the magazine of the accu

www.accu.org

{cvu}

Volume 31 • Issue 6 • January 2020 • £4.50

Features

Regulars

{cvu}
Volume 31 Issue 6
January 2020
ISSN 1354-3164
www.accu.org

On not doing anything

No-one likes to be called in the middle of the night with the news that some system is down and needs to be fixed *right now.* Of course, not everyone reading this works in an environment where that happens, but I suspect there are quite a few who have experienced the weary trudge into the office in the middle of the night to try and find out what's going on. Few of us operate at our best when roused from our beds under such conditions. Which is why it's exactly this example that I give when I am asked what I mean by saying I'm a Lazy Developer.

Much of being lazy in this respect is about getting machines to do the work so that I don't have to. Things like running the tests, or trawling the source code for simple errors. It's a bit more subtle than the phrase "so I don't have to" suggests: the machine is not only quicker than I would be, doing it by hand, it's also much more reliable. Tedious, repetitive tasks are very prone to human error, so automating the boring stuff seems to me to make perfect sense.

The same thinking applies to software deployments. I like to be as easy as not even having to click a button, if I can, but I can manage a single button click. Sure, I can probably follow a page or two of instructions on scripts to run, files to copy, servers to log into, and so on, but truly, why should I? Running scripts and copying files to places are all things machines do very well, and better than I can.

There is much more to being lazy than not having anything to do, however. It's all about confidence - mine, my team's, by boss's, and so on. I'm prepared to put a great deal of effort into being lazy *later*, if it means everyone is more confident things will just work. And especially if it means I'm confident I'm not going to get called up at 3am to fix a problem *I* caused because I wasn't paying enough attention.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 32.1: 1st February 2020
C Vu 32.2: 1st April 2020

Overload 155: 1st March 2020
Overload 156: 1st May 2020

WRITE FOR C VU

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

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

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Restaurant C++ and Pidgin Python

Pete Goodliffe looks at the idioms of language.

I guess that I'm a typical Briton. I am a tea-drinking fish-and-chip junkie who goes red 35 seconds after exposure to sunlight. I don't own any bulldogs, although I've been known to spout plenty of bull. And my grip of foreign languages is typically poor. I have what you'd politely call restaurant-French. Any slightly taxing conversation involves me gesticulating wildly, whilst repeating the same sentence *louder* and *more slowly* until the conversant finally understands what I'm saying. Or politely pretends that they do. It works. At least, until they give up and talk back to me in English.

So, not a natural French-speaker, then.

Just how good is your mastery of your programming languages? Is it like my French – do you have restaurant Java, or tourist Python? Or do you really *know* the languages you use? Do you need phrase books and cheat sheets, or are you fluent, knowing the natural language idioms? Do you have to screw your eyes up and think hard when crafting code to make sure that it makes sense? Or is your coding fluent? Do other readers understand what you write? Or do you write pidgin-code, translating idioms from a different language into the one you're writing?

A fluent French speaker doesn't think in English and convert their thoughts to French before speaking them. They *think* in French, and what they speak aloud comes naturally. There is no mismatch of idioms. There is no need for internal translation of the English idiom to the French idiom. To be truly effective in a programming language, to be able to craft Really Good Code, you have to operate in the same way.

There is a very real difference between fluent, idiomatic code, and pidgin-code. Stop for a moment and consider the number of ways that Listing 1 (Ouch!) offends you. Count them all. How is Listing 2 better? Is it any better? And what language is each one written in, anyway?

When we're staring at code, we like to see clear structure and code patterns that we're familiar with – the natural idioms of that language. Certain code patterns are offensive – they naturally cause us to sit up, take note, and apply extreme caution. The mere sight of a `goto` invokes the gag reflex; when we see messy and inconsistent layout, we feel bile rising; global variables cause an allergic reaction; and in the face of illogical and unmalleable structure, we're gripped with the urge to run away very quickly. This kind of judgement is a part of what sets great programmers apart from the merely adequate ones. And we all want to be great programmers, don't we? How advanced do you think your internal quality meter is?

Listing 1

```
bool ouch()
  {
    if (do_something() == FAILED)
   goto fail_1;
    if (do_something_else() != OK)
   goto fail_2;

    return true;
      fail_2:
    tidy_up_second_thing();
fail_1:
    tidy_up_first_thing();
return false;
}
```

Listing 2

```
bool better()
try
{
    do_something();
    do_something_else();
}
catch (...) { return false; }
```

Beauty is in the idiom of the beholder

It's interesting to note that our sense of 'beauty' is shaped by familiarity – by the prevalent *idioms* of the implementation domain. What constitutes natural and beautiful code differs from language to language. Idiomatic C code is quite a different beast from idiomatic Python. Listings 3 (idiomatic C code), 4 (the equivalent idiomatic Python code), and 5 (equivalent non-idiomatic Python code) illustrate this. The Python code in Listing 4 is idiomatic, but it could have been written like Listing 5 – that listing is a more direct translation of the original C code. But it's *not* idiomatic Python, and it doesn't look or feel 'right' as a consequence. Listing 5 is more verbose, and consequently harder to comprehend and more likely to harbour bugs.

And that's just a *really* small example. (After all, small examples are idiomatic for magazine columns.)

We become accustomed code that fits the natural idioms of the language we're using. Non-idiomatic code is most often what sets our internal alarm bells ringing. And rightly so. Idioms don't just look nice, they help us to write safe, correct code, avoiding the subtle pitfalls in the language. Like old wives' tales, there is a body of collected wisdom in our programming language idioms that is perilous to ignore.

Learning the specific idioms of a language are a rite of passage, and mark your mastery over the language, like a journeyman programmer becoming getting acquainted with his tools.

No idiom is an island

Important as they are, idioms are not sacred. Nor are idioms fixed and immutable. Fashion doesn't stand still; over time tastes change. Some

Listing 3

```
int list[] = { 1, 2, 3, 5, 8 };
for (int n = 2; n < 4; n++)
{
   do_something(list[n]);
}
```

Listing 4

```
list = [1, 2, 3, 5, 8]
for element in list[2:4]:
   do_something(element)
```

Listing 5

```
n = 2
while i < 4:
   do_something(list[n])
   n += 1
```

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

classic programming idioms have dated: Hungarian Notation used to be a conventional, safe, and well-regarded practice. These days it is not merely passé, but socially unacceptable: the modern equivalent of software leprosy.

Idioms don't have all the answers, either. Multiple idioms compete over the same coding practice, and no one is necessarily right. Some aspects of code beauty are not clear cut and are continually the root of religious debates. For example, what is your opinion on the following; do you even care about them?

- Do you indent with spaces or tabs?
- If you use C-like languages, where do you put your braces?
- Do you advocate *single-entry-single-exit* functions?
- Do you prefer the functional coding style, where functions have no side effects?

The functional coding paradigm, in particular, has gained mindshare recently as functional programming enjoys a renaissance and the industry begins to learn how powerful and tractable some of the functional idioms are. It's informed the design and common usage idioms of many traditionally 'non-functional' programming languages.

Idiom idiocy

But sometimes the desire for elegant, beautiful, idiomatic code can trip us up. Well intentioned use of idioms can bite you. Here's a cautionary tale involving C++. Now, C++ idioms are particularly amusing. The oft-cited Perl mantra is *There's more than one way to do it*. C++ is like that for idioms – there's always more than one idiom for anything in C++, and you can bet that each has zealots who fervently believe that theirs is the Only Right Way To Do It.

One of the most basic, contentious, C++ idioms is the naming member variables. Many of these idioms involve the subtle incursion of Hungarian Notation. See Listing 6 for examples.

Many C++ programmers prefix member variables by an underscore. However, this is dubious practice as the language standard reserves many identifier names beginning with an underscore. Class member variables are not actually one of the reserved cases, but this convention sails dangerously close to the wind. Also common is the **`m_`** prefix (where **`m`** stands for member). I've even seen the disgustingly cute **`my_`** prefix, e.g. **`my_member`**. Euch!

So, what do the C++ experts do? Herb Sutter advocates a trailing underscore on member variable names. Andrei Alexandrescu's books have sadly followed his lead here. I have to admit that I have a personal dislike for this approach as the variable names read *very* strangely.

Scott Meyers is the sensible advocate of minimalism – he writes the variable name, the whole name, and nothing but the name (see the end of Listing 6). To my mind, this approach makes sense. If you need any extra indication of memberyness then you probably have code that is too hard

Listing 6

```
class ExampleMemberNames
{
  // Many programmers prefix member variables by
  // "_"
  int _common;

  // Also common is an "m_" prefix
  int m_member;
  int m_another;

  // Herb Sutter advocates a trailing underscore
  int sutter_;

  // Scott Meyers is the sensible advocate of
  // minimalism
  int meyers;
};
```

Listing 7

```
class Foo
{
private:
  int thing;

public:
  // How would you name the ctor parameter?
  Foo(int thing_in) : thing(thing_in) {} // (1)
  Foo(int t) : thing(t) {}               // (2)
  Foo(int thing) : thing(thing) {}       // (3)
};
```

Listing 8

```
Foo::Foo(int thing) : thing(thing)
{
  if (thing == 1)
    thing = 2;
}
```

to read – your function has too many parameters, or your class is too large. Fix that problem, don't mask it with silly variable names.

Why does this simple naming issue matter? Well, it shouldn't, until we combine the Meyers Minimal Member Moniker Mechanism with another idiom. When constructing a C++ class, members are given their initial values in the member initialisation list. There's another naming minefield here: three of the options are enumerated in Listing 7.

They are only subtly different, are functionally equivalent, and each seems perfectly adequate. They are all common in modern C++ code. I've most often seen idiom 3; it's nicely symmetric and doesn't introduce another unnecessary name into the code.

