that was structured with two main processing loops. Upon inspection, these loops were almost entirely identical, with some minor differences for the type of data they were processing. This fact was not immediately obvious because each loop was 300 lines (of very dense C code) long! It was tortuous and hard to follow. Each loop had seen a different set of bugfixes, and consequently the code was flaky and unpredictable. A little effort to factor the two loops into a single version halved the problem space immediately; I could then concentrate on one place to find and fix faults.

Dead code

If you don't maintain it, your code can rot. And it can also die. *Dead code* is code that is never run, that can never be reached. That has no life. Tell your code to get a life, or get lost.

Listings 3 and 4 both contain dead code sections that aren't immediately obvious if you quickly glance over them.

Other manifestations of dead code include:

- Functions that are never called
- Variables that are written but never read
- Parameters passed to an internal method that are never used
- Enums, structs, classes, or interfaces that are never used

Comments

Sadly, the world is riddled with awful code comments. You can't turn around in an editor without tripping over a few of them. It doesn't help that many corporate coding standards are a pile of rot, mandating the inclusion of millions of brain-dead comments.

Listing 3

```
if (size == 0)
{
  // ... 20 lines of malarkey ...
  for (int n = 0; n < size; ++n)
  {
    // this code will never run
  }
  // ... 20 more lines of shenanigans ...
}
```

Listing 4

```
void loop(char *str)
{
  size_t length = strlen(str);
  if (length == 0) return;
  for (size_t n = 0; n < length; n++)
  {
    if (str[n] == '\0')
    {
      // this code will never run
    }
  }
  if (length) return;
  // neither will this code
}
```

Good code does *not* need reams of comments to prop it up, or to explain how it works. Careful choice of variable, function, and class names, and good structure should make your code entirely clear. Duplicating all of that information in a set of comments is unnecessary redundancy. And like any other form of duplication, it is also dangerous; it's far too easy to change one without changing the other.

Stupid, redundant comments range from the classic example of byte wastage:

```
++i; // increment i
```

to more subtle examples, where an algorithm is described just above it in the code:

```
// loop over all items, and add them up
int total = 0;
for (int n = 0; n < MAX; n++)
{
  total += items[n];
}
```

Very few algorithms when expressed in code are complex enough to justify that level of exposition. (But some ***are*** – learn the difference!) If an algorithm does need commentary, it may be better supplied by factoring the logic into a new, well-named function.

> Make sure that every comment adds value to the code. The code itself says *what* and *how*. A comment should explain ***why*** – but only if it's not already clear.

It's also common to enter a crufty codebase and see 'old' code that has been surgically removed by commenting it out. Don't do this; it's the sign of someone who wasn't brave enough to perform the surgical extraction completely, or who didn't really understand what they were doing and thought that they might have to graft the code back in later. Remove code completely. You can always get it back afterwards from your source control system.

> Do not remove code by commenting it out. It confuses the reader and gets in the way.

Don't write comments describing what the code *used* to do; it doesn't matter anymore. Don't put comments at the end of code blocks or scopes; the code structure makes that clear. And don't write gratuitous ASCII art.

Verbosity

A lot of code is needlessly chatty. At the simplest end of the verbosity spectrum (which ranges from infra-redundant to ultra-voluble) is code like this:

```
bool is_valid(const char *str)
{
  if (str)
    return strcmp(str, "VALID") == 0;
  else
    return false;
}
```

It is quite wordy, and so it's relatively hard to see what the intent is. It can easily be rewritten:

```
bool is_valid(const char *str)
{
  return str && strcmp(str, "VALID") == 0;
}
```

Don't be afraid of the ternary operator if your language provides one; it really helps to reduce code clutter. Replace this kind of monstrosity:

```
public String getPath(URL url) {
  if (url == null) {
    return null;
  }
  else {
    return url.getPath();
  }
}
```

with:

```
public String getPath(URL url) {
  return url == null ? null : url.getPath();
}
```

C-style declarations (where all variables are declared at the top of a block, and used much, much later on) are now officially passé (unless you're still forced to use officially defunct compiler technology). The world has moved on, and so should your code. Avoid writing this:

```
int a;
// ... 20 lines of C code ...
a = foo();
// what type was an "a" again?
```

Move variable declarations and definitions together, to reduce the effort required to understand the code, and reduce potential errors from uninitialised variables. In fact, sometimes these variables are pointless anyway. For example:

```
bool a;
int b;
a = fn1();
b = fn2();
if (a)
  foo(10, b);
else
  foo(5, b);
```

can easily become the less verbose (and, arguably clearer):

```
foo(fn1() ? 10 : 5, fn2());
```

Bad design

Of course, unnecessary code is not just the product of low-level code mistakes or bad maintenance. It can be caused by higher-level design flaws.

Bad design may introduce many unnecessary communication paths between components – lots of extra data marshalling code for no apparent reason. The further data flows, the more likely it is to get corrupted en route.

Over time, code components become redundant, or can mutate from their original use to something quite different, leaving large sections of unused code. When this happens, don't be afraid to clear away all of the deadwood. Replace the old component with a simpler one that does all that is required.

Your design should consider whether off-the-shelf libraries already exist that solve your programming problems. Using these libraries will remove the need to write a whole load of unnecessary code. As a bonus, popular libraries will likely be robust, extensible, and well used.

Whitespace

Don't panic! I'm not going to attack whitespace (that is, spaces, tabs, and newlines). Whitespace is a good thing – do not be afraid to use it. Like a well-placed pause when reciting a poem, sensible use of whitespace helps to frame our code.

ACCU18 Trip Report

Balog Pal reports his experiences from the 2018 ACCU Conference.

I had to skip last year, so was really eager to be there again and meet all the fine folks. This time I brought my young padawan and we decided to use a shared room (great deal as it's practically half price) in the hotel instead of finding a bed outside.

We arrived on Tuesday evening and I hoped to find an extra-schedule presentation as in previous years. And there was a local group meeting too really, just it ended by the time of our arrival. Too bad. So fast check-in and off to the bar to see who is already there. And indeed the area around the counter was filled with our people. I waved hi to John Lakos before asking for some ale, then started to process internal confusion alerts. some parts of the picture didn't match with my memory from 2 years ago. I even had to ask someone if that is John or not. And yeah, he was too, JL 2.0 (or maybe ½.0 would fit better). What is a great thing, last time I was actively worried for him.

Eventually I met most everyone already on site, but most people were pretty tired and time-lagged so it was off to sleep even before 2:00. There will be enough opportunity to catch up later.

I read the schedule around registration time and it, as usually showed an average over 3 'interested' flags per slot. Though the keynotes did not look very impressive, based on title and speaker at least, so I was prepared to just one good of the four, leaving enough room for a pleasant surprise. And after the opening we headed into the first keynote, that I found without any substance. On the bright side it was not against expectations.

After that I headed for Anthony's talk. (For sake of brevity I will not list my also-considereds as I could paste most of the schedule…) We learned about more 'advanced' tools in multithreading that are supposed to help in extreme scalability situations. Along with demonstrating problems that we encounter with the more basic tools. Especially without knowing details on how the hardware actually works. We also learned that hazard pointers and RCU proceeds well in the standardization.

After break I went for some nostalgia trip with Jez and Chris. In retrospect would probably better off with a different choice. Not even sure why, these days even nostalgia is not as it used to be? Then the last slot went to

BALOG PAL

Pal is an old fart who has been dealing with computers since the ZX81. His main focus is software quality and bug prevention. In the last decade he has been extending more to cover process and peopleware issues too. He can be contacted at pasa@lib.hu

Write Less Code! (continued)

Use of whitespace is not usually misleading or unnecessary. But you can have too much of a good thing, and 20 newlines between functions probably is too much.

Consider, too, the use of parentheses to group logic constructs. Sometimes brackets help to clarify the logic even when they are not necessary to defeat operator precedence. Sometimes they are unnecessary and get in the way.

So what do we do?

To be fair, often such a buildup of code cruft isn't intentional. Few people set out to write deliberately laborious, duplicated, pointless code. (But there are some lazy programmers who continually take the low road rather than invest extra time to write great code.) Most frequently, we end up with these code problems as the legacy of code that has been maintained, extended, worked with, and debugged by many people over a large period of time.

So what do we do about it? We must take responsibility. Don't write unnecessary code, and when you work on 'legacy' code, watch out for the warning signs. It's time to get militant. Reclaim our whitespace. Reduce the clutter. Spring clean. Redress the balance.

Pigs live in their own filth. Programmers needn't. Clean up after yourself. As you work on a piece of code, remove all of the unnecessary code that you encounter.

This is an example of how to follow Robert Martin's advice and honour 'the Boy Scout Rule' in the coding world: Always leave the campground cleaner than you found it. [1]

> Every day, leave your code a little better than it was. Remove redundancy and duplication as you find it.

But take heed of this simple rule: make 'tidying up' changes separately from other functional changes. This will ensure that it's clear in your source control system what's happened. Gratuitous structural changes mixed in with functional modifications are hard to follow. And if there is a bug then it's harder to work out whether it was due to your new functionality, or because of the structural improvement.

Conclusion

Software functionality does not correlate with the number of lines of code, or to the number of components in a system. More lines of code do not necessarily mean more software.

So if you don't need it, don't write it. Write less code, and find something more fun to do instead. ■

Questions

- Do you naturally write succinct logical expressions? Are your succinct expressions so terse as to be incomprehensible?
- Does the C-language-family's *ternary operator* (e.g., `condition ? true_value : false_value`) make expressions more or less readable? Why?
- We should avoid *cut-and-paste* coding. How different does a section of code have to be before it is justifiable to not factor into a common function?
- How can you spot and remove dead code?
- Some coding standards mandate that every function is documented with specially formatted code comments. Is this useful? Or is it an unnecessary burden, introducing a load of worthless extra comments?

Reference

[1] Robert C. Martin (2008) *Clean Code: A Handbook of Agile Software Craftsmanship*, Upper Saddle River, NJ: Prentice Hall.

Nico, a choice that just can't go wrong. And he showed a lot of new template-related material since C++11, traps with `auto` and `decltype(auto)`, `constexpr` and much more. Followed by 12 great lightning talks filled with humor and energy.

As an extra event after that I crashed the SG14 teleconfed meeting where Herb Sutter presented his new ideas about lightweight exception handling. One that would allow code generation similar to using return codes without messing up the source code. Both the material and the meeting itself was really interesting. The idea would open up exception handling on gaming and other high-performance platforms where it is now usually on the forbidden list. Everyone in the room and on remote agreed it is a thing worth pursuing. Beyond that, on details I observed that the votes went almost exactly crossed: the number of likes in the room matched the dislikes on remote and vice versa. Go figure. But it's okay, the pesky details can be figured out later to a better consensus.

It felt a long and exhausting day at that point but it seems the bar has really good regenerative environment, especially as almost everyone showed up. And the party went on to almost the sunrise as is supposed to happen.

Herb Sutter presented his new ideas about lightweight exception handling ... that would allow code generation similar to using return codes without messing up the source code

Thursday started with keynote on Kotlin. A thing I knew like nothing about (except a few mentions as successor of java). And by the presentation I liked it a LOT really, both the language and its dev environment. The only down point being it is promised to be ready 'by winter' without a firm specification of a continent and a year OTOH different sources strongly suggest that Winter is actually coming, so I have my hopes up and will try this thing in the near future.

Next I decided (looking at the program not sure how) to see how not to lead a team of software professionals. And it turned out as a really lucky choice. While it went at half or even third pace compared to others, the content was really touching, and felt a really honest recollection of one's mistakes and faults with a lot to learn from. On top of that I finally got the inspiration on how to make my next talk on ACCU that eluded me for good four years.

After the break I was really tempted to see John's where the presentation was slimmed similar to the presenter to mere 260 slides – but I decided to go live with Dietmar for a double handful of good reasons. And learned some really interesting tech. That it's even possible to get right on with simulated concepts before the real ones get implemented. It finished a little early so I could catch the tail of Hubert Matthews on reads and writes considered harmful. Turned out I completely misjudged the scope by the title and probably should have looked at this one from the start.

