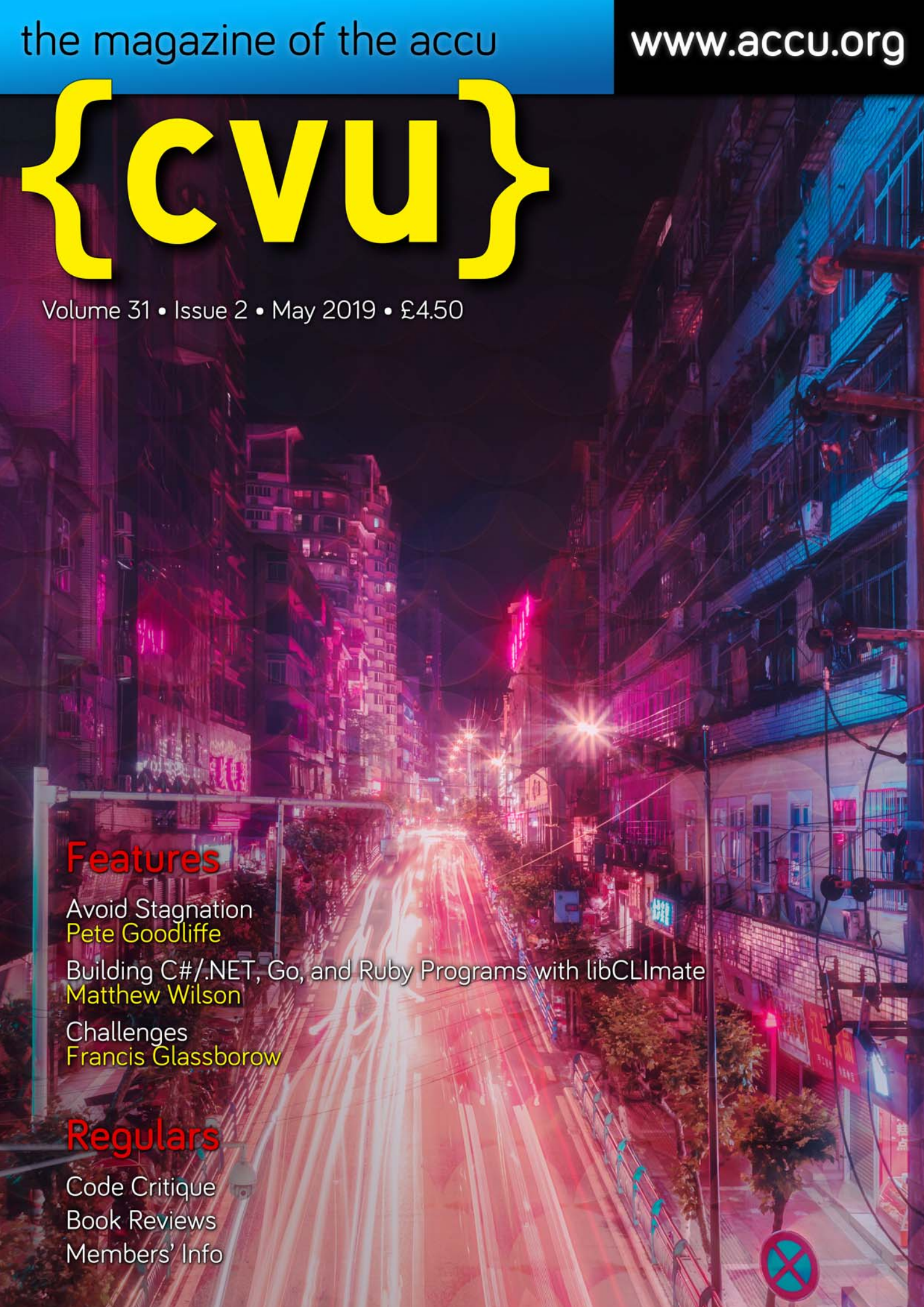
the magazine of the accu

www.accu.org

{cvu}

Volume 31 • Issue 2 • May 2019 • £4.50



Features

Regulars

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe

{cvu}
Volume 31 Issue 2
May 2019
ISSN 1354-3164
www.accu.org

accu

Full Nine Yards

Over the last few years, the term 'full-stack developer' has become a popular requirement in job adverts for programmers. It's commonly used to mean someone who can write code at each level of a multi-tiered application, from the UI, down through controller and logic layers, all the way to the database. The implication is that modern developers need to be masters of multiple technologies. UI very often means web-based, and there are myriad frameworks (to put it mildly!) in this space. Then there's the interop layer to the server-side, which itself might comprise multiple technologies, and finally a host of different database offerings.

Another term that's come to prominence of late is 'dev-ops', commonly used to mean someone who can write code, and understands the various technologies used for cloud-computing – of which, once again, there are many. This implies at least some level of security knowledge is needed, not just about user security, but for deployments, monitoring, support and so on.

A universal requirement for developers who can do *everything* risks over-generalization, and watering down the level of expertise needed for true mastery of a small number of specialized technologies. Writing a good UI that delivers the best user experience is a highly-skilled task. I've been privileged to meet some excellent UI designers, and some excellent computer programmers, but it's rare to find someone who excels in both. In the same vein, cyber-security is a very specialized skill, in particular for public-facing applications.

Chris Oldwood gave a talk at the ACCU 2019 conference, where he looked at multi-skilled developers from a slightly different perspective: the Full Pipeline Developer, who understands the full application life cycle from architecture, writing code, testing at all levels, build systems, deployment, monitoring and support. This definitely rang true for me in a way that neither 'full-stack' nor 'dev-ops' does: it captures the need to understand the stages required in software development in principle, rather than in terms of specific technologies or brands.

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 31.3: 1st June 2019
C Vu 31.4: 1st August 2019

Overload 151: 1st July 2019
Overload 152: 1st September 2019

WRITE FOR C VU

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

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

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Avoid Stagnation

Pete Goodliffe cautions us against allowing our programming skills to become stale.

Iron rusts from disuse; water loses its purity from stagnation... even so does inaction sap the vigor of the mind.
~ Leonardo da Vinci

When was the last time you learnt something new and exciting enough to put on your CV? When was the last time you were stretched beyond your capabilities? When was the last time your work made you feel uncomfortable? When was the last time you discovered something that delighted you? When were you last humbled by another programmer and encouraged to learn from them?

If the answers to these questions are 'the dim and distant past' then you have entered the *comfort zone*: a place that some regard as nirvana – where your life is easy and your work days are short and predictable. (Note that the 'distant past' is not so long ago when you measure in *programmer years*, which is why people find it so hard to estimate the duration of software projects!)

However, the comfort zone is a pernicious place. It's a trap. An easy life means you're not learning, not progressing, not *getting better*. The comfort zone is where you stagnate. Pretty soon you'll be overtaken by younger developer upstarts. The comfort zone is an express route to obsolescence.

> Be wary of stagnation. Seeking to become a better programmer, by definition, is not the most comfortable lifestyle.

Few people make a conscious decision to stagnate. But it can be easy to slip into the comfort zone and coast along your development career without realising. Take a reality check: is this what you're doing right now?

Your skills are your investment

Beware: maintaining your skill set is hard work. It involves putting yourself in uncomfortable situations. It requires a very real investment of effort. It can be risky and hard. You might even embarrass yourself. That doesn't sound entirely pleasant, does it?