Great.

Well, not quite. Idiom 3 has a hidden sting in its tail. Sure enough, in Listing 7 it works just as advertised. But consider what happens when you need some slightly more complex constructor logic, like Listing 8.

What is the value of the **`thing`** member when a **`Foo`** is constructed with the value 1? It's 2, right? Well, no it isn't. It's 1. *But how can that be* I hear you ask? The name **`thing`** inside the constructor is bound to the *constructor parameter*, not to the class' member variable. If you wanted to assign the member, you must write

```
this->thing = 2.
```

otherwise you're just writing to a temporary variable that will shortly be thrown away. This is a subtle but nasty way to introduce obscure bugs into your codebase. So, there you have a collision of idioms. Some idioms are bad for you! Hurrah for C++, and hurrah for idioms.

How could you alleviate this problem? There are many ways. For example, you could make the thing parameter **`const`** in the constructor implementation. But for built-in types passed by value, this isn't idiomatic! How else could you avoid it? (You could always choose to follow a different set of idioms. Or use a different language.)

The moral of the story

It's important to consider the idioms of the language you're working in – and to gauge the beauty and quality of code against the familiar idioms it should adhere to. Common language idioms have several important uses: they help to show the elegance, beauty, and artistry of a piece of code. They help you to write code that seems familiar and easy to work with, and they (usually) help you to avoid simple bugs. You can gauge your mastery of a language by how well you know its idioms.

It's particularly important to understand why these idioms exist. Learn to *think* in the programming language you're using, to think in terms of it idioms.

But don't blindly trust idioms. Idioms can be flawed. Always use your brain. Of course, if this seems like too much work, perhaps you should give up and produce boring, ugly code. Or learn to speak French properly instead. ■

Why I Don't Develop for iOS

Silas S. Brown tells a cautionary tale of the App Store business model.

When I wrote a utility to help a local cancer-research team organise an experiment they were doing, it occurred to me that other medical research labs might also benefit from the utility, so I contacted someone I know who worked at another lab to see if they were interested, but she said her lab does not risk running software unless it's been published in a peer-reviewed bioinformatics journal. Fair enough, I thought, so logically I set about getting a peer-reviewed publication about my utility. This has been published by Oxford University Press and I understand some labs are now starting to use it.

Now, academic journals come in two types: traditional closed-access, where the reader (or the reader's institution) pays to see the article, and modern open-access, where there is no charge to the reader but the author (or the author's institution) pays a publication charge. We opted for an open-access journal, but as my current affiliation with the university is merely that of being a teaching assistant in a different department, I realised that trying to get the university to pay my open-access publication fee would take a very long time (if it could be done at all), and I didn't want the whole world's cancer research to be waiting for that, so I just paid it out of my own pocket and said 'that should cover my donations to cancer research for a while'.

When I paid that open-access fee, there were two important things I knew: (1) that the paper had already been accepted in principle and (2) that the publication is permanent.

Paying for consideration

I might not have been so keen to pay the publication fee myself if it had merely been an 'entrance fee' that must be paid up-front before they even looked at my submission, with no possibility of refund if I'm rejected. That is what happens on Apple's App Store, so if you don't have too much money to burn on the expensive up-front subscription costs, not to mention equipment costs (you need to keep buying the latest Mac hardware to stay current), you had better be highly confident of your chances of passing Apple's app selection panel before you pay up to find out. My app was based on a browser, and I'd already had it rejected by Amazon (apparently because they don't want any competition to their own Silk browser on their store), so despite its popularity on Android I wasn't at all confident enough to risk paying for Apple to consider it.

Now, people do pay examination fees for driving tests. I chose 'driving tests' for this example because the result is a strict pass-or-fail (no B-grades etc: let's not consider 'Pass Plus' for now), the financial risk is not made complicated by the existence of scholarships or 'loans' that don't actually need to be repaid in the event of insufficient salary, and the test fee itself covers only the test and not any education that goes with it (well you're more likely to pass if you invest in some lessons too, but that's separate). Paying to get your app considered by the App Store could be likened to paying for a driving test: in both cases you might fail, in which case the only thing you gain from your test fee is the rejection letter. But at least with a driving test you can learn what went wrong and take it again. This is not always possible with the App Store. Yes they might point out a small problem that can be corrected for re-submission, but if they reject the very principle of your app, then that would be like failing a driving test on medical grounds. How many people pay for a driving test when they're likely to fail on medical grounds?

The other issue is the amount that's charged. If you're running a competition and you want individuals to enter, you had better not set the entry fee so high that they're put off from trying. In the case of the App Store, the entry fee is 100 pounds for the developer account plus the cost of any new equipment that's needed (although this is partially offset by the fact that you still get to keep some decent hardware in the event of failure), and the prize is simply that of being allowed to help users by publishing your app on the Store. (Yes if you have a wildly-popular app and you are willing to 'monetize' it then there might be a financial prize, but in most cases this is unlikely and I'd rather focus on 'helping users'.) I believe they have set this figure incorrectly and I'd rather tell my users to switch to Android.

If you're running a competition and you want individuals to enter, you had better not set the entry fee so high that they're put off from trying

Perhaps Apple would do better in business if they broke the fee into two parts: a low 'consideration' fee and a higher 'publication' fee. The first fee pays for the time of somebody to look at your app and say 'reject', 'accept if you pay the full fee', or perhaps in especially good cases 'accept without further fee' (after all Android manage to charge only 25% of Apple's fee and waive renewal payments, so I expect Apple can afford to take the best apps on their own merits).

Renewals

Hosting an app on the App Store comes with a rather high annual hosting fee. One year alone of this hosting fee is four times what Google Play charges for life (and that's not even counting the fact that you must keep up with Mac hardware that is supported by current versions of macOS in order to run acceptable versions of Xcode to submit to the Store; with Android you have a much broader choice of developer equipment). This is not as off-putting as the fact that you might pay all this and still be rejected, but it's still rather unpleasant if all you're trying to do is help the users.

Developer pooling?

I tried to find existing Apple developers to publish my app. After all, if they've already paid up, it shouldn't cost them extra to piggy-back my app onto their account as well, and I don't really care whose name appears on it as long as it gets out there. But the reactions I got were generally along the lines of 'I don't want to risk my standing with Apple by submitting code I didn't write', or 'if you say you want it to be a free app with no ads, I don't want to bother because I only want to monetise', or 'the latest Xcode says you're using a deprecated API, so I don't want to risk trying to submit.' Are Apple developers not even allowed to make a test release with a deprecated API call? I might want to see if the general idea of the

SILAS S. BROWN

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

if anyone out there is thinking of learning some iOS development, I'm sorry to say my current recommendation is 'don't bother', not unless you are absolutely certain of getting paid for it

app will be accepted by the Store editors, and to see how many users it gets, before deciding to take the trouble rewriting it to suit the ever-changing whims of Apple's API designers, especially if that means investing in new developer equipment. What's wrong with submitting anyway and seeing if we can get a 'yes in principle' first?

Is there an ACCU member out there who happens to have a paid-up App Store account and might be willing to compile and submit apps from other ACCU members? Bear in mind this means taking final responsibility for the code: although Apple allows you to use other people's code in your app (think 'libraries') if you have a licence to do so from the original developer, it's still considered your fault if the library does something obnoxious like renders the end-user's phone unusable, so if you're going to run such a service you had better be good at proof-reading source code, and it's understandable to be trigger-happy on the 'reject' button if it looks a bit like 'underhanded C', although the mere use of deprecated APIs is not in itself underhanded, it's just something that would need fixing in the event of the apps being accepted and getting enough users to make further development worthwhile.

Losses

I don't expect Apple to read this, but in case any other readers are thinking of setting up a hosting business like the App Store, it might be worth considering how much money Apple might be losing from their less-than-ideal treatment of app developers.

Naturally I recommend users choose Android over Apple if they want to use my software. I did once get a sharpish reaction along the lines of 'you shouldn't be telling people which phone to choose, it's YOUR fault if it doesn't run on Apple, YOU fix it', but that tends to be the exception. Few people would immediately rush out to change their phone just to use my program (it's not THAT good), but I have heard it has entered into people's Android-versus-Apple decisions when they are going to buy a new phone anyway, and I have heard of people not switching from Android to Apple because doing so would lose my app.

I now have about 4,000 active users on Android (having lost about 1,000 over 6 months when a large website started providing a similar-but-not-as-complete service); I don't know how many of my users have thought of switching to Apple, but in the worst case (from Apple's perspective) I could be holding them back from over a million dollars worth of phone sales just because Apple wouldn't let me try their submission process without paying a thousand up-front. Multiply that by a few other nice app developers and you can see Apple might be missing out an serious money because of their decision to kill off the 'long tail' (the area under the graph contributed by small less-popular apps that might not amount to much individually but add up to more than you think when they're all taken together).

I don't know about running businesses: I'm just a developer; don't expect me to make good business decisions; for all I know, there might be a good financial reason for Apple to take those losses. But there might not. Either way I feel it should be pointed out.

And if anyone out there is thinking of learning some iOS development, I'm sorry to say my current recommendation is 'don't bother', not unless you are absolutely certain of getting paid for it (as in you have already secured employment at a company that now wants you to learn it); the costs are too high otherwise. This does not bode well for the long-term future of the platform. ■

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

The Refactor Part of TDD – Reducing to the Min with FizzBuzz

Simon Sebright considers whether de-duplication in refactoring can be too aggressive.

I recently had a little time on my hands, at the same time a clearish head and a laptop with a development environment on it, so decided to keep my eye in with a short FizzBuzz TDD kata session.