The last slot is a no-brainer choice with the pub quiz. (And who thinks it's fair to run Kevlin in this slot too.) No matter if it is with Jon at the helm while Olve can't make it to continue the series of yey-s. The format is a little different, instead of guessing what the code will produce we have to make up the shortest program that compiles and uses all the provided tokens. What could be better than a quiz with freebie drinks (kudos to Bloomberg folks) than to win it too. And it is not hard using my top talent of picking the proper team. That is not a hard guess either: the one with Richard Smith on. :) It turned out an interesting format in practice, though somewhat spoiled by abuse of templates where gcc maximizes opportunity in the 'ill-formed, no diagnostic required' realm, just make sure to have dependent expressions. Where Richard comes extremely handy. :) for the record the second team came up really close too.

After we had the lightnings squeezed in the small room as the big place was prepared for the dinner party. This year it was moved up from Friday which was a great thing. Another round of fun, and while the previous day Jon Kalb and Phil Nash lobbied for East const (the more agile people even could grab a bracelet beyond learning this cool name for the old phenomenon), now the conservatives pointed out how uncool it is to call the old-way 'West const'. When it is correctly 'const West'. Quite obviously – after the fact.

While the dinner commenced I went pubbing with Ralph, Anna and some more folks, and afterwards resumed the session in the bar till a good morning.

Friday started with Lisa's keynote about 'shape of the program'. Which turned out really enchanting for me. Possibly a big part was for her voice: it sounds extremely like Laurie Anderson and even matches the intonation. (I asked later, not deliberately, it just happens by chance). I only missed the music in the background, but I almost heard it internally. And the topology attached to the code was interesting too, providing a different section compared to what we encounter in our editors. Two out of three suspect keynotes with thumbs up, great outcome. After the note I was trying to query people on the mentioned Laurie Anderson similarity with not much luck: those I tried were not familiar. (Maybe we would need some talks on evolution of music?).

But that led to another lucky accident, my last victim was Michel Grootjans and I noticed his extremely artistic sketches of the keynote (really, look up his Facebook page!), and I missed the pick of the next session as it started in the room I appeared to be at the bell. So I learned about graphs from Dom Davis and an interesting new query facility based on those. Really insightful, I'm glad for this switch as I did not originally consider this to attend.

Also this day brought serious reinforcements: Francis Glassborow, Jonathan Wakeley and Mike Wong.

In the lunch break there was an ACCU meeting up in the lounge (this year it was dead space, no arcades, no booze and no people, hope it will return in the future). It was the preparation for the next-day's AGM, and a successful one too as we found a willing candidate for secretary quite out of the blue.

Then I attended Jon Kalb's summary of the past, present ad future of C++, that goes parallel to Uncle Bob's talk on 'future of programming' (obviously 95% being about the past 70 years…:). followed by the event I had been waiting for for so long: a grilling of the committee. Which was pretty tuned back on actual grilling and more filled with all kinds of fun.

Another round of great lightning talks was followed by the Microbrewery event (more kudos to Bloomberg) with food, booze and super-short chess tournaments. No kidding, 2 and a half minutes on clock that is barely enough to just move a piece let alone think what to do. Alan won the event and deserves all the praise despite losing the extra play-off against the local champion. We also had demo of music played by running programs on a rPi based thing. That unfortunately turned boring after the first five minutes. Most people tried to talk with each other but it was really hard with the loud noise so I rather left for some night walk in Bristol and back to the bar as usual. And the event was fine and alive, unlike at similar time in the previous years with the ACCU dinner on Friday when it felt like the end of the world.

On the last day I started with another obvious choice: Marshall and Jon paired up like Penn and Teller and introduced us to the trials and tribulations of those who try to implement the standard library. (And with handicaps too: turned out they proposed plenty of things to make their life easier, and those were even adopted into the standard or are on the way – but to keep the codebase compatible with many different compilers most of those improvements are on the not-to-use list. DOH. And we learned some interesting tricks too.

After that came the only slot I was not really moved by either talk or by title. Decided to go with the Total War, but it was mostly a letdown. In the break we had the AGM, pretty effective, due to the preparation on the previous day.

Don't Assume Any Non-Zero exit() Will Fail!

Silas S. Brown shares his finding on process exit codes.

I recently came across a little 'gotcha' on a BSD system which also affects GNU/Linux systems. I was using **make** to run a series of commands including a Python script, and I assumed that **make** would stop if the Python script exitted with non-zero status. In the Python script I had done **os.system()** to run a shell command, and, if the result is non-zero, then **sys.exit()** with that result to fail the Python script as well:

```
err = os.system("...")
if err: sys.exit(err) # DON'T DO THIS!
```

Except **make** carried on regardless. It didn't stop at the failure.

It turned out that **os.system()** was returning 256. And in the **bash** shell on both BSD and Linux, the command:

```
python -c 'import sys; sys.exit(256)' || echo fail
```

prints nothing.

The POSIX standard **wait()** and **waitpid()** calls provide only the lowest 8 bits of the exit status in **WEXITSTATUS**, so an exit status of 256 would be read as 0. The newer **waitid()** call returns 32 bits of exit status, but obviously not all system tools use it. (The underlying C standard defines only **EXIT_SUCCESS** and **EXIT_FAILURE** as 0 and 1; it says nothing about how platforms treat other values.)

But why was **os.system** returning 256 in the first place? As can be seen from:

```
>>> import os
>>> os.system("exit 3")
768
```

the return value of **os.system()** is not (on Unix/Linux) the straightforward exit code of the program it called. So we turn to the Python docs:

> On Unix, the return value [of **os.system()**] is the exit status of the process encoded in the format specified for **wait()**

which links to a page saying:

> exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

Incidentally the man page of the underlying C **system()** function on BSD also says, "The system() function returns the exit status of the shell as returned by **waitpid(2)**" which says you're supposed to call **WEXITSTATUS** to get the exit status from it if **WIFEXITED** is true.

So to be more portable, you have to do something like:

```
if (err & 0xFF) == 0: err >>= 8
sys.exit(err)
```

or check what platform you're on and act accordingly.

On Windows, **os.system()** straightforwardly returns the exit code of the command interpreter. But not all Windows command shells actually pass on the exit code of the program they ran. `cmd.exe` does, but the older `command.com` on Win9x systems didn't, and more worryingly third-party 'power shells' and such are not guaranteed to (the Python docs say check their documentation). So the only way you can guarantee to find out the real exit code is to run the program directly with **os.spawn***, or use the (new in Python 2.4) **subprocess** module, instead of using a system shell. ■

SILAS S. BROWN

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

ACCU18 Trip Report (continued)

For the last slot another (for me) obvious choice going with Mike Wong. He was fighting off some cold, but even with that radiates enough stamina to put a small space station into orbit. He told us about the progress in the many study groups he is involved in. And the framework created to finally harness the power in GPUs we have around that are quite underused. I really look forward to having this SYCL stuff put to use.

And the conference is heading into conclusion with Seb Rose's keynote that I originally picked as the only one promising. And it was a great one too. Seb was out for hiking in the beautiful woods and mountains of South-France. And found some rally fitting parallel with our everyday practice of software development. From estimating to clashing our well-conceived plans with reality. Then embracing all the change the latter forces on us. This is really a talk everyone should see, packing much beauty, laughs along with wisdom.

At the end Seb gave the audience some exercise: grab a piece of paper and write down something one learned here at the conference and plans to put it into practical use as they return to work. And exchange the paper with someone around who will query for the progress a few weeks ahead. (Yeah, that wisdom from agile schools about making things actually happen by declaring them in public. Unlike all the promises we just make to ourselves and way too easy to drop out…)

I did not fill the paper as though I did learn plenty of interesting things, none of them fit as immediate change in my work. But I did think up some promises: one, to write and submit a trip report. That, if you read this, actually happened. Second, I want to hang on to the inspiration and prepare a talk for the next ACCU conf. (For what I also promised myself earlier, that I shall have the prez complete at submission time, not just a title, summary and hope to make it up in the last 3–4 months time…) And as a third thing I had an idea for a C++ proposal to be submitted to WG21 with some gathered feedback (Richard Smith said 'it looks useful' and 'I see no immediate problems' so it should have a chance). Let's hope this public announcement can create enough starting momentum for them to not sink under boring problems of everyday struggles.

In the closing words Russel asked for some praise for Francis Glassborow who created this whole thing now called ACCU and he got a loud Hurrah. And also hinted that he has a preliminary promise for the venue same time next year, a good seed for ACCU2019.

As a summary, it was a great conference and I look forward to the next one already. In the meantime I started to watch the talks I missed – we have some great progress here, I recall in 2013 there was just one camera, then two and now almost all talks are recorded. Also it took 2–3 months for them to appear on YT, now the first ones came up just few days later and the rest is coming at steady pace. The one thing I miss are the photo galleries. If you have one, please send a link to our website so it is shared. Also waiting to read other people's trip reports. If you were there, write up your experience! And see ya all on the next instances.■

Everyday Coding Habits for Safety and Simplicity

Arun Saha has some simple advice for forming good habits in C++.

A habit is a choice that we deliberately make at some point, and then stop thinking about, but continue doing, often every day [1].

A lot of times, minor differences in code can lead to significantly better code. In this article, let us go together through a few such cases which we encounter in our everyday coding life. None of these is revolutionary (in fact, you may have seen them before); rather this is a collection of suggestions how small changes – a few characters in most cases – can make big differences. [The examples in this article are in C++; a lot of them apply in C as well.]

Always initialize automatic variables

Do not define uninitialized variables. For example, don't do this:

```
int result;
```

This defines the variable but it is not initialized to any known value. (It will have an indeterminate value.) I have seen a lot of bugs arise for this reason alone. Often such bugs stay hidden for a long time and express themselves one fine morning. For example, this may happen when the compiler or the compiler version is changed.

Instead, initialize it to some known value:

```
int result = 0;
```

The exact value to be initialized is not important; choose something that is appropriate to the context.

Note that this is different from accessing (reading) uninitialized variables (which can be detected by the compiler warning option **`-Wuninitialized`**). Here, the recommendation is to never leave a defined variable uninitialized, even if it is not being accessed.

As you might have realized, this applies only to *automatic* (or local) variables of the primitive types like **`char`**, **`int`**, **`double`** with all possible signedness and width, **`float`**, and **`bool`**. If **`T`** is a type with sane default constructor (such as **`std::string`**), then

```
T result;
```

is not a problem.

One of the reasons for this behavior is that the primitive types do not have constructors. Despite that, a compiler may choose to emit an instruction for 'zero' initialization. However, with C++ and C compilers – together with their users – being very performance sensitive, such (initialization) instructions are not usually emitted since they are not required by the standard.

[Would it make sense for compilers to provide a knob such as **`-fzero-initialize-primitives`**?]

Prefer defining variables only when needed

Don't define a variable too early only to assign it later. For example, don't do this:

```
int result = 0;
// many
// intermediate
// lines
result = ComputeSomething(input);
```

Rather, define and initialize it together at the point it is used.

```
int result = ComputeSomething(input);
```

If the variable is not expected to be modified thereafter, then mark it constant as:

```
int const result = ComputeSomething(input);
```

In fact, try organizing the flow of logic and code such that it is a series of **`const`** variables. It not only prevents inadvertent modification of the variable, it also makes it easy for a reader to follow the flow; it will be one less thing to track in her working memory. It is considered that the number of objects that a human can hold in working memory is 7 ± 2 [2]. I try to be safe and be on the lower side!