It's therefore not something that many people feel naturally inclined to do. You spend so many hours of the day working, don't you deserve to have an easy life and then go home to forget all about it? It's natural to learn towards the familiar and the comfortable.

Don't do it!

You have to make a conscious decision to invest in your skills. And you have to make that decision repeatedly. Don't see it as an arduous task. Delight in the challenge. Appreciate that you are making an investment that will make you a better programmer, and a better person.

> Expect to invest time and effort to grow your skill set. This is a worthwhile investment; it will repay itself.

An exercise for the reader

How can you shake yourself up right now? Here are some changes to make that will push you out of the comfort zone:

- Stop using the same tools; there might be better ones that will make your life easier if you'd only learn about them.
- Stop using the same programming language for every problem; you might be smashing a walnut with a sledgehammer.
- Start using a different OS. Learn how to use it properly. Even if it's one that you don't like, spend a while trying it out to really learn its strengths and weaknesses.
- Start using a different text editor.
- Learn keyboard shortcuts and see how it impacts your workflow. Make a conscious effort to stop using a mouse.
- Learn about a new topic, something that you don't currently *need* to know. Perhaps deepen your knowledge of maths or of sorting algorithms.
- Start a personal pet project. Yes, use some of your precious spare time to be geeky. Publish it as open source.
- Manoeuvre yourself to work on a new part of your project, one you know little about. You might not be productive immediately, but you'll gain a wider knowledge of the code and will learn new things.

Consider expanding yourself beyond the programming realm:

- Learn a new language. But *not* a programming language. Listen to an audiobook teaching series on Japanese on your drive into work.
- Rearrange your desk! Try to look at the way you work in a new light.
- Start a new activity. Perhaps start a blog to journal your learning. Spend more time on a hobby.
- Take up exercise: join a gym or start running.
- Socialise more. Spend time with geeks *and* with non-geeks.
- Consider adjusting your diet. Or going to bed earlier.

Job security

Being a better developer, one with a more rounded skillset, one who is constantly learning, will increase your job security. But ask yourself if you really need that: *are* you in the right job?

Hopefully you are in the right career: you enjoy programming. (If you don't, consider seriously if a career change might be a good option. What would you *really* like to do?)

There is a danger in staying in one job or one role too long, of doing the same thing over and over with no new challenges. All too easily, we get entrenched in what we're doing. We like being local experts; the king of our little coding castle. It's comfortable.

Perhaps it's now time to move on to a new employer? To face new challenges and move on in your coding journey. To escape the comfort zone.

Staying put is usually easier, more familiar, and more convenient. In the recent rocky economic climate, it's also the safer bet. But it might not be the best thing for you. A good programmer is courageous, both in their approach to the code and their approach to their career. ■

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

Building C#/.NET, Go, and Ruby Programs with libCLImate – Part 1: Ruby

Matthew Wilson demonstrates command-line processing.

This article, the fourth in a series about software anatomy, describes some of my ongoing efforts to provide simple, succinct, easy, and, most importantly, consistent command-line argument processing across multiple languages. It, and the next one, build upon the observations of the first three articles, which pertained to building command-line interface (CLI) programs in C and C++, illustrating how the topic may be handled for the languages C#/.NET, Go, and Ruby, all of which have featured heavily in my career in the intervening years. As well as introducing readers to the respective Go, .NET and Ruby variants of the **libCLImate** and **CLASP** libraries, I will discuss how the features of these younger languages offer increased expressiveness and flexibility over C and C++, including how to work around some of their limitations.

Introduction

Command-line interface (CLI) programs often take arguments from the operating environment in order to control their behaviour. The shell breaks up the command-line (based on whitespace, except where quotes are operative) and presents this sequence of argument strings to the program. There are several, sometimes contradictory, conventions for how to classify arguments, including by the IEEE [1], GNU [2], Go [3].

Consider the command-line:

```
$ count_slocs "C++ code" -D --terse
  --log-file=./log1 '*.cpp' '*.hpp'
```

In this case, the program `count_slocs` would be passed six arguments, based primarily on the separating whitespace. There's a label argument `"C++ code"` which is enclosed within double-quotes, which instructs the shell to treat all characters within (but not including) the double-quotes as a single argument. There are two arguments containing wildcards – `'*.cpp'` and `'*.hpp'` – that are each enclosed within single quotes, in order to tell the shell not to expand them. Each of these three arguments is just a *value*.

The remaining three arguments are different, insofar as they are prefixed with hyphens. The arguments `-D` and `--terse` are *flag*s. The argument `--log-file=./log1` is an *option*.

The terms *flag*, *option*, and *value* are my names for these concepts, as described in my article 'An Introduction to CLASP, part 1: C' [4]. The basic rules are pretty simple:

- if the argument does not begin with a hyphen (`-`) then it is a *value*; else
- if the argument contains an equals sign (`=`) then it is an *option* (and the parts before and after the equals sign are known as the *option name* and *option value*, respectively); else
- it is a *flag*.

There are, of course, complications, including:

- After the first occurrence of the special flag `--`, all arguments should be treated as values (regardless of any leading hyphen(s));
- Some program(mer)s prefer to be able to specify options separated by a space, e.g. the above option would alternatively be specified as `--log-file ./log1`. Naturally, for this to be recognised as two (separate argument) parts of a single option, something in the program has to know that `--log-file` is an option name, and that the next argument will the option value;
- Some program(mer)s prefer to be able to specify multiple flags (that have a single hyphen and a single letter) in combination so that, for example, the following command-line consisting of such flags:

  ```
  $ myprog -c -D -x
  ```

  can be expressed alternately as

  ```
  $ myprog -cDx
  ```

  and obtain precisely the same behaviour.

(There's also a common convention to use the special flag `"-"` to mean standard-input, but that doesn't affect parsing so I won't mention it further.)

Obtaining command-line arguments

Different languages have different mechanisms for obtaining command-line arguments:

- C and C++ are provided the argument pair `argc : int , argv : char*[]` to the program entry point `main()`;
- C#/.NET are provided an array of `System.String` to the program entry point `Main()`;

MATTHEW WILSON

Matthew is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

Avoid Stagnation (continued)

Questions

- Are you currently stagnating? How can you tell?
- What was the last new thing you learnt?
- When did you last learn a new language? When did you last learn a new technique?
- What new skill should you learn next? How will you learn it? What books, courses, or online material will you use?
- Are you in the right job at the moment? Do you enjoy it, or has all the joy been sucked out? Are you working the 9-to-5, or are you enthused and engaged to see your project succeed? Should you look around for new challenges?
- When did you last get a promotion? Or a pay raise? Does a job title have any real meaning? Does your job title bear any relation to your skills?

- Python and Ruby do not have program entry points in the same way, and instead obtain the `sys.argv` property (a list) and global variable `ARGV` (an array), respectively;
- While Go does have a program entry point, it takes no arguments. The program-arguments are obtained from the exported array (`[]string`) `os.Argv`.

CLASP

CLASP stands for Command-Line Argument Parsing and Sorting. In essence, CLASP libraries parse command-lines to recognise flags, options, and values, and then presents them in a convenient set of sequences as part of a data structure representing the command-line (along with other useful information, such as program-name, and so forth).

The CLASP library concept started out with a C/C++ library, cunningly called CLASP, which was introduced to the world in [5] and further discussed in the three preceding parts of this anatomies series [4] [6] [7].