The design part of the exercise was minimal (one function), and although I added a few more tests during the session, my main focus was on the Refactor (of Red, Green, Refactor) part – seeing what happens if I try to ruthlessly reduce things to the minimum. This means removing duplication and potentially superfluous concepts/complications.

The result, as can be seen on GitHub (a .Net Core Assembly project in Visual Studio 2019 [1]), raised a few questions in my mind as to how far one can/should go down that track. See for yourself, what you think about readability, understandability, maintainability, extensibility, etc., of the different versions of the code...

Getting to the FizzBuzz interface

First, I briefly describe how I ended up with the interface to the core logic.

With the first test:

```
[TestMethod]
public void Start()
{
  Assert.AreEqual(1, FizzBuzzer.Eval(1));
}
```

I generated a static method in my `FizzBuzz` class using IDE refactoring. It was red first (throws exception). Then I made it green:

```
static class FizzBuzzer
{
  internal static object Eval(int v)
  {
    return 1;
  }
}
```

Next came more tests (the Red of Red, Green, Refactor) and I went through a couple of versions:

```
internal static object Eval(int v)
{
  return v == 3 ? (object)"Fizz" : v;
}

...
```

Before arriving at the final interface (returns a string now) and a working implementation:

```
internal static string Eval(int v)
{
  return v%15 == 0 ? "FizzBuzz":
      v%3 == 0 ? "Fizz" :
        v%5 == 0 ? "Buzz" :
          v.ToString();
}
```

So, as we can see I take an integer being the number currently being 'played' in a game and return as a string the correct answer, which a player should say aloud.

Listing 1

```
[TestClass]
public class FizzBuzzTest
{
  [TestMethod]
  public void Numbers()
  {
    Assert.AreEqual("1", FizzBuzzer.Eval(1));
    Assert.AreEqual("2", FizzBuzzer.Eval(2));
    Assert.AreEqual("4", FizzBuzzer.Eval(4));
    Assert.AreEqual("7", FizzBuzzer.Eval(7));
  }
  [TestMethod]
  public void Singles()
  {
    Assert.AreEqual("Fizz", FizzBuzzer.Eval(3));
    Assert.AreEqual("Buzz", FizzBuzzer.Eval(5));
  }
  [TestMethod]
  public void Multiples()
  {
    Assert.AreEqual("Fizz", FizzBuzzer.Eval(6));
    Assert.AreEqual("Buzz", FizzBuzzer.Eval(10));
  }
  [TestMethod]
  public void Combined()
  {
    Assert.AreEqual("FizzBuzz",
      FizzBuzzer.Eval(15));
    Assert.AreEqual("FizzBuzz",
      FizzBuzzer.Eval(30));
  }
}
```

At this point, I could have stopped after making sure I had a suitably full set of tests (see Listing 1 for the tests). Also, I could have considered what happens with 0 or negative input. I decided to leave those (as they actually produce something sensible) and look at the refactoring opportunities.

Removing duplication and complications

There are some things being duplicated in the code:

- The use of the ternary operator, three times
- 3 and 5 are duplicated in 15 (as prime factors)
- The string literals `"Fizz"` and `"Buzz"` are to be found in the compound `"FizzBuzz"`

The first thing I did was to address the 15 and the `"FizzBuzz"`. I could do this in one go by concatenating the results of the two factors (see Listing 2).

SIMON SEBRIGHT

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

Listing 2
```
internal static string Eval(int v)
{
  string fizzness = v % 3 == 0 ? "Fizz" :
    String.Empty;
  string buzzness = v % 5 == 0 ? "Buzz" :
    String.Empty;
  string fizzBuzzness = fizzness + buzzness;
  return String.IsNullOrEmpty(fizzBuzzness) ?
    v.ToString() : fizzBuzzness;
}
```

Listing 3
```
internal static string Eval(int v)
{
  string Test(int test, int divisor,
    string whatIfDivisible)
  {
    return test % divisor == 0 ? whatIfDivisible :
      String.Empty;
  }
  string fizzness = Test(n, 3, "Fizz");
  string buzzness = Test(n, 5, "Buzz");
  string fizzBuzzness = fizzness + buzzness;
  return String.IsNullOrEmpty(fizzBuzzness) ?
    n.ToString() : fizzBuzzness;
}
```

It is perhaps clearer that the result is compounded from the two factors, the name **`fizzBuzzness`** intending to convey this concatenation (as opposed to declaring a result string and adding both to it in turn).

Note that we still have duplication of the ternary operator. We can get rid of that by implementing a local function to take care of that logic (see Listing 3).

But, we now have duplication of calling the **`Test()`** function twice. We can remove that using a loop (Listing 4).

This also has the effect of putting the 'data' for the program in one place. So rather than having hard-coded values in multiple lines, it's all in one bundle. I chose to use an anonymous class to hold the divisor data, to save introducing a new named class which would only be used in this context.

Notice that the **`Eval()`** function is now getting longer in terms of lines of code, although we have removed some duplication. That is not a bad thing.

We can now remove the loop by observing that there is a built-in function which will go through a collection in order and perform an action on

Listing 4
```
internal static string Eval(int n)
{
  string Test(int test, int divisor,
    string whatIfDivisible)
  {
    return test % divisor == 0 ? whatIfDivisible :
      String.Empty;
  }

  var factors = new[]
  { new { divisor = 3, term = "Fizz" },
    new { divisor = 5, term = "Buzz" } };
  string factorterms = string.Empty;

  foreach (var factor in factors)
  {
    factorterms += Test(n, factor.divisor,
      factor.term);
  }

  return String.IsNullOrEmpty(factorterms) ?
    n.ToString() : factorterms;
}
```

Listing 5
```
internal static string Eval(int n)
{
  string Test(int test, int divisor,
    string whatIfDivisible)
  {
    return test % divisor == 0 ? whatIfDivisible
:
      String.Empty;
  }

  var factors = new[]
  { new { divisor = 3, term = "Fizz" },
    new { divisor = 5, term = "Buzz" } };
  string factorterms =
    factors.Aggregate(String.Empty, (a, b) =>
    { return a + Test(n, b.divisor, b.term); });
  return String.IsNullOrEmpty(factorterms) ?
    n.ToString() : factorterms;
}
```

successive elements, accumulating a result. For example, with a sequence of integers, a 'seed' value of 0, a function adding each successive element to the current sum will give you the sum of all the elements. Or a seed of 1 can be used when multiplying to give the result of multiplying all the numbers in the collection together (lowest common denominator if your numbers are prime). Taking a starting seed value of an empty string, we can simply concatenate the results of each factor (see Listing 5).

The **`Aggregate()`** function is our friend here in .Net. Ruby has **`accumulate()`** and I am sure other languages have their equivalent.

The next step could be to take the contents of the **`Test()`** function and pass this directly as a lambda to **`Aggregate()`** (Listing 6).

Listing 6
```
internal static string Eval(int n)
{
  var factors = new[]
  { new { divisor = 3, term = "Fizz" },
    new { divisor = 5, term = "Buzz" } };
  string factorterms =
    factors.Aggregate(String.Empty, (a, b) => {
      return a + (n % b.divisor == 0 ? b.term :
      string.Empty); });
  return String.IsNullOrEmpty(factorterms) ?
    n.ToString() : factorterms;
}
```

I am still slightly annoyed that in the last line we have to reference **`factorTerms`** twice. In a language where an empty string evaluates to the boolean value false, we might be able to get around that. Another go might be to find or write a function which is specifically designed to take the first non-empty string argument from a list. This makes the **`Eval()`** function slightly simpler, but we would need tests for the new function. Listing 9 (overleaf) gives an example. I chose to pass objects to it, thus delaying the need to turn the integer n into a string in the **`Eval()`** function. Listing 7 is the result of that process.

Listing 7
```
internal static string Eval(int n)
{
  var factors = new[]
  { new { divisor = 3, term = "Fizz" },
    new { divisor = 5, term = "Buzz" } };
  string factorterms =
    factors.Aggregate(String.Empty, (a, b) => {
      return a + (n % b.divisor == 0 ? b.term :
      string.Empty); });
  return FirstNonEmpty(new object[] {
    factorterms, n });
}
```

Python has setdefault

Silas S. Brown shares a quick tip on Python.

I learned Python in the days when Python 1.x was still around, incidentally as a result of reviewing *Professional Linux Programming* by Neil Matthew and Richard Stones, in *CVu* 13(2), April 2001. Python 2.0 had been released in October 2000 but it took a while for it to be fully integrated into GNU/Linux distributions etc. I did start using Python 2 whenever possible (not least because of its Unicode support) and made 2.0 my code's minimum requirement, but I only recently discovered in 2019 that all these years I've been using a certain Python 1.x idiom that 2.x has a single instruction for.

So, public service announcement (in case anybody else has been labouring under the same incomplete understanding all these years), if you often write code like this:

```
if not myDict.has_key(k): myDict[k] = []
myDict[k].append(s)
```

then you are still in the 1.x days on two counts: use of `has_key` (nowadays you can simply say `if not k in myDict`), and non-use of `setdefault`. In every version of Python from 2.0 onwards, the above is equivalent to:

```
myDict.setdefault(k,[]).append(s)
```

and the only reason I can think of not to use this is if the constructor to the `[]` were so much overhead that you really don't want to run it unless needed (which isn't the case for the empty list, but might be the case for some user-defined type). Otherwise, `setdefault` away. ■

SILAS S. BROWN

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

The Refactor Part of TDD – Reducing to the Min with FizzBuzz (continued)

Hopefully, their names indicate what the function is doing, although I rather think `Aggregate()` is too generic – you need to look at the seed and function provided to work out what it is actually going to do. Knowing that .Net has the `String.Concat()` function, I broke the `Aggregate()` into two steps. Hopefully the naming of the intermediate variable makes it clear what the `Select()` is doing (picking out the strings of the divisible factors). Listing 8 could be the result.