Prefer input arguments to be immutable

The safety argument applies to function arguments too. Prefer making the function arguments **`const`** as much as possible. If the method is just reading the data without modifying, then the data must be **`const`**. So, instead of

```
void PrintInorderTraversal(TreeNode * root);
```

prefer

```
// Declaration in .h
void PrintInorderTraversal(
  TreeNode const * root);
```

I recommend going one step further by marking the input variable **`const`**. This can be done in the implementation file without putting it in the header file. This **`const`** prevents the local variable pointing to the data from accidentally getting modified and starting to point to different data.

```
// Definition in .cc
void PrintInorderTraversal(
  TreeNode const * const root);
```

The recommendation applies equally well for pass-by-reference arguments. There, instead of

```
bool IsValidIpAddress(std::string& input);
```

prefer

```
bool IsValidIpAddress(std::string const& input);
```

Accessor member functions must be const

In C++, member functions can be **`const`** when they are not modifying the object. Prefer making the member functions **`const`** whenever possible. For example, if a method, such as **`Print()`**, is not modifying the object, then instead of

```
struct Foo {
  <stuff>
  void Print();
  <more stuff>
};
```

ARUN SAHA

Arun works in software-defined data centers. He is passionate about building robust software infrastructure, engineering high-quality software, and improving productivity. Arun holds a B.S. and Ph.D. in Computer Science and can be reached at arunksaha@gmail.com.

prefer writing it as

```
struct Foo {
  <stuff>
  void Print() const;  // Note: 'const'
  <more stuff>
};
```

As you know, if a member variable is marked **mutable**, then even a **const** qualified method can modify it. A mutable variable is appropriate only in some specific circumstances (e.g. a mutex, a hit counter); don't overuse it.

Prefer default member initialization

Member variables of primitive types are initialized in the constructor. Usually, the constructor is in a different file and there is potential for the initialization to be missed. This tends to happen less during initial creation but more during later enhancements and maintenance. To prevent that, do not defer the initialization until the constructor; instead, prefer to initialize them inline in the class body.

Consider the following snippet.

```
// .h file
struct Foo {
  explicit Foo(int64_t);
  <stuff>
  int64_t fooId_; // Initialization deferred to
                  // ctor Potential to be missed
}

// .cc file
Foo::Foo(int64_t fooId) : fooId_{fooId} {}
```

For safety, I recommend writing it as the following using the default member initialization feature [3].

```
struct Foo {
  <stuff>
  int64_t fooId_{0}; // Known initialization if
                     // missed in .cc constructor
}
Foo::Foo(int64_t fooId) : fooId_{fooId} {}
```

Except for a few types like bit-fields and non-length-specified arrays, this can be used for most of the data types. Note that, if a member variable has a default member initializer and also appears in the member initialization list in a constructor, the default member initializer is ignored.

You may have noticed that the concept here is similar to an item above except that the variable type is different: automatic variable vs. member variable. In both cases, the motivation is to ensure that a variable is always initialized to a known value. Instances of such missing initializations may be detected by g++ with the warning option **-Weffc++** [4].

Prefer immutable member variables if you can

For class members which are initialized at construction and never change during the lifetime of the object, prefer marking them const. Consider the following example where the object carries the time instant when it was constructed.

```
using TimePoint = std::chrono::time_point
  <std::chrono::system_clock>;
inline TimePoint
GetTimePointNow() {
  return std::chrono::system_clock::now();
}
```

Instead of the following

```
// .h file
struct Foo {
  <stuff>
  int64_t   fooId_{0};
  TimePoint  creationInstant_;
}
// .cc file
Foo::Foo(int64_t fooId) :
  fooId_{fooId},
  creationInstant_{GetTimePointNow()} {}
```

prefer doing

```
// .h file
struct Foo {
  <stuff>
  int64_t const   fooId_{0};
  TimePoint const creationInstant_; // Note const
}
```

The motivation is similar to marking automatic variables **const**; safety from accidental modification and reduced burden of information tracking.

Prefer APIs to provide the result as a return value

Don't design an API to update a reference or a pointer to the result, let it *return* the result. For example, instead of doing

```
int result = 0;
ComputeResult(n, result);   // pass by reference
```

or

```
int result = 0;
ComputeResult(n, &result);  // pass by pointer
```

prefer doing

```
int const result = ComputeSomething(n);
```

This style allows the returned value to be saved in a **const** variable and to be used in the subsequent computation. This certainly applies to objects that are trivially copyable, such as the primitive types. For objects that are not trivially copyable, it can still be applicable thanks to:

1. Return Value Optimization (RVO)
2. Copy Elision
3. Move semantics

Note however that the rules of the above are quite involved and, in certain situations, they may not kick in. So, if you want to play safe and avoid it for non-trivially copyable types, that's understandable.

Prefer for-loops over while-loops

Instead of writing as a **while** loop, prefer writing loops as **for** loops whenever possible. This is not just a bias towards one keyword over another; this enables easier comprehension of loop invariants. For example, instead of the following

```
int i = 0;
while (i < n) {
  <possibly multiple lines of loop body>
  i++;
}
```

prefer writing it as

```
for (int i = 0; i < n: ++i) {
  <loop body>
}
```

In addition, this style has another advantage: the loop variable (here, **i**) is not 'leaked' to the outer scope *unnecessarily*.

The recommendation goes for linked list traversal too. Here, instead of

```
ListNode const * curr = head;
while (curr) {
  <possibly multiple lines of loop body>
  curr = curr->next;
}
```

prefer writing it as

```
for (ListNode const * curr = head; curr;
  curr = curr->next) {
  <loop body>
}
```

As if the above is not enough, the above applies to traversing from both ends. So, instead of

```
void Reverse(T * arr, int64_t const n) {
int64_t left = 0;
  int64_t right = n - 1;
  while (left < right) {
    swap(arr[left], arr[right]);
    left++;
    right--;
  }
}
```

prefer writing it as

```
void Reverse(T * arr, int64_t const n) {
  for (int64_t left = 0, right = n - 1;
  left < right; ++left, --right) {
    swap(arr[left], arr[right]);
  }
}
```

Note that although it is a by product, saving the number of lines is not the objective.

Prefer combining boolean conditions into a well-named variable

If the program logic requires evaluating multiple conditions, then instead of clamping them together in the conditional of an **if** statement, for example, instead of

```
if (someCondition(input) &&
  anotherCondition(input) &&
  !yetAnotherCondition(input)) {
    <body>
```

prefer

```
bool const input_valid = someCondition(input) &&
  anotherCondition(input) &&
  !yetAnotherCondition(input);
if (input_valid) {
  <body>
```

This has two benefits: the evaluated condition gets a human-readable name and it becomes easier to view what the condition evaluated to (i.e. true or false) using a log message or a debugger.

Prefer separate asserts for separate conditions

If you use asserts in your source code to validate assumptions, then do not combine multiple conditions into a single **assert** statement.

```
void Copy(char * dst, char const * src,
  size_t const length) {
    assert(dst && src); // 2 conditions combined
  <continued>
```

If you do so and the assertion fails, then the generated message would mention that the combined condition has failed (along with some meta information like filename and line number). The person who has to debug the problem (it could be you at a future time) would not be clear exactly why the assertion failed. Instead, break the single assertion into multiple simpler assertions.

```
void Copy(char * dst, char const * src,
  size_t const length) {
    assert(dst);
    assert(src);
    <continued>
```

Now, if the assertion fails, it would be obvious what exactly has failed!

Summary

The table at the bottom of the page provides an at-a-glance view of the above recommendations.

Software engineering is an eternal fight with complexity. The recommendations here are based on my experience as a software engineer. Over time, I have found them useful in constructing durable code. Unless your situation precludes, I recommend trying them out. Once they become a habit [1], you would cease thinking about them and focus on other important aspects of your code. ■

References

[1] http://charlesduhigg.com/how-habits-work/

[2] https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

[3] https://en.cppreference.com/w/cpp/language/data_members

[4] https://stackoverflow.com/questions/2099692/easy-way-find-uninitialized-member-variables

	Avoid doing	Prefer doing
1	`int result;`	`int result = 0;`
2	`int result = 0;` `int result = ComputeSomething(input);`	`int const result =` `ComputeSomething(input);`
3	`void PrintInorderTraversal(TreeNode * root);` `bool IsValidIpAddress(std::string& input);`	`void PrintInorderTraversal(TreeNode const * const root);` `bool IsValidIpAddress(std::string const& input);`
4	`void Foo::Print();`	`void Foo::Print() const;`
5	`int64_t fooId_;`	`int64_t fooId_{0};`
6	`TimePoint creationInstant_;`	`TimePoint creationInstant_;`
7	`ComputeResult(n, &result);`	`int const result = ComputeSomething(n);`

8	`int i = 0;`
	`while (i < n) {`
	`  <loop body>`
	`  i++;`
	`}`
		`for (int i = 0; i < n: ++i) {`
		`  <loop body>`
		`}`

9	`if (someCondition(input) &&`
	`  anotherCondition(input) &&`
	`  !yetAnotherCondition(input)) {`
		`bool const input_valid =`
		`  someCondition(input) && anotherCondition(input) &&`
		`  !yetAnotherCondition(input);`
		
		`if (input_valid) {`

10	`assert(dst && src);`	`assert(dst);` `assert(src);`

The Half-Domain/Half-Primitive Proxy

Chris Oldwood presents a pattern for abstracting client-side proxies for testing.

I recently spent a few years working in the retail sector building web APIs. Some of these projects were greenfield but I also worked on a number of existing (brownfield) services too. As you might expect working on variations of a theme you start to see patterns developing in the way the services are designed and implemented.

There are apparently only so many ways you can skin a cat and once you have chosen your programming language and web service framework many things naturally fall into place as you have bought into their paradigms. Plenty has been written about these frameworks and how to use them but that's not the pattern of focus here, this article looks at the other end of the wire – the client proxy – an area with less attention to detail.

Service proxy use cases

When building a web API the ultimate deliverable is the service itself, more commonly these days a REST API using JSON as the wire-level content format. If you're building an API in the enterprise you may have to support XML too but that seems to be heavily declining as JSON has been the lingua franca of Internet APIs for some time. The service might be deployed on premise or in the cloud and could be self-hosted, i.e. you own the hosting process, or run as part of a shared web service such as IIS. Hence any client that wants to talk to your service needs to be fluent in HTTP and JSON too. There are plenty of libraries available for handling these low-level details; the client use cases we are interested in though sit just a little higher up the call stack.

Acceptance testing

The most likely client in the beginning will be the set of acceptance tests used to describe and verify the service's various behaviours. In something reasonably small and well defined like a web service the balance of the testing pyramid [1] may be skewed in favour of more customer tests and less programmer tests as they provide the perfect opportunity for outside-in [2] development. Yes, unit tests are still highly useful for those behaviours which are much harder to invoke from the outside without exposing back-door endpoints, such as error recovery, but the lion's share can be invoked by an external client.

Therefore our primary need for a proxy is to facilitate the writing of acceptance tests for our service. Our test cases need to be able to invoke our API in a variety of ways to ensure we cover both the happy paths but also the less happy ones too by ensuring both malicious and accidentally malformed inputs are correctly handled and the caller is provided with as much information as possible to correct the error of their ways. For example we might write a test like the on in Listing 1 to check that a REST resource is correctly secured.

Listing 1

```
[TestFixture]
public class orders_are_secured
{
  [Test]
  public void
request_returns_unauthorised_when_token_missing()
  {
    const string customerId = "123456789";
    var request =
      new FetchOrdersRequest(customerId);
    request.RemoveHeader("Authorization");

    var proxy = new ShopProxy();
    var response = proxy.Send(request);

    Assert.That(response.StatusCode,
      Is.EqualTo(401));
  }
}
```

Deployment/monitoring/smoke testing

While acceptance tests take care of all the fine details about how the service works, you probably want something a little simpler to just 'kick the tyres' during (or just after) a deployment. You could re-purpose the acceptance tests, minus any that use private APIs to perturb the clock or other internal details, but they are often overkill when all you really want to do is exercise some core customer journeys. For example one journey might be to find the most recent order for a known test customer (as we've just exercised the journey to place an order for them and already know it succeeded): see Listing 2.