CLASP has subsequently been implemented in a number of different languages, including C#/.NET, Go, JavaScript, Python, and Ruby (all of which projects are available under https://github.com/synesissoftware).

As is the way of these things, different amounts of attention applied at different times, along with the strong influence of language limitations and conventions, have led to some differences, but fundamentally all perform the parsing and sorting (into flags, options, and values).

I found the use of CLASP to be a huge boon to productivity, not only because it provides often missing functionality, but also because having the same model in different languages means thinking time is kept to a useful minimum.

libCLImate

Despite the considerable advantages afforded by using CLASP, there are still tasks that one finds oneself performing again and again and again across projects. The most obvious ones are handling of the *de facto* standard `--help` and `--version`. But there are plenty of others, including:

- checking whether any unrecognised flags/options are received;
- checking whether any required flags/options have been specified;
- checking whether too many, or too few, values have been specified;
- issuing contingent reports of a consistent form, in particular that they begin with `program-name :`;

…and many more.

The situation begged for a higher-level library, implemented in terms of CLASP, that provides most/all of this behaviour.

In the third part of this series, 'Building C & C++ CLI Programs with the libCLImate Mini-framework' [7], I described in detail the design and implementation of the original **libCLImate** library, which provide many of these extra facilities for C/C++. Since that time (2015), I have continued to use the notion of a library providing boilerplate facilities that it is implemented in terms of CLASP, and have so far built and released variants for Go, C#/.NET, and Ruby. (It is reasonably likely that that Python version will be released by the time you're reading this.)

(NOTE: after an exhaustive process of consultation with the esteemed members of the ACCU, the name **libCLImate** was accepted so that "every pull request will result in CLImate change". Arf. Arf.)

As described in [7], due to the limitations of the runtime libraries the C/C++ **libCLImate** includes additional features and dependencies for diagnostics. Furthermore, because of the inherent limitations of the language, and because the library supports both C and C++, setting up many elements for a given program are still quite verbose. I plan soon to use some C++-11/14/17 features to address the second problem, and am optimistic that it can all be made simpler and more succinct in client code. However, the dependencies on the diagnostic libraries will likely stay, because they solve a still-glaring omission of the C/C++ environment.

Conversely, with the other languages, the implementation of **libCLImate** is much simpler, and the use of the library in programs is very succinct indeed, as I will shortly demonstrate. I think this has been because **libCLImate.Ruby** was the first (after the original), and I've noticed that when I write libraries in Ruby that subsequently are ported to other languages that the high expressiveness common in Ruby tends to get carried across.

libCLImate.Ruby

As mentioned above, since this is the oldest port, it's also the most fully featured (and widely tested). Let's consider how we might implement the notional program `count_slocs` from the introduction.

Basic boilerplate

The basic boilerplate of a libCLImate.Ruby program is as shown below:

```
#! /usr/bin/env ruby
require 'libclimate'
options = {}
climate = LibCLImate::Climate.new do |cl|
  # customise here
end
r = climate.parse ARGV
```

Even without customising it in any way, we get a fair amount of useful functionality:

- Support for `--help`:

  ```
  $ ./count_slocs --help
  ```

  yields the output:

  ```
  USAGE: count_slocs [... flags and options...]
  flags/options:
  --help
    shows this help and terminates
  --version
    shows version and terminates
  ```

- Policing any unrecognised flags:

  ```
  $ ./count_slocs -D
  ```

  yields the contingent report (and non-0 exit code):

  ```
  count_slocs: unrecognised flag '-D'; use
  --help for usage
  ```

- Policing any unrecognised options:

  ```
  $ ./count_slocs --log-file=./log1
  ```

  yields the contingent report (and non-0 exit code):

  ```
  count_slocs: unrecognised option '--log-
  file=./log1'; use --help for usage
  ```

Specifying version

However, if we run with `--version` we get a nasty surprise, as shown in Figure 1, overleaf.

This is because we haven't specified any version information. This can be done as a string, or an array of strings or integers (or mixed), as in:

```
climate = LibCLImate::Climate.new do |cl|
  # customise here
  cl.version = [ 0, 0, 1 ]
end
```

Now running with `--version` gives:

```
count_slocs 0.0.1.alpha-1
```

Specifying program-specific flags

Now let's add support for the flags `-D` and `--terse`. This is done via the `Climate#add_flag` method, which takes a name, an optional number of options, and an optional block to be executed if the flag is specified on the command-line (see Listing 1).

Figure 1

```
[...].../gems/clasp-ruby-0.16.0/lib/clasp/cli.rb:88:in
`generate_version_string_': options must specify :version or :version_major [ + :version_minor [ +
:version_revision [ + :version_build ]]] (ArgumentError)
  from [...].../gems/clasp-ruby-0.16.0/lib/clasp/cli.rb:282:in
`show_version'
  from
[...]libCLImate/libCLImate.Ruby/trunk/lib/libclimate/climate.rb:179:in `show_version_'
  from
[...]libCLImate/libCLImate.Ruby/trunk/lib/libclimate/climate.rb:321:in `block in initialize'
  from
[...]libCLImate/libCLImate.Ruby/trunk/lib/libclimate/climate.rb:466:in `block in parse'
  from
[...]libCLImate/libCLImate.Ruby/trunk/lib/libclimate/climate.rb:438:in `each'
  from
[...]libCLImate/libCLImate.Ruby/trunk/lib/libclimate/climate.rb:438:in `parse'
  from ./count_slocs:9:in `<main>'
```

Listing 1

```
climate = LibCLImate::Climate.new do |cl|
  cl.version = [ 0, 0, 1, 'alpha-1' ]
  cl.add_flag('-D', help: 'runs in debug mode') {
    options[:debug] = true }
  cl.add_flag('--terse',
    help: 'minimal program output') {
    options[:terse] = true }
end
```

When either of these flags is specified on the command-line its respective presence is recorded in the **options** hash, for consumption in the program proper. The **help** strings are used for the **--help** flag, which I'll show in full later.

Specifying program-specific options

Adding support for the **--log-file** option follows in the same vein (see Listing 2).

As you can see, this is very similar to adding a flag, except that the block takes a parameter of the actual parsed option, the value of which is placed into the **options** hash. If **o.value** is nil, which would happen if the last argument of the command-line is **--log-file** (without an **=**) an abort is executed, as in:

```
$ count_slocs -D --terse --log-file
```

which yields the contingent report (and non-0 exit code):

```
count_slocs: no log-file specified
```

If you also wished to reject an empty log-file (e.g. is passed **--log-file=**) then you could do:

```
( options[:log_file] = ( o.value ||'')).empty?
and cl.abort "no log-file specified"
```

Constraining values

Let's now consider the values. In the example we had three values: the label, the implementation files filter, and the header files filter. Now obviously in the real world it's more likely that rather than two filters there'd be one or more filters, but for now let's pretend we want exactly two. As the program stands right now you can specify any number of values and it won't care.

Listing 2

```
climate = LibCLImate::Climate.new do |cl|
  ...
  cl.add_flag('--terse',
    help: 'minimal program output') {
    options[:terse] = true }
  cl.add_option('--log-file',
    help: 'specifies the log-file') do |o|
    options[:log_file] = o.value or \
      cl.abort "no log-file specified"
  end
end
```

One way to do this is to obtain the values from the parse result, **r**, as in:

```
r = climate.parse_and_verify ARGV
unless 3 == r.values.size
  climate.abort "must specify three values; use
  --h-help for usage"
end
```

But that's just a generic message, and you're having to manually check. Instead, you can constrain the values, and give meaningful names to the values by customising the **Climate** instance (see Listing 3).