Is smaller better, what is smaller?

So now we have a function of 'only' 4 lines (depending on your preferred line length, we could also inline the variable `factorTerms`), and practically no logic in the implementation. The only real logic is the divisibility test, which is, after all, the core of FizzBuzz. Instead, we rely on other functions to do the hard work for us, combining them in a crafty way.

Now, the interesting questions come. Imagine you are involved in this codebase. You are in a team with people both less and more experienced than you. Your team is responsible for this code, and possible changes to the game, or new but similar games.

Looking back through the various incantations of `Eval()`, which do you feel more comfortable with, given the following scenarios:

- It is just you maintaining this code
- You have juniors on your team
- You are all senior on the team
- Your product owner decides to change to 5 and 7 as prime factors
- You update the game to include any number which contains the factors as strings (e.g. 13, 23, 31 etc. for 3 and 51, 52, 53, etc.) for 5
- You need to make a new game FizzBuzzBazz with factors 3, 5 and 7

Conclusion

I certainly found the exercise interesting and illuminating, seeing how naive or 'clever' one can be with the same piece of functionality. In a normal day job, such questions will be constantly arising when working on code. Now and then it pays to think about your target audience of (you and) your co-developers. ■

Reference

[1] FizzBuzz: https://github.com/SimonSebright/FizzBuzz

Listing 8

```
internal static string Eval(int n)
{
  var factors = new[]
  { new { divisor = 3, term = "Fizz" },
    new { divisor = 5, term = "Buzz" } };
  IEnumerable<string> relevantFactorTerms =
    factors.Select((factor) => {
      return (n % factor.divisor ==
      0 ? factor.term : string.Empty); });
  string factorterms =
    String.Concat(relevantFactorTerms);
  return FirstNonEmpty(new object[] {
    factorterms, n });
}
```

Listing 9

```
internal static string FirstNonEmpty(object[]
  values)
{
  foreach (object o in values)
  {
    string s = o.ToString();
    if (!String.IsNullOrEmpty(s))
    {
      return s;
    }
  }

  throw new ArgumentException("FirstNonEmpty
  given collection with no empty strings");
}
```

Static C library and GNU Make

Ian Bruntlett shares his experiences with using 'make' to build a small test project.

My biggest problem with the GNU Make manual [1] was its lack of testable examples. This article is intended to partially redress that, by offering a very simple but expandable example. For simplicity, I am not using a TDD framework like check [2] or CppUTest [3] – that is likely to come later. The code is now available on my GitHub page [4].

My Makefile – the main bits

This is what happens when **make clean** is invoked:

```
$ make clean
rm -fv *.o libtrim.a test_ltrim test_rtrim
removed 'ltrim.o'
removed 'rtrim.o'
removed 'libtrim.a'
removed 'test_ltrim'
removed 'test_rtrim'
```

This is what happens when **make all** is invoked, after a **make clean**:

```
$ make all
gcc -g   -c -o ltrim.o ltrim.c
ar rvU libtrim.a ltrim.o
ar: creating libtrim.a
a - ltrim.o
gcc -g -L.    test_ltrim.c -ltrim -otest_ltrim
gcc -g   -c -o rtrim.o rtrim.c
ar rvU libtrim.a rtrim.o
a - rtrim.o
gcc -g -L.    test_rtrim.c -ltrim -otest_rtrim
```

Show me the code

First off, a header file to list what will be in our static C library.

```
// trim.h
// (c) Ian Bruntlett, October 2019
// trim left-hand side text from a string,
// in situ
extern void ltrim(char s[]);

// trim left-hand side text from a string, into
// a user-provided buffer
extern void ltrimcpy(char *dest, char *src);

// trim right-hand side text from a string,
// in situ
extern void rtrim(char text[]);

// trim right-hand side text from a string,
// into a user-provided buffer
extern void rtrimcpy(char *dest, char *src);
```

This library is tiny – it is only made up of two files – `ltrim.c` and `rtrim.c` but is OK as an example. Typically, many more object files would be part of the library. Listing 1 is one of them.

IAN BRUNTLETT

On and off, Ian has been programming for some years. He is a volunteer system administrator (among other things) for a mental health charity called Contact (www.contactmorpeth.org.uk). He is learning low-level and other, higher-level, aspects of programming.

Listing 1

```
// ltrim.c
// (c) Ian Bruntlett, October 2019
#include <string.h>
#include <ctype.h>
#include "trim.h"

void ltrim(char s[])
{
  if ( !isspace(s[0]) )
    {return;}
  // Get index of first non-space
  int non_space_index=0;
  while ( isspace(s[non_space_index]) )
    { ++non_space_index; }
  // copy text characters over the white space
  int dest_index=0;
  while ( s[dest_index++] =
    s[non_space_index++] )
  ;
}
void ltrimcpy(char *dest, char *src)
{
  strcpy(dest, src);
  ltrim(dest);
}
```

The functions in Listing 1 are used to trim white space from the left-hand side of a C string. I think they work as they have passed various tests (more on that later). The companion source file, `rtrim.c` (Listing 2) trims white space from the right-hand side of a C string.

Listing 2

```
// rtrim.c
// (c) Ian Bruntlett, October 2019
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "trim.h"
#define TEST_MAX (100)

void rtrim(char text[])
{
  int length = strlen(text);
  if (!length)
    return;
  for (int rhs = length-1;
       isspace(text[rhs]);
       --rhs
       )
    text[rhs] = '\0';
}
void rtrimcpy(char *dest, char *src)
{
  strcpy(dest,src);
  rtrim(dest);
}
```

Listing 3

```
// test_ltrim.c
// (c) Ian Bruntlett, October 2019
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "trim.h"

#define TEST_MAX (100)

void test_ltrimcpy(char *test_argument,
  char *expected_result)
{
  char test_result[TEST_MAX];
  assert(strlen(test_argument) < TEST_MAX);
  ltrimcpy(test_result, test_argument);
  int success = !strcmp(test_result,
    expected_result);
  if ( success )
    { printf("OK :" );  }
  else
    { printf("Failed :" );  }
  printf("argument : '%s' result : '%s'
    : expected '%s'\n", test_argument,
    test_result, expected_result);
  assert(success);
}
int main(int argc, char *argv[])
{
  if (argc==1 )
  {
    test_ltrimcpy("  Hello ", "Hello ");
    test_ltrimcpy("Hello ", "Hello ");
    test_ltrimcpy("","");
    test_ltrimcpy("     ","");
  }
  else if (argc==3)
    test_ltrimcpy(argv[1],argv[2]);
  else
    printf("%s incorrect arguments\n", argv[0] );
  return 0;
}
```

So that is some code to test. I have two test programs, `test_ltrim.c` and `test_rtrim.c`, that I have used as test beds for the above code. Listing 3 is `test_ltrim.c`. If run with no parameters, it runs a built-in set of tests. Otherwise it uses two parameters from the command line (usage: **`test_ltrim "test string" "expected result string"`**) for convenience.

So that is the code. How do we build and run it? It is possible (but error-prone) to continually type in commands to build the code prior to running it. Step forward GNU Make. It automates the build process and typically needs to be provided with a Makefile so it knows what you want it to do. Listing 4 is my Makefile.

This makefile can build the test executables (**`test_ltrim`** and **`test_rtrim`**), the library modules (`ltrim.o` and `rtrim.o`), and the library itself (`libtrim.a`), when the command **`make`** is run. That runs the default goal which, in this case, is called **`all`** and depends upon **`test_ltrim`** and **`test_rtrim`**. Make then looks at those targets and determines that as well as those targets, both `ltrim.o` and `rtrim.o` are required and are to be stored in the static C library `libtrim.a`.

Makefiles

To quote the manual, Makefiles contain five kinds of things:

1. **Explicit rules**, like those for **`test_ltrim`** and **`test_rtrim`**.
2. **Implicit rules** that define how a source file with a particular file extension can be processed to build a destination file with another file extension. Like telling the compiler how to build object files from C source files. They tend to be customisable by setting variables such as **`CFLAGS`** and **`ARFLAGS`**. To avoid conflicts between Make's built-in variables and user-defined variables, the user-defined variables are usually given lower-case names.
3. **Variable definitions** (there are two types of variables). Typically used to specify pieces of text that are likely to be repeated. For example, in my makefile there is a line that defines the variable executables to contain the text **`"test_ltrim test_rtrim"`**.
4. **Directives** for more advanced Makefiles. See the manual [1].
5. **Comments** – lines starting with **`#`**.