Listing 2

```
public static void ViewLatestOrder(
  CustomerId customerId, ...)
{
  var shop = new ShopProxy();
  var token =
    shop.AcquireCustomerToken(customerId,
    credentials);
  var orders = shop.FetchLatestOrders(customerId,
    token);
  . . .
}
```

Deployment isn't the only time when you might want to exercise your API in this way though; you'll want to keep doing it all through the day as part of your monitoring strategy. The use of 'canaries' is a common technique for firing requests into your system at repeated intervals to allow you to record how it's behaving. If you can capture some performance data at the same time [3] you have an easy way to track the change in performance over time from the client's perspective (if you can invoke it from another data centre) and correlate that with the load at those moments in time.

Even during development or bug fixing it's still useful to be able to run the smoke tests quickly over your local build, especially if the acceptance tests take some shortcuts, such as hosting the API in-process. For example, whenever changing something other than functional code that will affect the set of deployed artefacts, like adding or removing assemblies or updating 3rd party package, it's useful to do a full build and deploy locally and then run the smoke tests before committing. This is to

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

ensure nothing has been left out, like a configuration setting or binary that will cause the service to choke on start-up and unnecessarily disrupt the delivery pipeline.

Support

In my recent 'Toolbox' column Libraries, Console Apps and GUIs [4] I described how I've found that home-grown tools often have a habit of being useful to a wider audience, such as a support team. Hence when you have a simple tool that can exercise a few basic journeys you already have all the code you need to provide fine-grained access to your service through, say, a command line interface.

For something like a REST API which has been built on open protocols and for which the Internet provides a bewildering array of tools it might seem a little wasteful putting a custom tool together. In the early life of an HTTP service a tool like CURL or a Chrome extension such as Postman can provide everything you need to poke the endpoints and see the responses. However once you start needing to make calls out to more than once service to accomplish a bigger task, such as requesting an access token from an authentication service, the friction starts to accumulate.

At this point that shell script which fuses together CURL for the HTTP aspects, along with, say, JQ for the JSON parsing starts to become a little unwieldy, especially if you're on the Windows platform where these tools aren't installed by default. A Windows heavy enterprise will probably use an NTLM based proxy too for accessing the outside world which can create another world of pain if you're trying to use open source tooling [5]. Hence putting together a little .Net based tool (or PowerShell script which uses .Net under the covers) can hide this unnecessary noise from the (support) user. Fortunately CURL is one of those tools which can punch its way through an NTLM proxy (with the relevant command line switches), although you still need to watch out for certificate problems if the organisation uses its own. (Self-signed certificates are a common technique in the enterprise at the moment for monitoring traffic to the outside world.)

With the various API endpoints exposed through a little command line tool it makes it really easy to automate simple support and administration tasks. Instead of stitching together low-level tools like CURL and JQ you can work at a higher level of abstraction which makes the already daunting tasks of comprehending a shell script just that little bit easier. For example I might need to check the loyalty points balance for an arbitrary customer that would first require a special, short-lived 'administration token' to be obtained using my separate administrative credentials:

```
> PointsAdmin view-balance --customer 123456
--administrator %MY_ADMIN_LOGIN% --password
%MY_ADMIN_PASSWORD%
42.5
```

This approach can also be used as an alternative to monkeying around with the database. While there may be times when a one-off database tweak might be necessary it's preferable to factor support and administration functions directly into the API (secured of course) so that the chances of corrupting any data are minimised by accidentally violating some invariant which is enforced in the service code rather than the database (a necessity with the newer document-oriented databases).

Client SDK

One natural by-product of creating an independent proxy for your service is that you have something which other service consumers might find useful, assuming of course they are using the same language or runtime. In the enterprise arena where the desire is to minimise the number of unique languages and runtimes in play it's highly likely that one more consumers may find your wares of interest. (With the CLR and JVM the programming language does not have to be the same but in my experience enterprises still limit the number of languages even on these runtimes despite the built-in interoperability benefits.)

That said it's easy for shared libraries like this to become a burden on the team, especially when the consumers assume they form a supported part of the product. If that is a responsibility the team wishes to take on, then so be it, but unless this is agreed upon the proxy should be considered nothing more than a 'leg up' to help a consumer bootstrap their use of the service. They should have the choice of whether to fork your source and customise it, or build their own. Removing this supporting role often goes against the commonly established enterprise goal of maximising efficiency, but it is necessary to ensure the higher purpose of decoupling the service and consumer deliveries where possible.

On the face of it providing a proxy may not seem like much of a burden but the interfaces and data types are only one part of what it takes to write a solid, reliable client. Once you start factoring in logging [6], instrumentation, and monitoring [7] the need to provide hooks to allow your code to interoperate with whatever product choices the caller has chosen for these concerns adds to the complexity. If you plan to use your own technical and domain types [8] too, e.g. Optional, Date, CustomerId, etc. then these need to be easily accessible so the consumer doesn't need to ship half your service libraries as well as their own just to use it.

Typical evolution – multiple choices

In my experience these various use cases, when they do eventually appear, will end up as a discrete set of different libraries or tools. Consequently by the time the pattern begins to surface the design has long passed the point where a simple refactoring can bring together the various strands into a single coherent component. What follows below is one common path I've observed.

Acceptance tests

The proxy used in the acceptance tests will no doubt evolve in a very rough-and-ready fashion. Being 'merely' test code there will be little thought put into how the tests need to talk to the service under inspection and so the code will probably suffer from a form of Primitive Obsession [9] where the native HTTP client library will feature heavily. Eventually common utility methods will be introduced but no serious refactoring will take place and the need to verify both happy and error paths will lead to a smorgasbord of overloads with some throwing and others returning values. Being used solely for testing (at this point) there's a good chance that the test framework assertion methods will be interwoven with the proxy code (rather than there being a clear boundary between 'sending the request' and 'validating the response'); for example, see Listing 3.

In the early days of an API where you're not even sure what conventions it will adopt, such as what URL structure to use, query parameters vs body, headers, etc. the API needs to evolve quickly. If the initial focus is on the happy path then the size of the proxy interface may be quite small and easily maintainable but as the error paths begin to play a role and

Listing 3

```
public string OpenTestAccount(string customer)
{
  var request =
   new OpenAccountRequest{ Customer = customer };
  var response =
   ShopProxy.Send<OpenAccountResponse>(request);
  Assert.That(response.StatusCode,
    Is.EqualTo(200));
  Assert.That(response.AccountId, Is.Not.Empty);
  return response.AccountId;
}
[Test]
public void closing_an_account_returns_ok()
{
  var accountId = OpenTestAccount(". . .");
  var request = new CloseAccountRequest{
    AccountId = accountId };
  var response = ShopProxy.Send(request);
  Assert.That(response.StatusCode,
    Is.EqualTo(200));
}
```

more control is needed around the URL, headers and body content, the more work is needed to keep everything in sync and make sure that every test really does exercise the behaviour it thinks it does. If the API changes quite radically, e.g. a change in the 'resource' structure, then a lot of ad hoc test code might need fixing up – treating test code as a first class citizen is important for supporting rapid change.

Deployment/monitoring/smoke tests

As alluded to earlier if the service is based on open protocols, like HTTP, then it's certainly possible to cobble together whatever you need from established open source tools, like CURL. Deployment tools like Ansible and dynamic languages like Python have plenty of modules for both high-level tasks, like configuring a machine, and lower-level tasks, such as testing port connectivity and making basic transport requests. Therefore it's quite possible that unless the development and deployment teams are aligned (if they're not one and the same) they will each use what they know best and create their own method for testing the service's availability. This isn't a problem per se unless the development team wants to iterate quickly over their API and it causes friction down the delivery pipeline due to unexpectedly breaking the deployment and monitoring processes.

This is one of the reasons why operations' concerns must be factored into any user stories and the definition of 'done' should include monitoring and alerting updates as well the more obvious developer oriented tasks.

Support

A similar argument can probably be made for any support requirements – they will use whatever off-the-shelf tools they can find or are already used to. Similarly if they have a choice between directly accessing the database and trying to go through the API, the path of least resistance leads to the former choice unless their needs are also explicitly factored into the service design and tooling.

Client SDK

The reason for choosing open protocols is that it affords any consumers the ability to choose what technologies they prefer for accessing the service. Therefore on the principle of 'use before reuse' [10] there is unlikely to be a direct need to provide a formal client proxy unless the team happens to be responsible for maintaining a number of related services and therefore there is something to be gained from having a component that can be shared.

What is highly unlikely to occur (unless it's a conscious decision) is that the proxy developed for the acceptance tests can be easily factored out into a separate component that is then reusable in both scenarios. It is more likely the other way around, i.e. reusing the client SDK for some aspects of the acceptance tests (happy paths), if the refactoring costs are considered worth it.

Pattern vocabulary

Although the pattern uses the term 'proxy' it should be pointed out that this use is not entirely in keeping with the classic Proxy pattern described in the seminal book on Design Patterns by 'The Gang of Four' [11]. One of the key traits of the pattern described in that book is that the proxy exposes the same interface as the underlying object. In essence a proxy in their eyes is entirely transparent and acts purely as a surrogate for the real deal. In this article the underlying transport class provided by the language runtime fulfils that role, which for .Net would be HttpClient for the HTTP protocol.

One of the most important aspects of the Design Patterns movement was the attempt to introduce a common vocabulary to make it easier to talk about similar design concepts. Hence it feels a little awkward not to stick to the letter of the law, so to speak, and use it for something which is more loosely related. For me proxies have been placeholders for remote objects and services ever since I wrote my first RPC code on a Sun 3/60 workstation at university and therefore I feel the surrogate aspect of this design is in keeping with their pattern, even if the exposure of a different interface is not. Hopefully no one will be confused by my choice of term because I've painted outside the lines in a few places.

That does not mean though that we are entirely excluded from using the other patterns in that seminal work for describing our 'creation'. On the contrary this pattern is essentially just the composition of two of their other classics – Adapter and Façade.

Pattern structure

Figure 1 shows the general shape of the pattern:

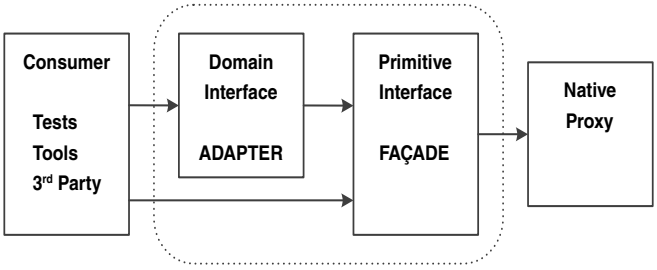


Figure 1

Back-end façade

Working slightly unusually from the network end back up to the caller we begin with the underlying transport class that provides the true proxy for the service we want to access which is often provided by the language runtime. Being a general purpose class this needs to expose everything that anyone would ever likely want to do to access a service using the type of network transport in question. For something like HTTP this could include everything ranging from simple JSON REST APIs up to streaming content of different media types using different encodings.

Our requirements however are often far more meagre and what we initially need is to reduce that all-encompassing interface down into something simpler and more focused. For example if we're talking to a microservice using HTTP and JSON we can simplify a lot of the interface and make many more assumptions about the responses. In design pattern terms this simplification is known as a façade.

Front-end adapter

With the transport layer interface largely simplified, albeit still with some elements of complexity to allow ourselves enough control over the more unusual requests we'll be sending, such as during testing, we now need to hide that grunge for the client that isn't interested in any of that but wants something richer and more intuitive.

Hence the front-facing half of the proxy is more akin to an Adapter from the original Gang of Four book. In this instance its role is to present an interface to the caller that deals in rich types, which is typically why we're using a statically typed language, and convert them into the more text oriented world of the façade. Likewise the return path transforms the responses from text to types and also maps any failures into a more suitable error mechanism.

There is an element of further simplification going on here too, so it is also façade-like in nature, but its primary purpose is to provide a different interface rather than a simpler version of the same one.