Now, if you run with, say, one argument:

```
$ ./count_slocs "C++ code"
```

You'll get the contingent report (and non-0 exit code):

```
count_slocs: implementation files filter not
specified; use --help for usage
```

And with two:

```
$ ./count_slocs "C++ code" '*.cpp'
```

You'll get the contingent report (and non-0 exit code):

```
count_slocs: header files filter not specified;
use --help for usage
```

Enhancing usage

If you take the advice and specify **--help** then you'll get the output shown in Figure 2, overleaf.

This is reasonably good, but it can be better. We can specify some informational lines as shown in Listing 4.

Now **--help** gives the output shown in Figure 3.

Listing 3

```
climate = LibCLImate::Climate.new do |cl|
  ...
  cl.add_option('--log-file', help: 'specifies
    the log-file') do |o|
    options[:log_file] = o.value or \
      cl.abort "no log-file specified"
  end
  cl.constrain_values = 3
  cl.value_names = [
    'label',
    'implementation files filter',
    'header files filter',
  ]
  cl.usage_values = '<label> <impl-filter>
  <hdr-filter>'
end
```

Figure 2

```
USAGE: count_slocs [ ... flags and options ... ]
<label> <impl-filter> <hdr-filter>

flags/options:
  --help
    shows this help and terminates
  --version
    shows version and terminates
  -D
    runs in debug mode
  --terse
    minimal program output
  --log-file=<value>
    specifies the log-file
```

Listing 4

```
climate = LibCLImate::Climate.new do |cl|
  ...
  cl.usage_values = '<label> <impl-filter>
    <hdr-filter>'
  cl.info_lines = [
    'libCLImate.Ruby example for CVu',
    nil,
    :version,
    nil,
  ]
end
```

Flag aliases

With the current definitions, we have to specify `-D` and `--terse` separately. However, CLASP (in all its guises) allows the combination of single-hyphen single-letter flags. This can be easily achieved by adding an alias property `-t` for `--terse`, as in:

```
climate = LibCLImate::Climate.new do |cl|
cl.version = [ 0, 0, 1, 'alpha-1' ]
cl.add_flag('--terse', alias: '-t', help:
'minimal program output') {
  options[:terse] = true }
end
```

Now the two can be combined, as in:

```
$ ./count_slocs -Dt ... other args ...
```

And this acts exactly as if they'd been supplied separately.

Option aliases

Options can have aliases two. For example, we could add an alias for `--log-file` to, say, `-f`, just in the same way as done for the flag above. This would allow for the long form expression seen already, as well as for a short(er) form as in:

```
$ ./count_slocs -l ./log1 ... other args ...
```

But there's a much more interesting way of using aliases with options. Consider that we change the flag `--terse` to an option `--verbosity`

Figure 3

```
libCLImate.Ruby example for CVu
count_slocs 0.0.1.alpha-1
USAGE: count_slocs [ ... flags and options ... ]
  <label> <impl-filter> <hdr-filter>
flags/options:
  --help
    shows this help and terminates
  --version
    shows version and terminates
  -D
    runs in debug mode
  --terse
    minimal program output
  --log-file=<value>
    specifies the log-file
```

Listing 5

```
VERBOSITIES = %w{ silent terse normal chatty
  verbose }
climate = LibCLImate::Climate.new do |cl|
  ...
  cl.add_flag('-D', help: 'runs in debug mode') {
    options[:debug] = true }
  cl.add_option('--verbosity', help: 'specifies
    verbosity of program output',
    values: VERBOSITIES) do |o|
      options[:verbosity] and \
        cl.abort 'already specified verbosity'
      options[:verbosity] = o.value
  end
  VERBOSITIES.each do |v|
    # === cl.add_alias('--verbosity=terse', '-t')
    cl.add_alias("--verbosity=#{v}", "-#{v[0]}")
  end
  ...
end
```

that can take a set of verbosity values and that, further, we have the flag `-t` instead be equivalent to specifying `--verbosity=terse`. This is supported by all versions of CLASP, and that filters through to use in **libCLImate.Ruby** (see Listing 5).

This code specifies an option, `--verbosity`, along with description string, a values list, and several combinable flag aliases, which would mean that the `-Dt` (and so forth) argument is still valid. And all this information is useful for the usage when applying `--help` (see Figure 4).

Obtaining results

In a real program, you will likely use blocks for flags and options as I've shown. But if you prefer, you can instead process them from the collections supplied in the parse result. For now, we'll just print them out thusly:

```
puts "flags (#{r.flags.size}):"
r.flags.each { |f| puts "\t#{f}" }
puts "options (#{r.options.size}):"
r.options.each { |o| puts "\t#{o}" }
puts "values (#{r.values.size}):"
r.values.each { |v| puts "\t#{v}" }
```

Figure 4

```
libCLImate.Ruby example for CVu
count_slocs 0.0.1.alpha-1
USAGE: count_slocs [ ... flags and options ... ]
   <label> <impl-filter> <hdr-filter>
flags/options:
  --help
    shows this help and terminates
  --version
    shows version and terminates
  -D
    runs in debug mode
  -s --verbosity=silent
  -t --verbosity=terse
  -n --verbosity=normal
  -c --verbosity=chatty
  -v --verbosity=verbose
  --verbosity=<value>
    specifies verbosity of program output
    where <value> one of:
      silent
      terse
      normal
      chatty
      verbose
  --log-file=<value>
    specifies the log-file
```

With the command-line

```
$ ./count_slocs -Dt --log-file=./log1 "C++ code"
'*.cpp' '*.hpp'
```

This yields the output:

```
flags (1):
  -D
options (2):
  --verbosity=terse
  --log-file=./log1
values (3):
  C++ code
  *.cpp
  *.hpp
```

Final code

All, up all this code (excluding the printing of the flags/options/values collections) comes in at under 45 lines of code, as shown in full in Listing 6.

Listing 6

```
#! /usr/bin/env ruby
require 'libclimate'
VERBOSITIES = %w{ silent terse normal chatty
  verbose }
options = {}
climate = LibCLImate::Climate.new do |cl|

  # customise here
  cl.version = [ 0, 0, 1, 'alpha-1' ]
  cl.add_flag('-D', help: 'runs in debug mode') {
    options[:debug] = true }
  cl.add_option('--verbosity', help: 'specifies
    verbosity of program output',
    values: VERBOSITIES) do |o|
      options[:verbosity] and \
        cl.abort 'already specified verbosity'
      options[:verbosity] = o.value
  end

  VERBOSITIES.each do |v|
    cl.add_alias("--verbosity=#{v}", "-#{v[0]}")
  end

  cl.add_option('--log-file', required: true,
    help: 'specifies the log-file') do |o|
      options[:log_file] = o.value or \
        cl.abort "no log-file specified"
  end

  cl.constrain_values = 3
  cl.value_names = [
    'label',
    'implementation files filter',
    'header files filter',
  ]
  cl.usage_values = '<label> <impl-filter>
    <hdr-filter>'

  cl.info_lines = [
    'libCLImate.Ruby example for CVu',
    nil,
    :version,
    nil,
  ]
end
r = climate.parse_and_verify ARGV
```

Next time

In the next instalment I'll provide a much more chopped down presentation of equivalent programs in C# using libCLImate.NET and Go using libCLImate.Go, and then will go into details about the different facilities in the respective languages that help with the expressiveness and flexibility of such libraries. More importantly, perhaps, I will also go into those aspects in the languages that impede such, and how I have gone about addressing them with somewhat advanced features - anonymous structures and reflection in C#; extensions and monkey-patching in Ruby; strong typedefs and type-switches in Go - in order to have a consistent programmer and user experience across the different technologies.