Listing 4

```
# makefile for libtrim.a static library, using
# archive members as targets.
# (c) Ian Bruntlett, October 2019

CC = gcc
CFLAGS = -g
ARFLAGS = rvU
executables =  test_ltrim test_rtrim
tar_filename = trim-source-$(shell date "+%Y-%m-
%d" ).tar

all: $(executables)

test_ltrim : test_ltrim.c trim.h
libtrim.a(ltrim.o)
  $(CC) -g -L.    test_ltrim.c -ltrim -o$@

test_rtrim : test_rtrim.c trim.h
libtrim.a(rtrim.o)
  $(CC) -g -L.    test_rtrim.c -ltrim -o$@

ltrim.o : ltrim.c trim.h

rtrim.o : rtrim.c trim.h

.PHONY: tar-it
tar-it:
  tar -cvf $(tar_filename) *.c *.h makefile

.PHONY: clean
clean:
  rm -fv *.o libtrim.a $(executables)
```

Rules

Rules tell Make how to behave. An explicit rule looks similar to this (note that some things can be left out):

```
targets : prerequisites
  commands that must be indented with a TAB
  character
all: $(executables)
```

The above code is an explicit rule that depends on the value of **`$(executables)`** which we know contains the two values **`test_ltrim`** and **`test_rtrim`**. So, when **`make all`** is invoked, it is akin to running **`make test_ltrim test_rtrim`**.

```
test_ltrim : test_ltrim.c trim.h
libtrim.a(ltrim.o)
  $(CC) -g -L.    test_ltrim.c -ltrim -o$@
```

The above rule has the target **`test_ltrim`**. Once this rule is processed, the file `test_ltrim` should have been created (it is an error if it isn't). It lists the prerequisites – these are files that the target (**`test_ltrim`**) depends upon. If they are changed, the target needs to be recompiled. This rule is special in that it has the prerequisite **`libtrim.a(ltrim.o)`**. Normally it would be l**`trim.o`**. However, this alternative syntax tells **`make`** that the target depends on `ltrim.o` which, in turn, is to be part of

an automatically generated static C library called `libtrim.a`. In Linux, library files are managed using the **`ar`** utility. To get the library created and updated properly, a built-in variable had to be set like this:

```
ARFLAGS = rvU
```

The command line to build **`test_ltrim`** is this:

```
$(CC) -g -L.    test_ltrim.c -ltrim -o$@
```

Where:

- **`$(CC)`** indicates that the default C compiler is to be used. In this case, gcc.
- **`-g`** specifies that debugging information is to be included
- **`-L.`** specifies that the current directory is also going to be searched for libraries
- **`test_ltrim.c`** specifies the source filename
- **`-ltrim`** specifies that the shared library `libtrim.a` must be consulted

`-o$@` is interesting. It states that the output executable name is the same name as the target – i.e. **`test_ltrim`**. **`$@`** is one of many special variables that are invaluable.

The rules for determining if `ltrim.o` and `rtrim.o` need to be recompiled are this:

```
ltrim.o : ltrim.c trim.h
rtrim.o : rtrim.c trim.h
```

They specify that the above object files depend on their namesake C source files as well as the header, `trim.h`. If I had used a default rule for this, the header file would not have been considered as a prerequisite.

Helper scripts (aka Phony Targets): clean and tar-it

For convenience, it is possible to put commonly used shell commands into the Makefile which can in turn be invoked when required. In my Makefile, **`make clean`** deletes all the binary files.

```
.PHONY: clean
clean:
  rm -fv *.o libtrim.a $(executables)
```

This also shows that, when maintaining Makefiles, you need to be conversant with Make's syntax and your shell's syntax. This can be confusing.

Early on in my makefile, this variable was defined by me.

```
tar_filename = trim-source-$(shell date
  "+%Y-%m-%d" ).tar
```

The above line is defines a variable, **`$(tar_filename)`**, which is set to some literal text and gets some of its value from running the date command. For example, if the above code was run today, **`$(tar_filename)`** would be set to the value `trim-source-2019-11-03.tar`. This variable is used later on in the makefile, to be used when **`make tar-it`** is invoked:

```
.PHONY: tar-it
tar-it:
  tar -cvf $(tar_filename) *.c *.h makefile
```

Git and Git-Hub (from my notes)

I confess, a long time ago, I wrote a very basic (but functional, nevertheless) source code sharing system called SCHOLAR (Source Code Held Online Archiving and Retrieving). Apparently this makes it harder for me to get to grips with git and GitHub (a site for hosting git repositories). This is not a tutorial. Fortunately for me, I had help from accu-general to get me going and, most recently, had help directly from Bronek Kozicki.

First I signed up for a (free) GitHub account (https://github.com). Then I started trying to do things. The next thing I should have done is use the GitHub webpage to create a new repository – called 'trim' for my code. I have (eventually) decided to go public on my email account and so I also went to https://github.com/settings/emails and deselected Block command line pushes that expose my email. I do hope that I have done the right thing. Another thing I should have done was these commands (your parameters will need your name and your email address):

```
git config --global user.email
  "ian.bruntlett@gmail.com"
git config --global user.name "Ian Bruntlett"
```

Before I could put files on GitHub, I needed a local repository. So I went to the directory I had my code in and typed:

```
git init
git status
git add *.c
git add *.h
git add makefile
git add readme.txt
git commit -m "First commit for libtrim support
code"
git remote add origin https://github.com/ian-
bruntlett/trim.git
git remote -v
```

Output of the above command:

```
origin  https://github.com/ian-bruntlett/trim.git
(fetch)
origin  https://github.com/ian-bruntlett/trim.git
(push)
git push -u origin master
```

Don't quote me on git! Now that I have an idea of which commands are most important to me, I will be reading the man pages in depth and reading a good book about it [5].

Conclusion

I hope this has been a helpful introduction to Make and static libraries. Happy coding! ■

References

[1] GNU Make manual https://www.gnu.org/software/make/manual/

[2] Check – see https://libcheck.github.io/check/ (or install the check package)

[3] CppUTest – see https://cpputest.github.io/ (or install the cpputest package)

[4] https://github.com/ian-bruntlett/trim

[5] *Version control with Git* by Jon Loeliger & Matthew McCullough.

How to Stay Out of a Webmaster's Bad Books

Silas S. Brown demonstrates how not all online resources are created equal.

I've made a Chinese-English 'dictionary supplement' of about 40,000 names and other things that aren't normally defined in dictionaries but are nevertheless useful to be recognised by text analysis tools.

Being public-spirited, I thought it was a good idea to put the resulting collection on my home page as a downloadable text file that you can import into the software of your choice.

Until one month this file got over 70 gigabytes of traffic from China, which the Apache server logs show was sent by about 1,600 machines in various provinces, downloading the whole file an average of 30 times per machine. The worst offender was a single machine downloading the file 40 times per day.

The logs showed all of the offending machines were claiming to be Chrome 51 (a 3-year-old browser), all the timestamps were during the day in China time, and in some cases the download was stopped part-way through the file. So my guess is somebody tried to write a search tool, used a fetch library claiming to be Chrome 51, and set it to read from my server until the search string is found or until end of file. How would we critique such code?

Rule 1: Cache locally if possible

You wrote a search tool? Great! Now, do you think it's possible somebody might want to use it more than once? After all, how often have you searched for a word, found the results not to your satisfaction, and thought of another word you could try instead? Might your users behave the same way? If you're downloading all 3 million bytes of my text file for every single search, that could multiply up pretty fast. Obviously you should try to store that file locally if you can, and check for updates only occasionally. (And when you do check for updates, there's an HTTP header called `If-Modified-Since` you can use to avoid a download altogether in the case of no change. The real Chrome 51 does this, which is part of the reason why I don't think your tool is right to pretend to be Chrome 51. But if it's too difficult to send an If-Modified-Since header, I won't mind an unconditional download as long as its frequency is low enough, say once a week at the most.)

too many people out there seem to think of websites as being like motorways

But let's say you are a beginner at coding and you don't know how to store a file locally. (But you're still going to have 1600 active users across China.) Then what? Well how about this idea: put the file on your *own* server! Or if you don't have your own server, find a service that will host it for you, but *you* must be the responsible person who owns the account on that service, not me. When I say my file is free (liberally licensed or public domain), I mean please copy it, and I say 'copy' precisely because I do *not* want all your traffic to come back to my original server. Yes I would appreciate people being up-to-date with the original (I don't really like seeing years-old outdated copies of my stuff floating around), but I shan't mind a reasonable delay before your mirror updates itself, and I'd much prefer that to having to foot the bill for your excessive traffic. Free is free copying-wise, but if your app might be big, you should at least do the right thing and help with the back-end infrastructure. Fair enough?

Rule 2: Identify yourself

When you fetch a resource from a Web server, you send it a `"User-Agent"` string to identify your browser. Some servers have a list of 'acceptable' browsers and block everything else, which annoys me, especially when by doing this they inadvertently block tools used by blind people. Consequently, some downloading tools are in the habit of choosing a string from an 'acceptable' browser (for example, Chrome 51 from 2016) and pretending to be that. But this is bad practice, because it prevents webmasters from finding you if there is a problem. So I suggest:

1. Try setting your User Agent to the actual name of your app, preferably with a URL, or at least something I can type into Google to find you. That way, if it starts misbehaving, the webmaster can actually start talking with you and working out a mutually acceptable solution, instead of just coming up with ways to block your app. True, some webmasters might say "I don't have time for this" and block you anyway, but it's surprising how many of us are in fact nice enough to try contacting you first if it's obvious how to do so.
2. *Only if* (1) turns out not to work because a misguided webmaster has set a list of 'acceptable' browsers, should you *then* start thinking about claiming to be an 'acceptable' browser to get past that misguided test. Even then, it's still a good idea to try identifying your app as an 'extension' to the acceptable browser, i.e. try sending the Chrome string with '(actually XYZ app)' added to the end.

Sending the Chrome string with nothing else added should be an absolute last resort. It may force the server administrator to block that particular version of Chrome (as I did) when they can't find out how to open a dialogue with you.

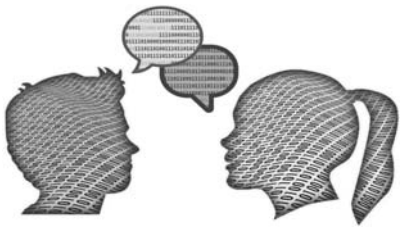
But let's suppose you're such a beginner that you can't figure out how to change the User-Agent string on your library, nor can you figure out how to change to a better library (I don't know which library it is that defaults to pretending to be Chrome 51, but I don't expect it to be a good one). Now what? How about this: See if the webmaster has published an email address (as I have), and send a courtesy email saying, "Hi, I wrote this app that downloads stuff from your server, what do you think?" They might be able to help you make the app better. At the very least, they'll know they can contact you if something goes wrong, so you might be able to have a conversation instead of being blocked.