Pattern interfaces

Now that we have a grasp on the various forces that are driving the design and a feel for the overall pattern shape we can look at the interfaces of the two halves in more detail. This time we'll look at them in the more conventional order starting with the consumer's perspective and then moving behind the curtain.

The domain interface

For normal production use it is desirable to move away from a typical 'stringly typed' interface and instead traffic in rich domain types like dates, product identifiers and enumerations. At this level, if you're using

a statically typed language, then the type system can work for you, both to help catch silly mistakes but also provide you with hints through tools like IntelliSense.

To invoke a simple query on the service shouldn't require you to know how to format the URL, or what goes in the header, or how dates and times need to be passed in JSON (which has no native date or time type). The domain level interface abstracts all of this away and hides enough of the transport details to make remote calls appear less remote (avoiding overly chatty interfaces).

```
var shop = new ShopProxy(hostname);
var lastMonth = DateTime.Now.AddDays(-30);
var orders = shop.FindOrders(lastMonth,
  Status.Delivered);
```

The network is unreliable though and so error handling probably takes the form of exceptions as recovery from low-level transient request failures will likely take the form of continuous retries somewhere further up the stack. A basic exception hierarchy can support the most common catastrophic error scenarios, such as timeouts, disconnections, service failures (e.g. 500 and 503), client errors (e.g. 400), etc. Buried in these exceptions might be more fine-grained diagnostic context but the exception type itself should be enough to indicate if it's a (fast) retry scenario, there is something more fundamentally broken at the client end and a (slow) retry or triaging is required, or the request is just broken and we need to 'return to sender'.

While exceptions might be a suitable technique for handling transport level failures there are a number of HTTP error codes which fall into the less exceptional end of the spectrum, such as 404 which means a resource couldn't be found. While this might be down to a bug in your client (you've requested the wrong thing due to a URL formatting error) if there is a chance, either through concurrency or due to later deletion, that it might not exist then an `Optional<T>` would probably be a better return type (for example, see Listing 4).

Likewise if updating a resource can fail due to concurrency conflicts then it's better to represent that in the interface so the caller stands a better chance of realising they need to do more work than simply keep sending the same request again and again. (They may choose to turn it into a 'conflict' exception type but that's their judgement call.)

A key rationale for the split in the design is that the domain-level proxy sits on top of the primitive-level proxy and therefore has the opportunity to compensate for idiosyncrasies in the representation chosen by the service so the caller gets a more cohesive interface. A common example is where a web API chooses to represent domain errors by returning a 200 status code and a body instead of using the preferred 4XX status codes. In my recent article on monitoring [5] I gave another example of where a 400 status code was used instead of a 404 when a resource was missing. In both these cases the normal REST semantics have been 'abused' and therefore we need to transform an apparent success or failure into the other realm – it's the job of the domain-level proxy to hide these kinds of

Listing 4

```
public Optional<Balance> FetchBalance(AccountId
accountId)
{
  var request =
    new FetchBalanceRequest(accountId);
  var response =
    proxy.Send<BalanceResponse>(request);

  if (response.Succeeded)
    return Optional<Balance>.Some(
       response.Content.Balance);
  else if (response.StatusCode == 404)
    return Optional<Balance>.None;
  else
    throw new BalanceFetchError(. . .);
}
```

Listing 5

```
public Optional<AccountId>
LookupAccount(CardNumber cardNumber)
{
  var request =
    new LookupAccountRequest(cardNumber);
  var response =
    primitiveProxy.Send<LookupResponse>(request);

  if (response.Failed)
    ThrowError(response);

  if ( (response.StatusCode == 200)
    && (!response.Content.Error.IsEmpty()) )
  {
    throw new
      AccountLookupError(response.Content.Error);
  }
  return response.Content.AccountId.ToOptional();
}
```

edges. (If the service provider ever releases a new version of the interface that corrects these defects the domain interface should remain unaffected.) See Listing 5.

As we'll see in a moment though the consumer always has the possibility of dropping down a level if the richer interface doesn't provide the kind of semantics they're looking for.

The primitive interface

While the outer layer focuses heavily on the happy path and tries to make things easier for the consumer by leveraging the type system, the inner layer provides a more raw experience whilst still shielding the caller from all the gory details of the underlying transport mechanism. The aim is still to provide a coherent API whilst at the same time giving them more control over the various elements of the request and response when required.

Hence whereas the domain-level proxy will traffic in rich domain types the primitive-level proxy will likely deal in basic structures and string based values. The client will also have the option to control the headers and URL if required, whilst using any additional building blocks provided for those aspects they chose to adopt the default behaviour for. Another classic Gang of Four pattern, Builder, can be used to provide full control while helper methods can be added to get the ball rolling, see Listing 6 (for example).

When writing acceptance tests you usually want to create various malformed requests to test your validation code, these require as much of the request to be production-like as possible with only one aspect mutated in each scenario. Hence to avoid false positives the test should leverage as much of the real production code as possible to format the request whilst still being able to tweak the one aspect under test. For example we might write a test like the one in Listing 7 to check we only allow dates to be sent in ISO 8601 format.

The approach to parsing the response and error handling will also typically be more manual. Rather than throw exceptions the primitive-level proxy will largely return composite values that contain both the successful result and the error on failure. The caller then inspects the return value and queries the relevant part of the structure and acts accordingly. In the case of a successful request the body will contain the content, if any, whereas in the failure scenario any combination of the result code, headers and body could be used to formulate a richer error value. A complex result type for a JSON based REST API might look something like Listing 8.

Although we could have used an `Optional<T>` type for the success and error contents and foregone the Succeeded property I've chosen instead to make those an implementation detail here (not shown) and force the caller to query the Succeeded property to decide which content object should then be inspected – Content or Error.

Listing 6

```
public class FetchOrdersRequest
{
  public Headers Headers { get; }
  public Content Content { get; }

  public class Content
  {
    public string AccountId { get; set; }
    public string FromDate { get; set; }
    public string ToDate { get; set; }
  }
}

public static FetchOrdersRequest
  BuildFetchOrdersRequest(this
  ShopProxy proxy, AccountId accountId,
  Date from, Date to)
{
  return new FetchOrdersRequest
  {
    Headers = proxy.DefaultHeaders,
    Content = new Content
    {
      AccountId = accountId.ToString(),
      FromDate = from.ToIso8601(),
      ToDate = to.ToIso8601(),
    },
  };
}
```

Network-level problems, such as failing to resolve the target hostname, could be dealt with either way. On the one hand sticking entirely to exceptions or return values makes for a more consistent interface however in reality these low-level failures may be outside the remit of what you're trying to allow recovery of, just like an out-of-memory issue. (In the enterprise arena where I've applied this approach the only recovery from network issues is essentially to 'wait it out'. Host configuration errors can usually be spotted at deployment time by proactively probing during service start-up.)

Summary

Like any design pattern this is a solution to a problem in a particular context. If your context is building a service for which you intend to write some client code in a number of different guises, such as acceptance tests or other tooling, where you will likely need a mixture of high-level and lower-level interaction, then splitting your proxy in half may make sense. Naturally you should be wary of over-engineering your design, however at the same time keep an eye out to see if you can reduce some complexity

Listing 7

```
[Test]
public void
non_iso_8601_format_from_date_generates_an_error()
{
  // . . .
  var request = new FindOrdersRequest(accountId,
    fromDate, toDate);
  request.Content.FromDate = "01/01/2001";

  var response =
    proxy.Send<FindOrdersResponse>(request);

  Assert.That(response.StatusCode,
    Is.EqualTo(400));
  Assert.That(response.Error.Content,
    Does.Contain("ISO-8601"));
}
```

Listing 8

```
public class HttpResult<ContentType, ErrorType>
{
  public bool Succeeded { get; }
  public int StatusCode { get; }

  // On success.
  public ContentType Content { get; }

  // On failure.
  public ErrorType Error { get; }

  // For diagnostics.
  public Headers Headers { get; }
  public string Body { get; }
}
```

on the client-side by removing a little duplication before it becomes irreversible. ■

References

[1] Test Pyramid, Martin Fowler, https://martinfowler.com/bliki/TestPyramid.html

[2] TDD – From the Inside Out or the Outside In?, Georgina Mcfadyen, https://8thlight.com/blog/georgina-mcfadyen/2016/06/27/inside-out-tdd-vs-outside-in.html

[3] 'Simple Instrumentation', Chris Oldwood, *Overload* 116, https://accu.org/index.php/journals/1843

[4] 'In The Toolbox: Libraries, Console Apps & GUIs', Chris Oldwood, *C Vu* 30–2, http://www.chrisoldwood.com/articles/in-the-toolbox-libraries-console-apps-and-guis.html

[5] 'The Curse of NTLM Based HTTP Proxies', Chris Oldwood, http://chrisoldwood.blogspot.com/2016/05/the-curse-of-ntlm-based-http-proxies.html

[6] 'Causality – Relating Distributed Diagnostic Contexts', Chris Oldwood, *Overload* 114, https://accu.org/index.php/journals/1870

[7] Monitoring: Turning Noise into Signal, Chris Oldwood, *Overload* 144, https://accu.org/index.php/journals/2488

[8] 'Primitive Domain Types – Too Much Like Hard Work?', Chris Oldwood, http://chrisoldwood.blogspot.com/2012/11/primitive-domain-types-too-much-like.html

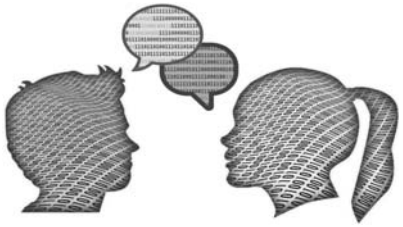
[9] 'PrimitiveObsession', *C2 Wiki*, http://wiki.c2.com/?PrimitiveObsession

[10] 'Simplicity Before Generality, Use Before Reuse', Kevlin Henney, https://medium.com/@kevlinhenney/simplicity-before-generality-use-before-reuse-722a8f967eb9

[11] *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, https://en.wikipedia.org/wiki/Design_Patterns

Code Critique Competition 112

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



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

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

Last issue's code

I'm playing around with simple increasing sequences of small numbers and seeing what various different generators produce. I had problems that sometimes the result was wrong – for example if I produce a number that overflows the integer size I'm using – and so I added a postcondition check that the output sequence is increasing. However, I've found the check doesn't always 'fire' – it works with gcc with or without optimisation but fails with MSVC *unless* I turn on optimisation. I've produced a really simple example that still shows the same problem, can you help me see what's wrong?