For now, here're three teaser trailers for next time.

More powerful option value processing in Ruby

As a little teaser for next time, permit me to illustrate how the verbosity values can be made even than bit more useable, to both user and programmer, by allow option value initials to be specified on the command-line and then automatically tested and converted into Ruby symbols in the code. Consider the changes in Listing 7.

This uses one of the **`String`** class extensions provided by the **Xqsr3** library (another of mine - https://github.com/synesissoftware/xqsr3) to allow string values to be interpreted in full or via a contraction (the part inside the square brackets) against a known set of values such as **`VERBOSITIES`**, obtaining as the result a Ruby symbol. If not matched, **`nil`** is obtained, and the **`abort`** is fired. For example, the command-line

```
$ ./count_slocs --verbosity=s ... other args ...
```

cause the **`options[:verbosity]`** value to be **`:silent`**, whereas the command-line

```
$ ./count_slocs --verbosity=j ... other args ...
```

precipitates the contingent report (and a non-0 exit code):

```
count_slocs: invalid verbosity 'j'; use --help
for usage
```

count_slocs in Go

Listing 8 (overleaf) contains the equivalent program in Go (written in terms of libCLImate.Go and CLASP.Go).

count_slocs in C#

Listing 9 (also overleaf) contains the C# version.

Listing 7

```
require 'libclimate'
require 'xqsr3/extensions/string/
  map_option_string'
...

VERBOSITIES = %w{ [s]ilent [t]erse [n]ormal
  [c]hatty [v]erbose }
...
  cl.add_option('--verbosity', help: 'specifies
    verbosity of program output',
    values: VERBOSITIES) do |o|
      options[:verbosity] and cl.abort 'already
        specified verbosity'
      options[:verbosity] =
        o.value .map_option_string(VERBOSITIES)
        or \
        cl.abort "invalid verbosity '#{o.value}';
        use --help for usage"
  end
  VERBOSITIES.each do |v|
    v = v.gsub /[\[\]]/, ''
    cl.add_alias("--verbosity=#{v}", "-#{v[0]}")
  end
  ...
```

Listing 8

```
package main
import (
  clasp "github.com/synesissoftware/CLASP.Go"
  libclimate "github.com/synesissoftware/
    libCLImate.Go"
  "fmt"
  "os"
  "strings"
)
const (
  verbosities =
   "silent|terse|normal|chatty|verbose"
)

func main() {
  debug := false
  verbosity := ""
  fl_Debug := clasp.Flag("-D").
    SetHelp("runs in debug mode")
  opt_Verb := clasp.Option("--verbosity").
    SetHelp("specifies verbosity of program
    output").
    SetValues(strings.Split(verbosities, "|")...)
  climate, err := libclimate.Init(func
      (cl *libclimate.Climate) (error) {
    cl.AddFlagFunc(fl_Debug, func () {
      debug = true })
    cl.AddOptionFunc(opt_Verb, func
      (o *clasp.Argument, a *clasp.Alias) {
        verbosity = o.Value
    })
    for _, v := range(strings.Split(verbosities,
      "|")) {
        cl.AddAlias("--verbosity=" + v,
          "-" + v[0:1])
    }
    return nil
  })
  if err != nil {
    fmt.Fprintf(os.Stderr,
      "failed to create CLI parser: %v\n", err)
  }
  _, _ = climate.ParseAndVerify(os.Args)

  // Program-specific processing of flags/options
  if 0 != len(verbosity) {
    fmt.Printf("verbosity is specified as: %s\n",
    verbosity)
  }
  if debug {
    fmt.Printf("Debug mode is specified\n")
  }
  // Finish normal processing
  return
}
```

Listing 9

```
namespace CVu_example
{
  using global::LibCLImate;
  using Clasp =
    global::SynesisSoftware.SystemTools.Clasp;
  using System;
  static class Program
  {
    static int Main(string[] args)
    {
      bool debug = false;
      string verbosity = null;
      Climate climate = Climate.Init((cl) => {
        cl.AddFlag("--debug"
        , new {
          Alias = "-d",
          Help = "runs in Debug mode" }
        , (f, a) => {debug = true;}
      );

        cl.AddOption("--verbosity"
        , new
        {
          Alias = "-v",
          Help = "specifies the verbosity",
          Values = new string[] { "silent",
            "terse", "normal", "chatty",
            "verbose" },
          Default = "chatty",
          Required = true,
        }, (o, a) =>
        {
          verbosity = o.Value;
        });
        cl.AddAlias("--verbosity=chatty", "-c");

        cl.SetInfoLines(
          "libCLImate.NET examples",
          null,
          ":version:",
          null
        );
        cl.SetUsageValues("<path>");
      });

      var r = climate.ParseAndVerify(args);

      if(null != verbosity)
      {
        Console.WriteLine(
          "verbosity is specified as: {0}",
          verbosity);
      }

      if(debug)
      {
        Console.WriteLine(
          "Debug mode is specified");
      }
      return 0;
    }
  }
}
```

References

[1] The Open Group Base Specifications Issue 7, 2018 edition, section 12: Utility conventions, at http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

[2] GNU manual: http://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html

[3] The Go Programming Language: https://golang.org/pkg/flag

[4] Anatomy of a CLI Program written in C, Matthew Wilson, *CVu* 24.4, September 2012.

[5] An introduction to CLASP, part 1: C, Matthew Wilson, *CVu* 23.6, January 2012

[6] Anatomy of a CLI Program written in C++, Matthew Wilson, *CVu* 27.4, September 2015.

[7] Building C & C++ Programs with the libCLImate Mini-framework , Matthew Wilson, *CVu* 27.5, November 2015

Assembly Club

Ian Bruntlett compares dialects of assembly code.

The first rule of Assembly Club is that no-one writes assembly. This article is not intended to teach anyone assembly language, but to help them on that journey. For these particular adventures, I used Ubuntu 18.04.2 LTS (64 bit, x86_64) on a refurbished ThinkPad.

Here are the packages I installed:

- emacs – the one true text editor
- make – for building executables
- nasm – a nice assembler
- yasm – an even nicer assembler
- gas – the assembler we have to put up with because it is available everywhere
- gdb – the GNU debugger.

When it comes to assembler, there are two major dialects – Intel and AT&T. Intel syntax is supported by the bulk of the tutorials and the GNU tools default to AT&T syntax. However, they can be persuaded to accept Intel syntax (experience indicates it is a bit of a compromise).