Unfortunately, it seems there are still too many people out there who seem to think of websites as being like motorways. You can send a whole bunch of cars down a motorway without having to think about 'will the presence of these cars cause unacceptable congestion', unless it's an enterprise fleet so big that you already know how to manage the problems. Coders need to realise that some of our servers are more like small private roads with a guard at the gatehouse keeping an eye on the situation, who lets individuals pass but might start putting up barriers if you bring in a tour group without talking about it first. I did set the block message to something that makes sense if a human reads it, but I don't know if the code's author will, and I doubt I'll ever know the name of their program, which is a pity because I would have reached out to them and helped them fix it if it had identified itself. ■

SILAS S. BROWN

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

Code Critique Competition 121



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 doing some simple text processing to extract sentences from a text document and produce each one on a separate line. I'm getting a space at the start of the lines and wonder what's the best way to change the regexes to remove it?"

```
$ cat input.txt
This is the first sentence. And the
second. This is
the
last
one.

$ sentences input.txt
This is the first sentence.
 And the second.
 This is the last one.
```

Listing 1 contains the code. As always, there are a number of other things you might want to draw to the author's attention as well as the presenting problem.

Listing 1

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <regex>
using namespace std;

int main(int argc, char** argv)
{
  std::ifstream ifs(argv[1]);
  std::stringstream ss;
  char ch;
  while (ifs >> std::noskipws >> ch)
    ss << ch;

  string s = ss.str();

  smatch m;

  while (regex_search(s, m,
    regex("[\\s\\S]*?\\.")))
  {
    std::string sentence(m[0]);
    std::regex whitepace("[\\s]+");
    std::regex_replace(
      std::ostream_iterator<char>(std::cout),
      sentence.begin(), sentence.end(),
      whitepace, " ");
    std::cout << std::endl;
    s = m.suffix().str();
  }
}
```

Critiques

Pete Cordell <accu@codalogic.com>

Being one of the few people in the world who actually likes regular expressions, I thought I'd have a play with this Code Critique.

In the order that thoughts arose in the file, I noticed that the code used the common `using namespace std;` line but then prefixed many functions and classes with `std::` anyway. I'd prefer the code to either rely on the directive or not, and it should commit to one way or the other.

I opted for not using the directive. That said, these days I feel that in a modern language strings should be treated as first class types similar to `bool`, `int` and `float`. I therefore changed `using namespace std;` to `using std::string;`.

This does make the primary loop where the majority of the regex work is done somewhat noisy. As much by way of experiment I added statements such as `using std::smatch;` at the beginning of the function to see how it looked. I actually quite like this as it gives the reader some foresight that these are the sorts of toys this function is going to be dealing with.

Going back to the function declaration for `main()`, g++ 9.2.0 on Wandbox complained that `argc` wasn't used. The simple way to fix this is to do use `int main(int, char** argv)`.

The reading of the input into a string can actually be done as a one liner:

```
string s((std::istreambuf_iterator<char>(ifs)),
  std::istreambuf_iterator<char>())
```

This looks like some weird incantation to me but I guess there are only so many ways that an input stream can be used to initialise a string, so a reader maybe able to guess what was going on even if they weren't sure.

The core `regex` in the example is `"[\\s\\S]*?\\."`. This relies on the non-default non-greedy matching specification `"*?"`. To me, this is an advanced `regex` feature and best avoided if possible. (If nothing else, the syntax looks scary.) The desired goal of the `regex` is to match anything up to and including the first encountered full-stop. I think this is better expressed using a negative character class match such as `"[^.]*\\."`.

The `"[^.]*\\."` expression also captures leading whitespace. My simple solution to this is to change the expression to `"\\s*([^.]*\\.)"`. This skips over the leading whitespace and creates another capture group for what we want. So where in the loop we previously had `m[0]` for what the whole regular expression captured, I changed this to `m[1]`.

The `"[\\s]+"` `regex` for whitespace is over complicated and can simply be done as `"\\s+"`.

The last line of the original loop was `s = m.suffix().str();`. Creating a new string for the subsequent iteration feels like the wrong way to do things. My solution to this was to set up a `string` iterator of the

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

form `string::const_iterator is( s.cbegin() )` and then use that in the loop condition as `regex_search(is, s.cend(), m, ...`. At the end of the loopb the iterator is updated using `is += m.length()`. Whether this is the idiomatic way to do this I don't know. I look forward to finding out. [Editor: you could use `is = m.suffix().first;`]

My modified code is:

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <regex>
using std::string;
int main(int , char** argv)
{
  using std::smatch;
  using std::regex_search;
  using std::regex;
  using std::regex_replace;
  std::ifstream ifs( argv[1] );
  string s((std::istreambuf_iterator<char>(ifs)),
    std::istreambuf_iterator<char>());
  smatch m;
  string::const_iterator is( s.cbegin() );

  while (regex_search(is, s.cend(), m,
    regex("\\s*([^.]*\\.)")))
  {
    string sentence(m[1]);
    regex whitepace("\\s+");
    regex_replace(
      std::ostream_iterator<char>(std::cout),
      sentence.begin(), sentence.end(),
      whitepace, " ");
    std::cout << std::endl;
    is += m.length();
  }
}
```

James Holland <James.Holland@babcockinternational.com>

In order to make corrections, it is necessary to understand how the existing program works. Regular expressions can be a little difficult to fathom and so I will need to take things one step at a time. The first statement of interest is the `regex` parameter within the control section of the `while` loop. Enclosed within the square brackets are two shortcut characters. The first is `\\s` and denotes a space character. The second is `\\S` and denotes any character that is not a space. Either of these shortcuts are accepted as a match because they are enclosed in square brackets. This has the effect of matching any character including spaces.

The next `regex` character is `*`. This has the effect of including any number of characters in a single match. Sentences consist of many characters and so this required. The next `regex` character is `?`, which means stop trying to include characters in the match as soon as a match is found. Lastly, `\.` signifies that a match is to end in a full stop.

The problem with this regular expression is that it will match sentences with leading spaces as well as sentences without leading spaces. This can be remedied by adding `\S` to the beginning of the expression. This requires a match not to start with a space. The software now provides the desired result.

Incidentally, `(.|\n)` could be used in place of `[\\s\\S]`. The parentheses are required to obtain the correct association of operators but introduces a capture group. To tell the `regex` that a capture group is not required, `?:` is inserted after the `(` character giving `(?:.|\n)`. It is possible to make the value of `reg` slightly simpler by using raw literals. This removes the need for the 'escape' characters and becomes `R"(\S[\s\S]*?\.)"` and is probably the shortest representation.

It should be noted that in the provided program `regex_replace()` removes all occurrences of one or more spaces from the sentence and replaces each occurrence with a single space.

There are still some oddities with the program, however.

There is no need to prefix standard library components, such as `cout` and `ifstream`, with the namespace `std` as the `using` directive has been used to make the contents of the `std` namespace available. Also, there is no need to read the contents of the file into a `stringstream` and then copy the `stringstream` to a `string`. The file can be read into a `string` directly. I show how this can be done in my version of the program.

The `regex` object within the control section of the `while` loop is created for every iteration of the loop and wastes significant time. The creation of the `regex` object should be brought outside the `while` loop and so created only once. The same applies to the `whitespace regex`.

Using `regex_search` repeatedly within a loop is not good practice. Although it appears to work in this application, there are `regex` iterators available that are designed to find every match within a source `string`. I have adopted this approach in my version of the program.

To make a more self-documenting program, `const` could be applied to identifiers that do not change their state after initialisation. Such objects include `whitespace` and `sentence`. It is not necessary to flush `cout` after writing every sentence and so a simple `'\n'` would suffice instead of `endl`.

It is also possible to add `regex_constants::optimize` to the `regex` constructors to optimise for speed but I did not find any improvement in this particular application. Incidentally, the original (corrected) code took about 6.2 seconds to extract 4800 sentences from a test file (but not to display them). The version I provide took 0.1 seconds. An impressive improvement.

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <regex>
#include <sstream>
#include <string>

using namespace std;

int main(int arg, char** argv)
{
  ifstream ifs(argv[1]);
  const string s((istreambuf_iterator(ifs)), {});
  const regex reg(R"(\S[\s\S]*?\.)");
  const sregex_iterator end;
  const regex whitespace("[\\s]+");
  for (sregex_iterator pos(s.cbegin(),
    s.cend(), reg); pos != end; ++pos)
  {
    const string sentence(pos->str());
    regex_replace(
      ostream_iterator<char>(cout),
      sentence.begin(), sentence.end(),
      whitespace, " ");
    cout << '\n';
  }
}
```

Hans Vredeveld <accu@closingbrace.nl>

The last character of a sentence is a period. After this period, we first get some whitespace before the next sentence starts. The regular expression in the `regex_search` statement treats this whitespace as part of this next sentence. An easy solution is to consume whitespace at the beginning of a sentence separately and not output it:

In the `regex_search`, change the regular expression to `R"([\s]*([\s\S]*?\.))"`.

In the body of the **while**-loop, initialize sentence with **m[1]**, instead of **m[0]**.

By the way, I also changed the string literal to a raw string literal for readability. In regular expressions, the backslash is used a lot and this avoids having to double it in the string literal.

Now that we have solved the main issue, let's see what else we can improve. The first thing is the inconsistent use of functions and types from the **std** namespace. Most of the time they are used by prefixing them with **std::**, but on some occasions (declaration of **s** and **m**, use of the unnamed **regex** object in the **while**-statement) they are found because of the using-directive at line 7. I have a strong preference to not use **using**-directives and would remove it. The types and functions will have to be prefixed with **std::**. Note that, because of ADL, **regex_search** still will be found in namespace **std** without prefixing it with the namespace.

Leaving the code that reads the file as is for now, we see a **while**-loop that loops over the string **s** that is defined before the loop and updated in the last statement of the loop body. We can replace this by a **for**-loop with an **sregex_iterator** as loop variable:

```
std::string s = ss.str();
std::regex re(R"([\s]*([\s\S]*?\.))");
for (std::sregex_iterator it(s.begin(),
     s.end(), re);
     it != std::sregex_iterator();
     ++it)
```

The initialization of sentence has to be updated to use ***it** instead of **m** (that is no longer used).

This change also improves performance. A crude performance measurement with a large input file (the original test input concatenated 1,000 times) shows that execution time was reduced from roughly 1.5 seconds to 0.6 seconds (on a Linux laptop with a 4-core Intel i7@2.20GHz and 16GB).

In the loop bodyb a sequence of whitespace characters in the sentence is replaced by a single space for each sentence separately. It is more efficient to do this once on the entire input before entering the loop:

```
std::regex whitespace(R"([\s]+)");
std::string s = std::regex_replace(ss.str(),
    whitespace, " ");
```

The loop body is now reduced to simply streaming to **std::cout** (also **std::endl** is replaced by **'\n'** to avoid flushing the buffer after each line):

```
std::cout << (*it)[1] << '\n';
```

As for the execution time, we are now down to an execution time of about 0.08 seconds on the same input as above.

If we include the reading from file, we now fill the string **s** by reading the file one character at a time, including whitespaces, and then replace each sequence of whitespaces by a single space. We can combine these actions by reading one word (non-whitespace characters delimited by whitespace characters) at a time, while skipping whitespace and precede the words with a space when putting them in the **stringstream**. To keep the scope of the variable containing the words local to the loop, we change the **while**-loop to a **for**-loop and declare the variable in the **init** expression of the **for**-statement. The execution time with the large input file now goes down to about 0.03 seconds.

After all this refactoring, the entire program looks as follows:

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <regex>
int main(int argc, char** argv)
{
  std::ifstream ifs(argv[1]);
  std::stringstream ss;
  for (std::string word; ifs >> word;)
    ss << ' ' << word;
  std::string s = ss.str();
  std::regex re(R"([\s]*([\s\S]*?\.))");
  for (std::sregex_iterator it(s.begin(),
    s.end(), re);
    it != std::sregex_iterator();
    ++it)
  std::cout << (*it)[1] << '\n';
}
```

There are still some opportunities for change/improvement left. I will just remark on them here.

We now have a program with only one regular expression in it. We could replace this by using **std::find**, **std::string::find** or friends, but then we would have to manually skip over the spaces between sentences. That would make our program more complex, and I like the simplicity that we have now with the regular expression. Also, it would be interesting to see what compile-time regular expressions would do for this code.

With a very small change to the regular expression, the program can also handle sentences ending in an exclamation point or question mark. It will be more difficult to handle the decimal point in numbers with a fractional part properly.

When reading the file, the program does not do any error handling of its own. Most of the errors that need to be handled are already handled by **std::ifstream**, but that doesn't give any feedback to the user. The one case that is not handled and that leads to undefined behaviour, is when **argc == 0**. In that case, **argv[1]** is undefined. (If **argc == 1**, **argv[1]** is a null pointer.)

Commentary

This critique was in part generated by some discussion about the inefficiency of the C++ **regex** library. While I believe that there are genuine issues with the specification of the library, this critique demonstrates that at least *some* of the problems are due to inappropriate use of the library.

One key item to improve the use of **regex** is the understanding that there are two parts to the process – firstly generating the regular expression and second applying it to the target. In this critique, the creation of the **regex** objects each time round the loop is particularly troubling. There was thought given, when the **regex** library was written, to supporting regular expressions in an idiomatic C++ manner, and some of the solutions provided above demonstrate this.

The other part of the commentary is to recommend using raw string literals (available since C++11) for strings where the number of escape characters would otherwise make the string unreadable. This is very common with **regex** expressions as use of the backslash is common, and each use must be escaped when using a standard string literal.

Regular expressions were the first motivating example in the original proposal, N2146, including a line from a C++ program which read:

```
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
```

While Hans noted that the code invoked undefined behaviour if **argc** was 0, no-one pointed out that the program crashes (typically) if **argc** was 1 (as the argument is expected to point to a null terminated character string). I would like to have seen this case handled, for example by writing a usage message, to improve the usability of the program.

The winner of CC 120

All the critiques resolved the presenting issue of the leading spaces – although in slightly different fashions. I'm not sure there is necessarily a 'best way' – I think it depends what fits in best with the direction of the solution.

The solutions also improved the performance of the original code; I did some basic performance measurement of the three solutions and Pete and James's solutions seem of similar speed while Hans's is noticeably faster (a result of the change he made to perform whitespace handling on input)

Pete made a good call to remove the use of the non-greedy matching specification (`"*?"`) in the original **`regex`**; I agree with him that this can make the regular expression harder to understand for the reader.

I think this critique is a good place to use **`sregex_iterator`** and I was pleased to see two solutions did so. I think this better expresses the intent of the example; it may also be more performant (but this depends upon the standard library implementation being used.)

James's solution nicely illustrates the use of CTAD (Class Template Argument Deduction) in the initialisation of the string **`s`** using **`istream_iterator`** – you may need to enable experimental C++ support in your compiler to get this to compile pending the C++20 standard.

Hans also pointed out a few problems with the code – such as in handling sentences ending with punctuation and the lack of error handling and so I am awarding him the prize for this issue's critique.

Code Critique 121

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

> I wanted to offload logging to the console from my main processing threads, so I thought I'd try to write a simple asynchronous logger class to help me do that (and hopefully learn something about threading while I'm doing it). Unfortunately it only prints the first line (or sometimes the first couple) and I'm not really sure how best to debug this as the program doesn't seem to behave quite the same way in the debugger.
>
> ```
> $ queue.exe
> main thread
> Argument 0 = queue.exe
> ```
>
> or sometimes only:
>
> ```
> $ queue.exe
> main thread
> ```

The coding is in three listings:

- Listing 2 contains `async logger.h`
- Listing 3 contains `async logger.cpp`
- Listing 4 contains `queue.cpp`.

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

Listing 2

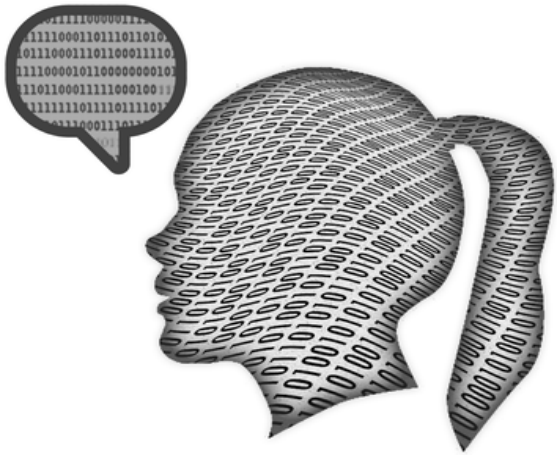
```
#pragma once

#include <iostream>
#include <queue>
#include <string>
#include <thread>

using std::thread;

class async_logger
{
  bool active_;
  std::queue<std::string> queue_;
  thread thread { [this]() { run(); } };

  void run();

public:
  async_logger();
  ~async_logger();
  void log(const std::string &str);
};
```

Listing 3

```
#include "async_logger.h"
#include <iostream>
using std::thread;

// This runs in a dedicated thread
void async_logger::run()
{
  while (active_)
  {
    if (!queue_.empty())
    {
      std::cout << queue_.front() << '\n';
      queue_.pop();
    }
  }
}
async_logger::async_logger()
{
  active_ = true;
  thread_.detach();
}
async_logger::~async_logger()
{
  active_ = false;
}

// queue for processing on the other thread
void async_logger::log(const std::string &str)
{
  queue_.emplace(str);
}
```

Listing 4

```
#include "async_logger.h"

int main(int argc, char **argv)
{
  async_logger logger;
  logger.log("main thread");
  thread test1([&logger]() {
    logger.log("testing thread 1");
  });
  for (int idx = 0; idx < argc; ++idx)
  {
    logger.log("Argument "
      + std::to_string(idx) + " = "
      + argv[idx]);
  }
  logger.log("main ending");
  test1.join();
}
```



The Standard Report

Guy Davidson reports from the C++ Standards Committee.

In this report I'm going to cover the National Body comments that were resolved at the recent meeting in Belfast. Ten motions came before the committee from the Core Working Group (CWG) for voting, and thirty-one from the Library Working Group (LWG). As promised last time, I need to start with a little more procedural detail. Brace yourself for acronyms.

Recall that in Cologne we closed the gate on the standard and declared C++20 'finished'. Of course, it isn't ACTUALLY finished. Each National Body (NB) has to check it over and ensure that it is correct. This includes my own NB, the British Standard Institute (BSI). A few weeks after the Cologne meeting the Committee Draft (CD) was published for examination. Within the BSI we divided up the work among ourselves, taking various chapters of interest. My burden included the first six clauses, covering 82 of the 1794 pages.

We had about eight weeks to complete our review and provide comments. Many people contributed, but we would of course like to share the burden. Indeed, if you have an interest in serving on the BSI C++ panel, do get in touch. Our comments were collated by our chair, Roger Orr, and submitted to the committee. In Belfast, the four working groups (WG), Library Evolution Working Group (LEWG), Evolution Working Group (EWG), LWG and CWG resolved NB comments and looked at smaller proposals for C++23, while the Study Groups (SG) continued their work of looking at new proposals for their domains.