Expected output, showing the postcondition firing:

```
1,2,3,4,5,6,7,8,9,10,
1,3,6,10,15,21,28,36,45,55,
1,1,2,3,5,8,13,21,34,55,
1,5,14,30,55,91,140,204,285,385,
Postcondition failed!
1,5,14,30,55,91,-116,-52,29,-127,
```

Listing 1

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
class postcondition
{
public:
  postcondition(std::function<bool()> check)
  : check_(check) {}
  ~postcondition()
  {
    if (!check_())
      std::cerr << "Postcondition failed!\n";
  }
private:
  std::function<bool()> check_;
};
template<typename T>
std::vector<T> get_values(int n,
  std::function<T(T)> generator)
{
  std::vector<T> v;
  auto is_increasing = [&v]() {
    return is_sorted(v.begin(), v.end()); };
  postcondition _(is_increasing);
  T j = 0;
  for (int i = 0; i != n; ++i)
  {
    j = generator(j);
    v.push_back(j);
  }
  return v;
}
```

Listing 1 (cont'd)

```
using out = std::ostream_iterator<int>;
template <typename T>
void sequence()
{
  auto const & result = get_values<T>(10,
    [](T i) { return i + 1; });
  std::copy(result.begin(), result.end(),
    out(std::cout, ","));
  std::cout << std::endl;
}
template <typename T>
void triangular()
{
  T i{};
  auto const & result = get_values<T>(10,
    [&i](T j) { return j + ++i; });
  std::copy(result.begin(), result.end(),
    out (std::cout, ","));
  std::cout << std::endl;
}
template <typename T>
void fibonacci()
{
  T i{1};
  auto const & result = get_values<T>(10,
    [&i](T k) { std::swap(i, k); return i + k; });
  std::copy(result.begin(), result.end(),
    out (std::cout, ","));
  std::cout << std::endl;
}
template <typename T>
void sum_squares()
{
  T i{1};
  auto const & result = get_values<T>(10,
    [&i](T j) { return j + i * i++; });
  std::copy(result.begin(), result.end(),
    out (std::cout, ","));
  std::cout << std::endl;
}
int main()
{
  sequence<int>();
  triangular<int>();
  fibonacci<int>();
  sum_squares<int>();
  sum_squares<char>(); // overflow expected
}
```

Can you help (and perhaps identify one or two other issues en route)? (Unfortunately, the first few lines from `fibonacci()` didn't appear in the printed copy of *CVu*. We apologise for the confusion this caused.)

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

Critiques

Balog Pal <pasa@lib.hu>

"I produce a number that overflows the integer size I'm using – and so I added a postcondition check that the output sequence is increasing." Hmm. Signed integer overflow is undefined behavior. If we have that in the program, postcondition checks will not help – or reliably detect problems. Let's see the code if it is indeed the source of the problems.

First, just a fast read from the top making notes of other problems.

The postcondition class has a converting ctor, we probably want an `explicit` here. It takes function by value, probably following some outdated wisdom on passing predicates by value. I learned the hard way that `std::function` is a fat beast and pass it around as const ref. Though for this case we make a copy of what is passed, so the preferred way is probably indeed taking it by value, but then move in instead of creating yet another copy.

Formally we're violating the rule of 3, but the intended use form will not lead to problems. The industrial version would probably inherit from `nocopy` and possibly require `noexcept` for the check function too. I guess the idea about using the check in dtor is to cover all exit paths automatically instead of explicit points. That's fine, but we need to be extra careful about object dependencies, avoid accessing objects in the check function that were already destructed.

`check_(check)` is a form I avoid and use the same name for member variable and parameter, experience shows that this is less error prone in general, unless the compiler has reliable warning on mistakes made when writing `check_(check_)`.

`get_values` again takes function by value. What's worse, it takes function for no good reason I can imagine. It's a template, should just have another parameter for type of passed generator and take the source without repackaging it. It uses the postcondition with a lambda, the dependency chain looks fine at glance. We're calling the generator with the previous aggregate and just storing it in a vector. The check also looks okay, with all the `std::` noise around (what I would not have in my code) it is odd to see the naked `is_sorted`, but it should work via ADL.

Then we have the sample sequence functions. With obvious blatant copy-paste for the printout part. That at least should be extracted into a function, but my preferred way would separate responsibilities and just return the vector with the sequence and have an outer framework that used these and arrange the printing or whatever other work.

In the implementation I prefer the initial value look `i = 0;` and `i = 1;` instead of the `{1}` and especially the `{}` form. I also note that naming of the local and lambda param would fare well in IOCCC, so we'll look ahead to deciphering what is really going on. It also shows that the interface is weak on design: it would be so much easier if we got both the sequence number and the aggregate as argument.

`auto const & result = get_values` is okay, we get life-extension on the result coming from the function. But these days I prefer just `auto const` as copy-elision is reliable for construction, and for the `&` case, the compiler is actually allowed to make a copy if it feels like it. Making the difference moot.

Let's check the generator lambdas. For sequences, there's little chance of getting it wrong. In `triangular` we increment `i`, that is correctly captured by ref (and explicitly too).

`fibonacci` deviates from all traditions that would use 2 values and roll them. Instead it struggles with just one. It looks to work through evil trickery, at least for types covered by `std::swap`. Which should be switched to the `using std::swap;` followed by unqualified call to pick up possible user-defined overloads, not just specialisations.

`sum_squares` has `i * i++`. Wow. We were looking for UB, but not this one. :) I'm skipping this discussion; if anyone is interested, please read the gigabytes we typed over the last 30 years about `i + i++` and similar expressions. This one is not covered even by the recent refining of the order of evaluation for C++17 (P0145R3) and we have an explicit example in the execution section. But even if `++` was reliably sequenced at `i++`, the outcome would still be unspecified and we may generate the wrong number depending on whether the increment happens before or after reading the first `i`. The fixed form should create the value first and do the increment afterward.

Finally, `main` leaves nothing to note. After we have fixed the previous problems, the `int` versions should work fine. The `char` version is indeed bound to overflow, let's look where. The lambda in `sum_squares` would be the obvious candidate, but it is actually okay: the expression has `char` type inside that gets promoted to `int`, the `+` and `*` are calculated on `int` without overflow and the return type, as we didn't say otherwise is `int` too.

So, UB dodged here; `j = generator(j);` must be the place where we narrow the `int` down to `char`. The MSVC compiler actually flags it for us, but not at that place: it is lost in the preceding lines of code. And this line is not picked anywhere. Ah, and it is right too: at that spot we call `generator()` that is a `T(T)` function and we store `char` as `char`. The narrowing magic is done inside the dark corners of `std::function` where it adapts our `int(char)` lambda. And with all the type-punning around even the compiler can't tell us the point.

Anyway, what we have is not an overflow in an expression but a narrowing conversion. Which is not UB, 'just' providing an implementation-defined value for the out-of-range case. And the values we saw in the vector suggest we got negative values and CAN expect the postcondition check to fire. I definitely missed something during the code reading.

In MSVC, I can usually debug inside lambdas and other odd locations, but no luck in this case. The breakpoint on the postcondition check just never triggers, despite it being called. So I wrote some dump code inside `check`. And found that in debug mode the vector `v` is empty. Thus passing the sorted check with flying colors....

DOH. My working theory is that we are hit by NRVO/move semantics. In `get_values`, we return a vector by value, and do that from a local variable named `v`. At return point `v` is considered xvalue so returning it via `move` is fair game. And in that case the content is moved out from `v` to the temporary that gets actually returned. Then the destructors are called and the check is executed on the 'unspecified-though-valid' state of the eviscerated `v`. That happens to look empty, but could be anything really. While in release build, the compiler probably uses NRVO, avoiding the copy/move completely. `v` never exists as a local variable but gets constructed at the caller's location, so the check function sees its content.

As experiment, I tried `return std::move(v);` in `get_values`, that kills NRVO and with that MSVC release mode became consistent with debug, and `check` was looking at empty vector in both.

Though it makes sense the way we saw, I made an extra check in the standard. And indeed, 9.6.3p3 clearly states:

> The copy-initialization of the result of the call is sequenced before the destruction of temporaries at the end of the full-expression established by the operand of the return statement, which, in turn, is sequenced before the destruction of local variables (9.6) of the block enclosing the return statement.

So we better commit an extra bullet to the check list on destructor-dependencies with the case of move-from-return beyond the early destruction cases.

James Holland <James.Holland@babcockinternational.com>

The problem with the student's code is associated with the return statement of `get_values()`. It seems to me that in debug mode, the vector `v` is being affected in some way before the destructor of `postcondition` is called. It is probable that Named Return Value Optimisation (NRVO) is employed resulting in the contents of `v` being moved to the calling function. This would leave `v` in a valid but unknown state. I suspect, in this case, `v` has been left with zero length. Furthermore, I suspect that the destructor of `postcondition` reads the modified

value of `v` and finds it empty. If this is the case, postcondition will conclude that `v` is sorted and not produce a warning message. (An empty string is considered sorted.)

To my mind, this is incorrect behaviour. If NRVO takes place by means of moving (as opposed to copying) the contents of `v`, then the destructor should inspect the location to where the contents of `v` has been moved. The destructor will then conclude that the string is not sorted, as expected. Alternatively, moving the contents of `v` should be prohibited if a reference to it is made by a destructor.

When the optimiser is enabled, I assume different code is generated that either calls the destructor of `postcondition` before any manipulation of `v` or does not move `v`. Either way, this would result in the destructor of `postcondition` seeing `v` with its original contents and conclude that the vector is not ordered and produce a warning message.

It is strange (and somewhat worrying) that the program behaviour is dependent on whether the optimiser is enabled. The conclusion is that Visual Studio is producing incorrect code when the optimiser is disabled, i.e. in debug mode.

Probably the simplest way to ensure that the destructor of postcondition is invoked before the value of `v` is returned from `get_values()` is to add a pair of braces as shown below.

```
template<typename T>
std::vector<T> get_values(int n,
  std::function<T(T)> generator)
{
  std::vector<T> v;
  auto is_increasing = [&v]()
    {return is_sorted(v.begin(), v.end());};
  { // Enclose postcondition in a new scope.
    postcondition _(is_increasing);
    T j = 0;
    for (int i = 0; i != n; ++i)
    {
      j = generator(j);
      v.push_back(j);
    }
  } // Enclose postcondition in a new scope.
  return v;
}
```

The destructor of `postcondition` will be called when the newly added closing brace is encountered. This is just before the `return` statement as required. The amendment produces expected results both in debug mode and release mode.

There is one other important matter to consider, that of undefined behaviour. The function `sum_squares()` contains a lambda with the code return `j + i * i++;`. This is problematic because it is not known when, during the evaluation of the expression, the variable `i` will be incremented. The result of the expression depends on when `i` is incremented. To produce a defined result, the `return` statement should be split into three separate statements as shown below.