The other reasons why programmers prefer Intel syntax over AT&T syntax can be found from the manual:

- AT&T immediate operands are preceded by **$**; Intel immediate operands are undelimited (Intel **push 4** is AT&T **pushl $4**). AT&T register operands are preceded by **%**; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by *****; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel **add eax, 4** is **addl $4, %eax**. The source, dest convention is maintained for compatibility with previous Unix assemblers. Note that instructions with more than one source operand, such as the enter instruction, do not have reversed order.
- In AT&T syntax, the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of **b**, **w**, **l** and **q** specify byte (8-bit), **word** (16-bit), **long** (32-bit) and **quadruple word** (64-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (not the instruction mnemonics) with **byte ptr**, **word ptr**, **dword ptr** and **qword ptr**. Thus, Intel **mov al, byte ptr foo** is **movb foo, %al** in AT&T syntax.
- Immediate form long jumps and calls are **lcall**/**ljmp** **$section, $offset** in AT&T syntax; the Intel syntax is **call**/**jmp far section:offset**. Also, the **far** return instruction is **lret $stack-adjust** in AT&T syntax; Intel syntax is **ret far stack-adjust**.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

For me, I have a basic rule of thumb to handle the differences between Intel code and AT&T code. For AT&T syntax code, think of the comma between operands as 'to' and for Intel syntax code think of the comma as 'equals'.

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.

How did I get here?

I learned assembly language on the Sinclair QL and Intel x86 processors. I am not an expert these days. I got to grips with Linux by reading *How Linux Works* [1] and *The Linux Command Line* [2]. I am currently reading two books and referring to online resources. The books are *Introduction to 64 bit Assembly Programming with Linux and OSX* [3] and *Low-Level Programming* [4]. As I progress, I am assembling a scrap book consisting of my findings.

Versions of Hello World

- Listing 1 shows Hello World written for NASM/YASM

  This program was built in stages. I used YASM to convert the assembler to an object file and the GNU linker (ld) to create an executable file. The parameters for YASM tell it to format its output as an ELF (Executable and Linking Format) file, with debug records in DWARF2 format (Debugging With Attributed Record Formats). The GNU linker takes the `hello.o` object file and creates an output executable called **program**.

  ```
  yasm -f elf64 -g dwarf2 hello.asm
  ld -o program hello.o
  ```

- Listing 2 shows Hello World written for the GNU assembler. It only runs on 64-bit Linux. I edited the original [5] to calculate string length when assembled.

  I will explain how it gets built later on, using a makefile for GNU Make.

- Listing 3 shows Hello World, written in part-Intel part-AT&T syntax. This is so you can use the GNU assembler whilst almost writing your code in Intel format. A kind of poorly documented poor relation to the previous examples.

  It is based on the 'hello world' program from *Introduction to 64 bit Assembly Programming for Linux and OSX* [3].

Listing 1

```
%include "syscalls.inc"
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
  ; 1 system call number should be stored in rax
    mov     rax, __NR_write
  ; argument #1 in rdi: where to write (descriptor)?
    mov     rdi, 1
  ; argument #2 in rsi: where does the string start?
    mov     rsi, message
  ; argument #3 in rdx: how many bytes to write?
    mov     rdx, 14
  ; this instruction invokes a system call
    syscall
quit:
  mov rax, __NR_exit ; 60 exit
  mov rdi, 0         ; exit code
  syscall
```

Listing 2

```
#-----------------------------------------------------
# Writes "Hello, World" to the console.
# To assemble and run:
# gcc -c hello.s && ld hello.o && ./a.out        or
# gcc -nostdlib hello.s && ./a.out               or
# as -a=hello.lis --gstabs -o hello.o hello.s
# ld -o hello hello.o
#-----------------------------------------------------
    .include "syscalls-att.inc"

    .global _start
    .text
_start:
    # write(1, message, 13)
    mov $__NR_write, %rax  # system call code
    mov $1, %rdi           # file handle 1 is stdout
    mov $message, %rsi     # address of string to output
    mov $message_len, %rdx # number of bytes
    syscall
      # invoke operating system to do the write
    mov $__NR_exit, %rax # system call code
    xor %rdi, %rdi       # we want return code 0
    syscall              # invoke operating system to exit

    .data
message:
   .ascii "Hello, world\n"
   .equ   message_len, . - message
```

- Listing 4 shows the Makefile for the previous examples. It is from the makefile for [5] and [6].

  It actually does a bit more than that. Typing **`make hello-intel`** makes the Intel/AT&T variant, **`make hello`** makes the AT&T variant.

Debugging – gdb

I am no expert on gdb. I have spent a lot of energy just getting things to build and run properly. However, I've downloaded a copy of the *GDB Quick Reference* and pasted it into my assembly scrap book. You start the debugger with **`gdb executable-name`**. If gdb reports that it is 'reading symbols', you have managed to create an executable with debug symbols available. You can also check this by typing **`file executable-name`** at a shell prompt. Once in the debugger, there are a whole load of commands available (see the available online documentation). Once in the debugger, I tend to set things up with these commands:

```
break _start
start
layout src
```

Listing 3

```
  .intel_syntax
  .global _start  # was global start

  .data           # was section .rodata
msg: .ascii  "Hello, world!\n"
     .equ msglen, . - msg

  .text            # was section.text
_start:
  mov %rax, 1      #; write(
  mov %rdi, 1      #; STDOUT_FILENO,
  lea %rsi, msg    #; "Hello, world!\n",
  mov %rdx, msglen #; sizeof("Hello, world!\n")
  syscall          #; );
  mov %rax, 60     #; exit(
  mov %rdi, 0      #; EXIT_SUCCESS
  syscall          #; );
```

and then I use either the **`step`** command or the **`next`** commands to trace through the code.

The future

There is a free book available online – *Intel 64-bit Assembly Language Programming with Ubuntu* [7] – that I will be working through. And I will be using Google and accu-general for help as well. ■

Thank you

I would like to thank Tom Hughes, Bill Somerville, Jonathan Wakely and Ahtu Truu for their patience and help on accu-general.

References

[1] Ward, Brian (2014) *How Linux Works: What Every Superuser Should Know* (2nd ed.), No Starch Press, ISBN-13: 978-1593275679

[2] Shotts, William E. Jr. (2019) *The Linux Command Line: A Complete Introduction* (2nd ed.), No Starch Press, ISBN-13: 978-1593279523

[3] Seyfarth, Ray (2014) *Introduction to 64 Bit Assembly Programming for Linux and OS X* (3rd ed.), CreateSpace Independent Publishing Platform, ISBN-13: 978-1484921906

[4] Zhirkov, Igor (2017) *Low-Level Programming: C, Assembly, and Program Execution on Intel 64 Architecture*, Apress, ISBN-13: 978-1484224021

[5] The source of the example, and also a learning resource: http://cs.lmu.edu/~ray/notes/gasexamples/

[6] https://www.devdungeon.com/content/how-mix-c-and-assembly

[7] *x86-64 Assembly Language Programming with Ubuntu* by Ed Jorgensen (2019), http://www.egr.unlv.edu/~ed/assembly64.pdf

Other resources used when learning assembly

- ABI for x64 architecture: http://refspecs.linuxbase.org/elf/index.html
- Assembly language manuals: https://software.intel.com/en-us/articles/intel-sdm
- 'Bluff your way in x64 assembler' by Roger Orr from *ACCU Conference 2017*, available on YouTube
- 'Enough x86 assembly to be dangerous' by Charles Bailey from *CPPCON 2017*, available on YouTube
- GNU gdb manual: https://www.gnu.org/software/gdb/documentation/
- GNU toolchain manuals (make, as, ld): https://www.gnu.org/manual/manual.html
- Introduction to Assembly: https://software.intel.com/en-us/articles/introduction-to-x64-assembly
- YASM manual: http://yasm.tortall.net/