The nature of comment resolution is that fine detail is considered on the CD. No new features are added to the Working Draft (WD). This isn't a matter of pedantry. While there are some editorial comments about grammar and style, there are also requests for clarification. The purpose of the wording is to minimise ambiguity before maximising clarity. However, there are also sometimes requests for complete removal of features. For example, in the pre-Belfast mailing the Bulgarian NB put forward a good case for the withdrawal of coroutines (individual comments aren't published as papers available from http://wg21.link).

It was the first time I had attended a committee meeting in the UK. There was a strange sense of being the host to my international friends. It was also the biggest BSI presence I had seen at a committee meeting: there were over twenty of us. On Friday, with nothing left to do before the closing plenary, we went out to find a pub for a celebratory drink, and found ourselves with a few other delegates trying to explain the finer points of Guinness and the traditions of the Irish covers band.

The next meeting is in Prague on February 9th. This should see the resolution of all NB comments and the completion of the Final Draft - International Standard (FDIS). This is sent out for ballot to the NBs, which is a simple pass/fail. If no NB fails the draft, then it will be published as C++20. Next time, I'll write about that final stage.

GUY DAVIDSON

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

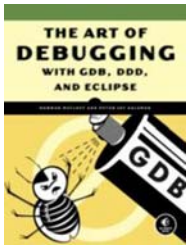
Reviews

The latest roundup of reviews.

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

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

The Art of Debugging with GDB, DDD, and Eclipse



By Norman Matloff and Peter Jay Salzman, published by No Starch Press, ISBN : 978-1-59327-174-9

Reviewed By: Ian Bruntlett

I have, in my time, debugged programs in a variety of ways and debuggers. I bought this book to become familiar with the debuggers in the GNU toolchain – that is gdb and ddd – but this book also covers the Eclipse IDE. A Linux/Unix-like environment and the C programming language is assumed although some coverage of C++ in the main text and some coverage of Java, Perl, and, Python is present in the final chapter.

Chapter 1, 'Some Preliminaries for Beginners and Pros', is well-meaning but I found its example (of an insert sort) to be confusingly written and a bad example to beginner coders. Also, the text refers to source code line numbers heavily so it would have been helpful to have line numbers for that example. However, the source code of examples is available for downloading so this is less of a problem (see nostarch.com/debugging.htm).

Chapter 2, 'Stopping to take a look around', is about breakpoints which can not only act as breakpoints but also as watch points and catch points. However, you will need to see the GNU gdb documentation to get info on catch points.

Chapter 3, 'Inspecting and Setting Variables', has an example of using a binary tree to sort input data. Whilst the examples given are good for debugging they make serious design compromises, presumably in the interest of demonstrating the debugger's facilities.

Chapter 4, 'When a Program Crashes', goes into detail about how Linux handles virtual memory and has an extensive debugging session fixing a badly broken C string handling library.

Chapter 5, 'Debugging in a Multiple-Activities Context', covers client/server network programs, multi-threading and parallel-programming, all of which are currently out of my depth.

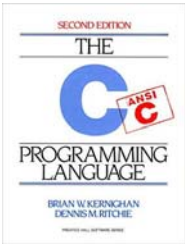
Chapter 6, 'Special Topics', covers syntax errors, shared and dynamic libraries, and debugging curses programs

Chapter 7, 'Other Tools', covers how text editors (vim in this case), the compiler, strace, ltrace, static code checkers (e.g. splint – C only) can all be used to prevent or deal with bugs and finishes off with a section on debugging dynamically allocated memory (malloc etc, free) using tools like Electric Fence, mtrace and `MALLOC_CHECK_`.

Chapter 8, 'Using GDB/DDD/Eclipse for Other Languages', covers Java, Perl, Python, SWIG, and (32-bit Intel) assembly language debugging.

A good book, especially if used alongside the dedicated manuals available for the tools covered here (www.gnu.org/manual/manual.html is a good place to start). You will need a background in C and Linux/Unix to get the most out of this book.

The C Programming Language 2e



Published by Prentice-Hall, ISBN 0-13-110362-8

Reviewed by Ian Bruntlett

This is the classic introduction to the C programming language. The first edition was originally published in 1978 and this second edition was published in 1988. A lot of things have changed since then but this is a worthwhile introduction nevertheless. There are other books on C but I am not sure which books to read after this one – if I could have a modern bibliography for this book, it would come in very handy – I intend to review other C books, in time.

After an introductory chapter, the basics of the language are catered for in this order:

- Types, operators, and expressions
- Control flow
- Functions and program structure
- Pointers and arrays
- Structures
- Input and output

After that, there is a chapter on 'The UNIX System Interface', which features interesting albeit limited implementations of **`fopen`**, **`getchar`**, **`putchar`**, **`opendir`** and **`malloc`**.

The appendices round off the book. The Reference Manual is based on the draft ANSI standard and presents a grammar of the language. The Standard Library appendix complements the grammar by offering up a summary of the standard library. The final appendix summarises the changes made to the language since the publication of the first edition of this book.

The source code for this book is available to download but I found the organisation of the code to be lacking. Fortunately, a kind soul has reorganised the code and made it available on GitHub (https://github.com/caisah/K-and-R-exercises-and-examples). Most Internet searches for this book bring up sites that either want to sell you the book or give you a pirate PDF copy. Because of that, I'm citing an errata page (https://s3-us-west-2.amazonaws.com/belllabs-microsite-dritchie/cbook/2ediffs.html)

The code examples are for tuition purposes only – this isn't the place to go looking for production code – however the UNIX System Interface chapter comes close.

Finally, the book ends with an index – a glossary would have been welcome, too. This book does show its age a bit but it still should be on the reading list of any would-be C programmer.

View from the Chair

Bob Schmidt
chair@accu.org

ACCU Autumn 2019

ACCU held a two-day conference in Belfast in early November, in conjunction with the BSI hosting the autumn WG21 meeting. Herb Sutter and Michael Wong were the opening and closing keynote speakers (respectively); there were eight session times and three tracks; lightning talks and a conference dinner which was attended by almost all attendees. Although this is the first time an event like this has been held in Belfast, over 100 persons attended.

Congratulations to Jamie Allsop and Roger Orr for coordinating the conference and chairing its program. Thanks also to event manager Laura Nason of Archer Yates Associates (assisted by Julie Archer) for doing a fantastic job of organizing both the WG21 meeting and the ACCU event.

Of course, a conference is only as good as its programme and we did have a fantastic line up. For that we must thank all those on the programme committee who each made a valuable contribution: Russel Winder, Timur Doumler, John McFarlane, Chris Kohlhoff, Garth Gilmour, and Felix Petriconi. Last but not least the speakers deserve a mention, for they ensured the delegates enjoyed the high quality of talk we've become accustomed to at an ACCU event.

This was ACCU's first autumn conference since 2011. I'm told that Jamie would like to make this an annual event. Given his success this year, I think we can look forward to ACCU Autumn 2020.

My thanks to Jamie Allsop and Julie Archer for providing information on the Belfast conference.

ACCU 2020

ACCU's flagship conference is scheduled for the 25th through 28th of March, 2020, with pre-conference workshops scheduled for March 24th. The conference once again will be held at the Bristol Marriott City Centre. Proposals have been submitted, and the conference committee is busy putting together the schedule of classes and workshops.

Keynote speakers for the conference have been announced, and they are: Patricia Aas (Owner, TurtleSec); Emily Bache (Technical Agile Coach, ProAgile); Kevlin Henny (consultant); and Sean Parent (Senior Principal Scientist, Adobe).

Registration for ACCU 2020 should be open around the time this issue is published. Make plans to join us in Bristol in March.

Important dates for the AGM

Date	Deadline	Countdown
29 December 2019	AGM announcement deadline	(AGM – 90)
28 January 2020	Nomination/proposal deadline	(AGM – 60)
15 February 2020	Draft agenda deadline	(AGM – 42)
29 February 2020	Agenda freeze	(AGM – 28)
7 March 2020	GM voting opens	(AGM – 21)
28 March 2020	AGM	

2020 AGM

ACCU's Annual General Meeting will be held on Saturday, 28 March 2020 at the Bristol Marriott City Centre, in conjunction with the conference. All members are encouraged to attend.

The important dates for the AGM are shown in the table at the top of the page.

The main objective of the AGM is election of committee members. Please note that the deadline for committee nominations is 28 January 2020. ACCU's Constitution paragraph 5.3.3 states

> Any member of the Association can stand as a candidate for election to any role on the committee. Any such member shall notify the Secretary in writing (letter or email), including names of a nominating member and a seconder, on or before the Proposal Deadline (described in 'Section 7 – General Meetings'). The same person cannot stand for more than one role in the same election.

If you are interested in serving on the committee, please make your interest known to our secretary, Patrick Martin (secretary@accu.org).

Call for volunteers

We are still trying to fill several chronically vacant committee positions. As a reminder, this summer the committee announced a new policy to try to encourage participation. Anyone who volunteers for one of the positions the committee deems chronically vacant will qualify for a temporary membership deferral. If already a member, a new volunteer will have their next membership payment deferred for one month for each month of service, up to a year. If not currently a member, a new volunteer will be instated as a member for up to a year, as long as they continue in the role.

The positions the committee has determined are chronically vacant are: publicity, study groups, social media, and web editor. In addition, the position of treasurer will qualify for the incentive, in order to allow Rob Pauer to retire after many years of service to ACCU.

Please contact me (chair@accu.org) if you are interested in one of these positions.

We need your help!

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

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

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

"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

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

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

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

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

qbs SOFTWARE