```
const auto k = i;
++i;
return j + k * k;
```

Finally, within the templated functions that generate the sequences, the student has defined the variable result as `auto const &`. This is perfectly valid syntax but it does give the impression that result is a reference to another variable that has been declared in the source code. This is not the case. What the compiler does, in effect, is to create a temporary variable and initialises it with the value returned from `get_values()`. The compiler then uses the value of the temporary variable to initialise the reference variable result. This is permitted because result is `const`. Had result been defined as non-`const`, the compiler would have issued an error message because changing the value of result (that refers to a temporary) does not make any sense as the temporary variable will soon disappear. The outcome of this is that initialising a `const &` variable has the same effect as initialising a `const` variable. Given this, I suggest removing the `&` from the definitions.

Jason Spencer <contact+pih@jasonspencer.org>

The headline bug here is caused by a possible optimisation that the compiler might perform and it may remove the values from `v` in `get_values` before they are checked.

Note that there is actually a typo in the print and PDF versions of this Critique – the `fibonacci()` function is missing the first three lines of the function. It was correct in the mailing-list announcement, on which this response is based.

To understand what is happening in `get_values`, we need to understand how the compiler generates code to manipulate the stack when a function is called. Typically space is reserved on the stack for the return value – since we're returning a vector let's call the object in this space `vret`. Space for the variables local to the function (including `v`) is then reserved on the stack. The variables are located in this space in the order in which they are declared within the function, and they're constructed in this order too. They are destructed in the opposite order to their construction.

When the return statement is executed, `v` is either copied (by copy constructor), or (since C++11) moved into `vret`. It is also possible for the copy/move to be elided, and for `v` to actually be in the space assigned for `vret` – this is called Return Value Optimisation (RVO) – specifically Named RVO (NRVO) because we're not constructing the return value in the return statement, we're returning a variable. This copy elision can occur whether there are side-effects of copying/moving or not.

Whether return by copy construction, move construction, or copy elision occurs depends a lot on the compiler, the compiler version, the flags used and the optimisation level, hence the complicated nature of the behaviour the student found.

The student was expecting `_` to be destroyed before `v`, and it will be. But we now face three possible scenarios at the destruction of `_`:

- `v` was *copied* to `vret` at the `return` statement – in which case, `v` is still valid and the test behaves as expected.
- `v` was *moved* to `vret` at the `return` statement – in which case, `v` is now empty and the test ***always*** passes because a vector with less than two values still causes `is_increasing` to return true.
- `v` *is* `vret` due to NRVO – in which case, the test behaves as expected.

It is the second scenario that prevents the error from appearing when it is expected.

The details of copy/move/RVO optimisations can be found at [1] and [2].

The other issues I'd comment on are:

- I'm not sure using `postcondition` to test is a good idea. The destructor, and therefore the test, will also be executed if an exception is thrown and not caught in `get_values` (thrown by a generator perhaps). The student may want to test `std::current_exception` in `~postcondition` before calling the test. And of course the test can't throw anything, so should probably be `noexcept`. Consider a templated function as a test that doesn't rely on RAII.
- Implicit casting of output to `int`:

  Since `out` is an alias for `std::ostream_iterator<int>` and it is used in the `sequence<T>`, `triangular<T>`, `fibonacci<T>` and `sum_square<T>` templates, the output sequence values, which should contain values of type `T`, are cast to `int` before being output. So if `sum_squares<float>;` is included in `main()` the values are truncated and printed as `int`s.
- The generator functions should do type checking on `T`, and potentially disable themselves for types that don't support the operators used, or for types that would produce nonsensical results.

  For example, `fibonacci<std::string>()` will work, but `fibonacci<std::complex<float>>()` won't. `T` has to

support one or more of: `operator++`, `operator+` and must have a `std::swap` overload depending on the generator. `T::operator<` is also required for the `is_increasing` test. Consider an `enable_if` [3] with `std::is_arithmetic` [4] and `std::is_swappable` [5] in the simplest case.

In terms of design I'd look more into the overflow checking and the generators.

The issue of robustly checking overflow is a complicated one and often involves tests alongside the operations, rather than testing the result, as was the case here. Overflow detection isn't that difficult to do in theory – most CPUs have an overflow flag (OF) which is set when the last ALU operation resulted in a value that overflowed the output register. You might also want to check the carry flag (CF) [6]. The problem is that there's no straightforward way of detecting this in C++. We could write inline assembler functions that perform the arithmetic operation and copy the OF flag to a local variable and act based on the state of that, but this would be architecture specific. It also limits optimisation – for example a typical compiler would swap a multiplication by a constant 8 into a left shift by 3, but this code wouldn't.

Another approach [7] is to, for each operation type, check that the output variable has enough capacity – this typically involves finding the position of the most significant set bit. An unsigned integer multiplication of variables A and B will never use more than MSB_A+MSB_B bits in the output, where MSB_A is the position of the MSB in A. The unsigned integer addition of A and B will never use more bits than MAX(MSB_A,MSB_B)+1. And so on.

Most compilers also have options to trap on overflow for arithmetic, but it can be complicated to handle this at runtime, and is compiler specific.

There are some mathematical functions in C++ that do support overflow detection in a portable and robust way. For example `std::math_errhandling` [8] can be used to detect some overflow and other conditions for floating point types.

`std::overflow_error`, while sounding interesting is only thrown by `std::bitset` conversion functions, and not general purpose arithmetic, but there's no reason not to use it in bespoke overflow testing code.

Another approach is to create a special type that wraps the arithmetic type, overloads the arithmetic operators, and checks for overflow after every operation. For example [9].

Yet another approach is to perform the operation on a larger (similarly signed) data type and cast to the requested type at the end, checking for truncation after the cast. This might not however catch all cases of overflow at every intermediary operation (or in fact the overflow may be required behaviour – an implicit modulo).

I'm not a fan of using `is_increasing` because although the program spec says the sequences are increasing, it is possible for aliasing to occur – for example, in a `sum_squares<uint8_t>` sequence, at the sixteenth term 256 is added to the running total – this overflows, but the total remains the same, and that term at least would pass the test (although earlier terms might not). I'm sure there are many other such examples.

Regarding the generators – they're not very versatile the way they are written here. I'd consider implementing them either as iterators, or as generator functors. In the current design the state of the sequence is store in variable `i`, which is a nice design, but if it were stored in an object then the object can be copied, reset, etc.. something like this:

```
template <typename T>
class arithmetic_progression_ {
  T n;
  T step;
public:
  typedef T result_type;
  arithmetic_progression_ (const T n = 0,
    const T step = 1) : n(n), step(step) {}
  T operator()() {
    n += step;
    return n;
  }
  T get_last() { return n; }
};
template <typename T> class triangular_ {
  T n;
  T level;
public:
  typedef T result_type;
  triangular_ (const T n = 0,
    const T level = 1) : n(n), level(level) {}
  T operator()() {
    n += level++;
    return n;
  }
  T get_last() { return n; }
};
template <typename T> class sum_squares_ {
  T n;
  T level;
public:
  typedef T result_type;
  sum_squares_ (const T n = 0,
    const T level = 1) : n(n), level(level) {}
  T operator()() {
    n += level * level++;
    return n;
  }
  T get_last() { return n; }
};
```

This is much more versatile because you can now generate as many values as you like (`25` in the example below):

```
std::vector<int> v;
std::generate_n ( std::back_inserter(v), 25,
  triangular_<int>() );
std::copy ( std::begin(v), std::end(v),
  std::ostream_iterator<int>(std::cout, ",") );
std::cout << '\n';
```

Or you can use an iterator wrapper (such as the Boost Generator Iterator Adaptor [10]) to print the sequence directly to any output stream:

```
arithmetic_progression_<int> a_p;
std::copy_n (
  boost::make_generator_iterator( a_p ), 15,
  std::ostream_iterator
    <decltype(a_p)::result_type>
    (std::cout, ",") );
std::cout << '\n';
```

Something to also consider is whether it wouldn't be beneficial to generate sequences at compile time. In the past this meant using templates [11], but there are now more options with `constexpr` [12]:

```
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

When it comes to generators inspiration can be taken from C# and Python. Paolo Severini [13] has an extensive write-up comparing C# and C++ generators and their implementation. He also looks at co-routine approaches.

References

[1] https://en.cppreference.com/w/cpp/language/copy_elision
[2] https://shaharmike.com/cpp/rvo/
[3] https://eli.thegreenplace.net/2014/sfinae-and-enable_if/
[4] http://en.cppreference.com/w/cpp/types/is_arithmetic
[5] http://en.cppreference.com/w/cpp/types/is_swappable
[6] http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt
[7] https://stackoverflow.com/questions/199333/how-to-detect-integer-overflow
[8] http://en.cppreference.com/w/cpp/numeric/math/math_errhandling

Program Challenge Report 3 and Challenge 4

Francis Glassborow comments on his last challenge and presents a new one.

I am a little disappointed by the (lack of) response to this challenge. I had very carefully phrased the challenge so as not to make it an actual 'quine' (self-replicating program), I also avoided mentioning 'quine' because that makes it trivial, just search the net.

As in all my challenges, anything not expressly forbidden is permitted. The simple answer which will certainly run on your machine is to write a program that reads in its own source code file and then prints it out.

If you are writing a self-replicating program that will run anywhere (including on my machine) you have an important consideration:

Can my machine execute a program compiled on your machine?

If we assume the answer is 'no' then it effectively means that the program has to be delivered as source code. Once you understand the significance of that the problem reverts to writing a program that will produce the correct output on the machine on which it is compiled.

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 (continued)

[9] https://accu.org/index.php/journals/324
[10] https://www.boost.org/doc/libs/1_62_0/libs/utility/generator_iterator.htm
[11] https://www.jasonspencer.org/articles/series/
[12] https://en.cppreference.com/w/cpp/language/constexpr
[13] https://paoloseverini.wordpress.com/2014/06/09/generator-functions-in-c/

Commentary

As all three entries explain, the cause of the differences in behaviour is an interaction between destructors and generating the return value. The 'feature' was added to C++11 when move semantics were added to the language and returning from a named local variable was allowed to *move* the value. I do not know at what point in the process anyone realised that this change could cause the difficulties that this critique demonstrates.

I think the critiques between them covered most of the points in the critique.

The winner of CC 111

The critiques all detected the problem with the unpredictable behaviour, but James additionally provided a solution to the troublesome behaviour – using an extra scope to ensure the destruction occurs before the return.

As Pal stated, the problem with signed overflow is that this produces undefined behaviour and attempting to detect this can be troublesome. This is an active area of discussion on the C++ committee...

Using destruction to verify post-conditions can have problems, as Jason points out, because of the danger of throwing an exception while unwinding from an exception, which will simply terminate the program.

Both Pal and James pointed out the UB in **`sum_squares`** in the expression **`j + i * i++`** but James additionally provided some code for a correct replacement.

All three critiques gave the code a good look, but on balance I consider James' critique was the most helpful to the person with the problematic code so I have awarded him the prize for this issue.

Code Critique 112

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

Further to articles introducing D, I am attempting to use the event-driven Vibe.d server library, which I understand to be like a native version of Python's Tornado and potentially a 'killer app' of D as I haven't seen any other mature event-driven server library in a compiled language.

I would like to build a back-end server that performs some processing on the body of the HTTP request it receives. To begin with, I would like it to simply echo the request body back to the client.

My code works but there are three problems: (1) when I issue a POST request with lynx, 3 spaces are added to the start of the response text, (2) I cannot test with nc because it just says 404 and doesn't log anything, and (3) I'm worried that reading and writing just one byte at a time is inefficient, but I can't see how else to do it: I raised a 'more documentation needed' bug at https://github.com/vibe-d/vibe.d/issues/2139 but at the time of writing nobody has replied (should I have used a different forum?)

The code is in Listing 2.

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

```
import vibe.d;
void main() {
  auto settings = new HTTPServerSettings;
  settings.port = 8080;
  listenHTTP(settings, (req, res) {
    ubyte[1] s;
    while(!req.bodyReader.empty()) {
      req.bodyReader.read(s);
      res.bodyWriter.write(s);
    }
  });
  runApplication();
}
```

Now I suspect that many of you thought I was asking for a 'quine' in C++. Without knowing the name it is very hard to cheat by looking the answer up on the internet. Our ancestors were right in the belief that knowing something's true name gives you power over it.

Here is the only response I got.

Response from James Holland

I thought one way was to write a very simple C++ program that prints a string that contains the source code of the program, something like the following.

```
#include <iostream>
int main()
{
  std::cout << "#include <iostream>\nint
  main()\n{\n  std::cout << \"\";\n}\n";
}
```

The trouble is the program is not quite complete because its output is as shown below.

```
include <iostream>
int main()
{
  std::cout << "";
}
```

There is nothing in the quoted string so let's fix that.

```
#include <iostream>
int main()
{
  std::cout << "#include <iostream>\nint
  main()\n{\n  std::cout << \"#include
  <iostream>\\nint main()\\n{\\n  std::cout
  << \\\"\\\";\\n}\\n\";\n}\n";
}
```

We are not there yet as this program produces the following.

```
#include <iostream>
int main()
{
  std::cout << "#include <iostream>\nint
  main()\n{\n  std::cout << \"\";\n}\n";
}
```

Which produces the following.

```
#include <iostream>
int main()
{
  std::cout << "";
}
```

There is nothing in the quoted string so let's fix that. But wait. We have been here before. We can keep on adding to the string forever and still not have a program that reproduces itself. This approach is not going to work. This is where I cheated and tried to discover how other people have solved this problem.

Apparently, one method is to divide the program into two parts. The first part contains data that represents the second part of the program. The second part contains code that firstly prints the data as it is presented in the first part of the program and secondly uses the data (again) to print the second part of the program in the form of its source code. (I am sure you are still with me.) The result is a printout of the complete program's source code. Such a program (one that outputs its own source code) is called a Quine, after Willard Van Orman Quine (1908–2000), or so I have discovered. My attempt is in Listing 1.

It would be difficult to produce the data by hand and so I have written a small program – see Listing 2 – to do the job for me. It takes as input the program source code (minus the first part) and produces the equivalent in decimal form. The program's output is then pasted into the original source code to form the complete Quine program.