Listing 4

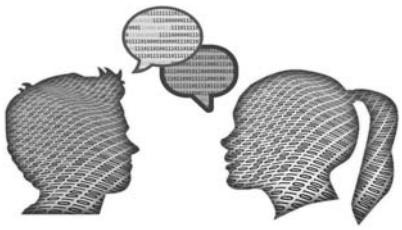
```
ASFLAGS= -a=$*.lis --gstabs

hello-intel : hello-intel.o
  ld -o hello-intel hello-intel.o
fib : fib.s
  gcc -ggdb -no-pie -o fib fib.s
hola : hola.s
  gcc -ggdb -no-pie -o hola hola.s
hello : hello.o
  ld -o hello hello.o
all : hello hola fib

clean:
  rm -f hello *.o *.lis
  rm -f printf hola fib hello-intel
```

Code Critique Competition 117

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 trying to extract the count of letters in a file and while the program itself seems to work OK, I'm not getting the debugging output I expected. I was expecting to see a debug line of each letter being checked during the final loop, but I only get some of the letters shown.

For example, using `sample.txt` that consists of "This is a some sample letter input" I get:

```
letter sample.txt
Seen T
Seen h
Seen n
Seen o
Seen r
Seen u
Commonest letter: e (4)
Rarest letter: T (1)
```

Why are only *some* of the letters being shown as **Seen**?

Can you solve the problem – and then give some further advice?

Listing 1 contains `logging.h` and Listing 2 is `letter.cpp`.

Listing 1

```
#include <iostream>
namespace
{
  class null_stream_buf
  : public std::streambuf
  {
  } nullbuf;
  std::ostream null(&nullbuf);
}
#ifdef NDEBUG
#define WRITE_IF(X) if (false) null
#else
#define WRITE_IF(X) ((X == true) ? std::cout
  : null)
#endif
```

Listing 2

```
#include "logging.h"
#include <fstream>
#include <map>

// get the most and least popular chars
// in a file
int main(int argc, char **argv)
{
  WRITE_IF(argc < 2)
    << "No arguments provided";
  if (argc > 1)
  {
    std::ifstream ifs(argv[1]);
    std::map<char, int> letters;
    char ch;
    while (ifs >> ch)
      ++letters[ch];

    std::pair<char, int> most =
      *letters.begin();
    std::pair<char, int> least =
      *letters.begin();
    for (auto pair : letters)
    {
      WRITE_IF(pair.second) << "Seen "
        << pair.first << std::endl;
      if (pair.second > most.second)
      {
        most = pair;
      }
      else if (pair.second < least.second)
      {
        least = pair;
      }
    }
    std::cout << "Commonest letter: "
      << most.first << " ("
      << most.second << ")\n";
    if (most != least)
      std::cout << "Rarest letter: "
        << least.first << " ("
        << least.second << ")\n";
  }
}
```

Critique

Hans Vredeveld <accu@closingbrace.nl>

I can see a couple of issues with this code, one of them resulting in only part of the letters being shown as **Seen**. Let's go through the code starting with `logging.h`.

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

The first thing we encounter is a class **null_stream_buf** and variables **nullbuf** and **null** that are defined in an unnamed namespace. The idea behind an unnamed namespace is that you can define things in a translation unit for use in that translation unit only without having to think about whether a name is already used in another translation unit. The compiler will create unique names for them in each translation unit, so there is no ODR violation. The result of this is that we will have as many instances of **null** and **nullbuf** in our binary as there are translation units. A waste of space in the binary.

If **null_stream_buf** were a buffer for some device (or file), the result would be a messy output. Messages written to **null** in different translation units would end up in different buffers and only when the underlying

library code decides to write the buffer to the device would the data be written. With different, independent, buffers, the order the data is written will appear random and the data will typically contain partial messages. This is similar to when two console applications write simultaneously to the console (or a single log file).

To get the expected result of the messages appearing in the output in the order they were written to the stream, we have to get rid of the unnamed namespace. The class **null_stream_buf** has to be defined in the global namespace (or in a named namespace), and the variables **nullbuf** and **null** have to be declared **extern** in the global namespace (or in a named namespace). Also, **nullbuf** and **null** each have to be defined in just one implementation file.

Now we have only one output stream **null** and one buffer **nullbuf** in our application, but there is still an issue with **null_steam_buf**. It doesn't override any of the virtual functions of **std::streambuf**. In particular, it doesn't override **int_type overflow(int_type ch)** in **std::streambuf**. The default implementation in **std::streambuf** always returns end-of-file. As a result, as soon as a single character is written to **null**, the fail-bit and bad-bit will be set. Basically, **null** will tell us that it's full and can't accept anything any more, but we happily continue to stream data to it. For a null stream, this is very strange. We expect it to always accept data and just discard it. To get this behaviour, we have to implement a public override for the overflow member function:

```
int_type overflow(int_type ch) { return ch; }
```

As an aside, note that, while the code is legal C++, clang++ (version 5.0.1) with all warnings enabled warns that **nullbuf** and **null** both require global and exit-time destructors. The warnings tell us that we may have a problem when there are multiple global variables in different translation units. We could get rid of these warnings by removing the global variables.

Moving to the next lines in `logging.h`, we come to the **WRITE_IF** macro. When **NDEBUG** is defined, nothing is logged because of the **if (false)**. It doesn't matter what **ostream** object follows. It could be **std::cout** instead of **null**.

When **NDEBUG** is not defined, compiling with g++ -Wall gives us a warning on **WRITE_IF(argc < 2)**, suggesting we put parentheses around the argument. The parentheses can best be put around **X** in the definition of **WRITE_IF**. The real problem with this macro can be seen in its other use in the application, **WRITE_IF(pair.second)**. After macro substitution, we have the comparison **(pair.second == true)**. However, **pair.second** is of type **int** and that means that we do an integer comparison and the **bool** constant **true** gets promoted to the **int** 1. The result is that we only print the characters that occur only once in the input file. What we need is a conversion to **bool** of **pair.second**, which we get with

```
#define WRITE_IF(X) ((X) ? std::cout : null)
```

With this modification, we will stream to **std::cout** when **X** evaluates to **true** and to **null** when it evaluates to **false**. But, why stream to a null output stream that discards everything that is streamed to it at all? When we change the definition to

```
#define WRITE_IF(X) if (X) std::cout
```

we only send data to a stream when we want to see it. If we now change the definition in the case that **NDEBUG** is defined to

```
#define WRITE_IF(X) if (false) std::cout
```

we can remove the global variables **null** and **nullbuf**, and the class **null_stream_buf**.

Now that we have **WRITE_IF**'s definition right, let's have a quick look at the statements where it is used in `letter.cpp`. The first time it is used, some text is output without a newline at the end. The second time, the newline is output through **std::endl**. While I prefer that the logging framework takes care of outputting the newline at the end, **WRITE_IF** is such a simple logging macro that this is not realistic. In this case it is up to the user to end each statement with outputting a **'\n'**, but not through **std::endl**, as Chris Sharpe described so well in *Overload* 149.