Listing 1

```
const char data[]{
  35, 105, 110, 99, 108, 117, 100, 101, 32, 60,
  105, 111, 115, 116, 114, 101, 97, 109, 62, 13,
  10, 35, 105, 110, 99, 108, 117, 100, 101, 32,
  60, 115, 116, 114, 105, 110, 103, 62, 13, 10,
  13, 10, 105, 110, 116, 32, 109, 97, 105, 110,
  40, 41, 13, 10, 123, 13, 10, 32, 32, 115, 116,
  100, 58, 58, 99, 111, 117, 116, 32, 60, 60, 32,
  34, 99, 111, 110, 115, 116, 32, 99, 104, 97,
  114, 32, 100, 97, 116, 97, 91, 93, 123, 34, 59,
  13, 10, 32, 32, 115, 116, 100, 58, 58, 115,
  116, 114, 105, 110, 103, 32, 115, 101, 112,
  97, 114, 97, 116, 111, 114, 59, 13, 10, 32, 32,
  105, 110, 116, 32, 99, 111, 108, 117, 109, 110,
  95, 99, 111, 117, 110, 116, 32, 61, 32, 48, 59,
  13, 10, 32, 32, 102, 111, 114, 32, 40, 99, 111,
  110, 115, 116, 32, 105, 110, 116, 32, 99, 32,
  58, 32, 100, 97, 116, 97, 41, 13, 10, 32, 32,
  123, 13, 10, 32, 32, 32, 32, 115, 116, 100, 58,
  58, 99, 111, 117, 116, 32, 60, 60, 32, 115,
  101, 112, 97, 114, 97, 116, 111, 114, 32, 60,
  60, 32, 99, 59, 13, 10, 32, 32, 32, 32, 115,
  101, 112, 97, 114, 97, 116, 111, 114, 32, 61,
  32, 34, 44, 32, 34, 59, 13, 10, 32, 32, 32, 32,
  105, 102, 32, 40, 43, 43, 99, 111, 108, 117,
  109, 110, 95, 99, 111, 117, 110, 116, 32, 61,
  61, 32, 49, 54, 41, 13, 10, 32, 32, 32, 32,
  123, 13, 10, 32, 32, 32, 32, 32, 32, 99, 111,
  108, 117, 109, 110, 95, 99, 111, 117, 110, 116,
  32, 61, 32, 48, 59, 13, 10, 32, 32, 32, 32, 32,
  32, 115, 101, 112, 97, 114, 97, 116, 111, 114,
  32, 61, 32, 34, 44, 92, 110, 32, 32, 32, 32,
  32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32,
  32, 32, 34, 59, 13, 10, 32, 32, 32, 32, 125,
  13, 10, 32, 32, 125, 13, 10, 32, 32, 115, 116,
  100, 58, 58, 99, 111, 117, 116, 32, 60, 60, 32,
  34, 125, 59, 92, 110, 92, 110, 34, 59, 13, 10,
  32, 32, 102, 111, 114, 32, 40, 99, 111, 110,
  115, 116, 32, 97, 117, 116, 111, 32, 99, 32,
  58, 32, 100, 97, 116, 97, 41, 13, 10, 32, 32,
  123, 13, 10, 32, 32, 32, 32, 115, 116, 100, 58,
  58, 99, 111, 117, 116, 32, 60, 60, 32, 99, 59,
  13, 10, 32, 32, 125, 13, 10, 125, 13, 10};

#include <iostream>
#include <string>

int main()
{
  std::cout << "const char data[]{";
  std::string separator;
  int column_count = 0;
  for (const int c : data)
  {
    std::cout << separator << c;
    separator = ", ";
    if (++column_count == 16)
    {
      column_count = 0;
      separator = ",\n                    ";
    }
  }
  std::cout << "};\n\n";
  for (const auto c : data)
  {
    std::cout << c;
  }
}
```

Francis states there are two categories of solution; one that will run on my computer and one that will run on his computer. I have no idea what computer Francis uses. It should not really matter. All that is required is for the computer to have a standard conformant C++11 compiler and for the source code (Quine.cpp) to be compiled and run. The resultant executable should print its own source code on any system. If the output is directed to a file, the content of the file will be identical to the original source code.

Francis replies

James is quite right up to a point. The problem is that his program assumes that my machine is using ASCII and there is no such requirement in the Standard (nor could there be). I do not have an Apple computer to hand but I strongly suspect that it would fail to produce the required output because it codes End of Line differently. However that is being a little picky. More significantly there are still machines around that use EBCDIC. I leave the reader to research that. It is a common flaw in those writing portable code to assume that all computers use ASCII or Unicode (where the base page is, if I recall correctly, ASCII).

I think that fully portable quine source code is actually quite difficult and almost certainly requires some tool support to deal with variant text coding.

Then there is the point that I think James missed so perhaps it was missed by many readers: A conforming C program is almost certainly going to be a C++ one as well. At least that is one of the objectives of WG21 (The ISO C++ Standards Committee). That objective is very frustrating when developing new core facilities for C++. As we know, C++ can easily add new types via library implementations (think of support for complex numbers) and C has to do that by adding to its basic type system. Where both languages want the same type but both want (need) to do it their way providing a compatible mechanism can be hard.

Now the simplest C quine I know is:

```
#include <stdio.h>
char*s="#include <stdio.h>%cchar*s=%c%s%c;%cint
main(){printf(s,10,34,s,34,10,10);}%c";
int main(){printf(s,10,34,s,34,10,10);}
```

It also depends on the program encoding newline as 10 and double quotes as 34. It has the advantage of having only those two dependencies and so is relatively easy to adapt to systems using other character encodings.

I wonder if anyone can refine the above program so that it becomes independent of the character coding. Careful use of escape sequences might achieve the desired result and so produce a program that will run on any machine.

Challenge 4

I was wondering what to challenge the readers with next. While still undecided I went to the April meeting of the Oxford branch of ACCU. One of the other attendees (sorry I have forgotten who it was) offered this one:

Write a program that outputs the numbers from 1 to 100 without using **`if`**, **`for`**, **`switch`** or **`while`**.

I cannot think of a way to do this in C other than the trivial solution of simply writing tedious outputs, but I certainly can manage it in C++. However, I am going to add a small refinement, your program must take two integer inputs: first and last. It must output the integers from the first to the last. Note that I have not specified that the first is lower than the last.

For starters try the simple form where first = 1 and last =100. Some of you may think 'recursion', but that requires a conditional to ensure termination, or does it?

There is also a second form of solution that relies on the way that C++ initialises arrays.

Over to you. Please submit your solutions even if they are just partial ones. Sometimes ideas that do not quite work can be refined when others see where you are trying to go.

Listing 2

```
#include <fstream>
#include <iostream>

int main()
{
  std::ofstream out_file("Quine.dat");
  out_file << "const char data[]{";

  std::ifstream in_file("..//Quine.cpp");

  std::string separator;
  int column_count = 0;
  char c;
  while (in_file.get(c))
  {
    out_file << separator.c_str() <<
static_cast<int>(c);
    separator = ", ";
    if (++column_count == 16)
    {
      column_count = 0;
      separator = ",\n                    ";
    }
  }
  out_file << "};";
}
```

View from the Chair

Bob Schmidt
chair@accu.org

ACCU has been very busy since my last report.

2018 AGM and election

ACCU's Annual General Meeting was held on Saturday, 14th April. The draft minutes of the meeting are available to members online [1], so I won't go into too many details here. The most important part of the AGM, from my perspective, was the nomination and election of Patrick Martin to the role of Secretary. (More on Patrick below.) This, in addition to the re-election of the rest of the executive committee (Robert Pauer, Treasurer; Matt Jones, Membership Secretary) means we can continue to conduct all regular ACCU business for the next term.

The following people were also elected or re-elected to non-executive committee roles: Nigel Lester, Local Groups; Ralph McArdell, At-large; Roger Orr, Publications; and Emyr Williams, Standards. Guy Davidson and Niall Douglas were approved to continue as Auditors.

ACCU 2018

ACCU held its annual conference at the Marriott City Centre in Bristol in April. I'm pleased to report that the conference was a success. Attendance was higher than ever – 450 total attendees, with an average of 320 per day. The attendees and speakers were also the most diverse they have ever been.

Our conference chair, Russel Winder, and the members of his conference committee are to be commended on the fine job they did putting the conference program together. Julie Archer and her team from Archer Yates Associates (AYA) continue to perform their organizational magic in keeping the conference running smoothly.

Conference committee: Gail Ollis, Roger Orr, Felix Petriconi, CB Bailey, Anastasia Kazakova, Jon Kalb, Nina Dinka Ranns, Francis Glassborow.

AYA team: Charlotte Tanswell, Laura Nason, Helen Wormall, Marsha Goodwin.

My thanks (and hopefully, all of yours, too!) go out to Russel, Julie, and their respective team members for another great conference.

Next year's conference is already in the planning stages. I'll have more information for you on ACCU 2019 as the conference gets closer.

Committee spotlight

We have two new volunteers!

Patrick Martin volunteered to run for, and was elected, ACCU Secretary at the AGM. Patrick is a software engineer, music fan and lapsed science PhD, in roughly that order. Patrick has worked for SunGard, Deloitte and Touche, Bloomberg, Aquila Group Holdings, among others. He has done a fair amount of 'backend' and 'front-end' work in various arenas and the inevitable consequence is chalking up exposure to a number of tools and methodologies. He also has ambitions to write on technical matters that interests him. [Start with *CVu* and *Overload*, Patrick!]

Daniel James volunteered to look into the process by which our magazines are converted to ePub format, which haven't been available since the web editor position went vacant last year. Daniel is a software designer and programmer specializing in information security. He has been a member of ACCU since 2000, and is a regular attendee of the conference. Daniel also presented at ACCU 2018 [2].

Please join me in thanking Patrick and Daniel for volunteering their time to ACCU.

GDPR

Nigel Lester has done an excellent job doing a deep dive into the intricacies of the EU's new General Data Protection Regulation, and developing ACCU's response to keep us in compliance. As a result we have a new page on our website – Privacy Policy – located in the ACCU Menu [3]. Still in draft form and going through refinements as of this writing, ACCU's Privacy Policy enumerates the data we collect; who we do and do not share it with; and your rights and our responsibilities under the GDPR.

Thanks to Nigel and everyone who assisted him in the drafting of our Privacy Policy.

ACCU World of Code

Several months ago Andy Balaam introduced us [4] to his blog aggregator at Planet Code [5]. At the time he volunteered to help ACCU set up a similar aggregator on our site. Webmaster Jim Hague worked with Andy to get our ACCU aggregation page up and running [6]. If you would like your blog to be considered for inclusion in the aggregation, please contact Jim at webmaster@accu.org. Thanks to Andy and Jim for getting additional content on our web site.

Twitter

If you pull up our website (www.accu.org) or our Twitter feed (www.twitter.com/accuorg) you might have noticed a small increase in our Social Media activity. I've been trying to post something every week in order to keep our front-page content fresh. If you have any ideas for additional content – don't tell me, volunteer for the Social Media position on the committee!

Call for volunteers

We still need volunteers. Last issue I promised to drop your name if you volunteered, and sure enough, in this issue I've put Patrick's and Daniel's names up in ~~bright lights~~ not-so-bright print. You could be next!

Web Editor	Book Reviews
Social Media	Publicity
Study Groups	

On a personal note, this marks the start of my third year as Chair. I'd like to express my thanks to everyone who voted in our election this past March, and also thank you for once again putting your trust in me. It is an honour and a privilege to serve you.

References and notes

[1] Draft 2018 AGM Minutes: https://accu.org/content/agm/AGM-2018-Minutes(Draft).pdf

[2] 'Cryptography for Programmers', Daniel James: https://www.youtube.com/watch?v=3wiYUEyXC00

[3] ACCU Privacy Policy: https://accu.org/index.php/privacy_policy

[4] Planet Code, *CVu* 29-5, November 2017: https://accu.org/index.php/journals/2431

[5] Planet Code: http://www.artificialworlds.net/planetcode/

[6] ACCU World of Code: https://blogs.accu.org/

"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