Let's continue with `letter.cpp` and in particular the **while**-statement that reads the file. I keep on wondering whether this **while**-statement does what the code's author intended. The condition of the **while**-statement, **ifs >> ch**, uses formatted input to read a character from the file. The effect of this is that it will skip over whitespace, but not over e.g. punctuation. Do we really want to ignore spaces, but count commas and full stops? As the map containing the data is called letters, I suspect that punctuation should also be ignored. Then an extra check with **std::isalpha** is in place:

```
while (ifs >> ch)
  if (isalpha(ch)) ++letters[ch];
```

Next is the **for**-loop. I don't like the use of **pair** as a variable name, as it can cause confusion with the well-known **std::pair**. My first suggestion would be to name it **letter**, but whether that is a good name is also open for debate. Instead we can replace the entire **for**-loop with a call to **std::minmax_element**:

```
auto [least, most] =
 std::minmax_element(letters.begin(),
   letters.end(),
   [](const auto& a, const auto& b) {
     return a.second < b.second;
   });
```

For this to work, we need to **#include <algorithm>**. With this change we lose the debug output of which letter we have seen, but in my opinion it has also lost its meaning.

Finally a note about the includes. The **std::cout** in `letter.cpp` is only made known through the include of `logging.h`. I strongly believe that any identifier used in a file should be declared in the file itself or brought in by an include in the file itself or, for an implementation file, its associated header file. This means that `letter.cpp` should **#include <iostream>** directly. For the same reason I would prefer to **#include <utility>** for **std::pair** (not needed any more when using **std::minmax_element**) in `letter.cpp`, and to **#include <ostream>** and **#include <streambuf>** in `logging.h` for **std::ostream** and **std::streambuf** (not needed any more after the rewrite of **WRITE_IF**).

James Holland <James.Holland@babcockinternational.com>

There are two features to the software than I thought could be the cause of the problem and, therefore, worth investigating. The first is the **null_stream_buff** class; the second is the macro **WRITE_IF**.

Although **null_stream_buff** may be a somewhat novel mechanism, it appeared to be working as desired, namely to produce no output when called. My attention then turned to the macro. The use of all but the simplest macros is discouraged as they can lead to some very obscure bugs that can be very difficult to diagnose. I was hopeful that the problem with the software debug output had something to do with the use of such macros.

I decided that the best way to proceed was to discover what the source code looked like after being preprocessed. This was accomplished by using the **-E** compiler switch and observing the resulting output. The statement of interest is the **WRITE_IF** macro within the **for**-loop of **main()** and is shown below.

```
WRITE_IF(pair.second) << "Seen "
  << pair.first << std::endl;
```

After preprocessing, the code the compiler sees is as follow.

```
((pair.second == true) ? std::cout:null)
  << "Seen " << pair.first << std::endl;
```

It can now be seen that whether **std::cout** or **null** is called depends on the comparison of **pair.second** with **true**. The type of **pair.second** is **int** and represents the number of occurrences of a particular letter within the text file. The type of **true** is **bool**. The problem now boils down to how the C++ compares for equality an **int** and a **bool**. From observing the program output, it can be seen that debug information is produced only when the value of **pair.second** is 1. This suggests that, in effect, the boolean value of **true** is converted to an **int** of value 1. The

comparison, therefore, consists of comparing the value of `pair.second` with 1. Output will, therefore, occur only when `pair.second` is equal to 1. This is consistent with observations.

The above analysis suggests ways of correcting the problem. One way is to convert `pair.second` to a boolean value with a `static_cast` before the comparison is made. A value 0 will be converted to `false` and any other value will be converted to `true`. Now, `false == true` has a value of `false` and `null` will be called; `true == true` has a value of `true` and `std::cout` will be called. This is as required and is borne out in practice by compiling and running the modified code shown below.

```
WRITE_IF(static_cast<bool>(pair.second))
  << "Seen " << pair.first << std::endl;
```

The comparison of a boolean value with `true` is a redundant operation. This suggests a simpler way of correcting the program. Instead of casting `pair.second` to `bool`, the macro defined in `logging.h` can be changed from

```
WRITE_IF(X)((X == true) ? std::cout:null)
```

to

```
WRITE_IF(X)((X) ? std::cout:null).
```

With this arrangement `pair.second` (an `int`) will be inherently converted to `bool` with a value `true` if `pair.second` is anything other than 0. This is as required and produces the required result.

Taking a slightly wider view of the program, it would seem preferable to modify the `if` statement that warns that `pair.second` is zero. The modification would add an `else` clause that prints the letter and the number seen. This would remove the need for the macro.

It is always worth considering whether a 'hand-rolled' loop can be replaced with an existing algorithm from the standard library. In the code presented, a `for`-loop scans through a `map` to determine the minimum and maximum of `map`'s values. There is a standard library algorithm named `minmax_element` that sounds as if it could to the same job as the `for`-loop. With the use of a lambda, as shown below, it can.

```
const auto [least, most] =
  std::minmax_element(letters.cbegin(),
    letters.cend(),
    [](const auto & a, const auto & b) {
      return a.second < b.second;
    });
```

It should be noted that if more than one minimum or maximum element exists, `minmax_element()` returns the first minimum and the last maximum element. The elements are stored alphabetically within letters and therefore, in this case, `minmax_element()` returns `T` as the rarest and `s` as the commonest. This is different from the code presented which returns the first commonest, namely `e`. If the software is to find the first commonest, two other algorithms (both in the standard library) can be used in place of `minmax_element()` (with a slight loss in efficiency) namely `min_element()` and `max_element()`. I have assumed the first commonest letter is to be returned as in the original code resulting in my version of the program shown below.

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <map>
int main(int argc, char **argv)
{
  if (argc < 2)
  {
    std::cout << "No arguments provided";
  }
  else if (argc > 1)
  {
    std::fstream ifs(argv[1]);
    std::map<char, int> letters;
    char ch;
    while (ifs >> ch)
      ++letters[ch];
    const auto compare_function =
      [](const auto & a, const auto & b) {
        return a.second < b.second;
      };
    const auto least =
      std::min_element(letters.cbegin(),
        letters.cend(), compare_function);
    const auto most =
      std::max_element(letters.cbegin(),
        letters.cend(), compare_function);
    if (most != letters.cend())
    {
      std::cout << "Commonest letter: " <<
        most->first << " (" << most->second <<
        ")\n";
      if (most != least)
      {
       std::cout << "Rarest letter: " <<
        least->first << " (" << least->second
        << ")\n";
      }
    }
  }
}
```

I have also added some protection against the input file being empty.

Balog Pál <pasa@lib.hu>

Eh, stream-based logging. I hope this is one of the last specimens, we finally get a sensible text formatting library in C++ standard and can leave this monster behind along with all the related quirks. But that is besides the point, let's read the code.

- We have a `logging.h` that looks like a header-only library designed for this specific purpose
- namespace `{}` in header: this is normally frowned upon, but fits with the purpose. We can live with this much bloat
- I'm puzzled on the empty subclassing. On a review, I'd require a good explanation on why we have that `null_stream_buf` class instead of just using `std::streambuf`. If it has some ADL-related reason to exist, that must appear in comments
- 3 names are used up for no good reason, internal purpose only. Especially `null` stands out. A snap-in like this should use only non-clashing names beyond its public interface
- We got to the main feature, that is a macro. I postpone discussion in the idea.
- The name `WRITE_IF` looks suboptimal too
- The macro is conditional on `NDEBUG`. That is a fishy choice, as `NDEBUG` controls assert, and this feature is not related.
- The first version could be just `null`, the `if(false)` part serves to save runtime processing of the `<<`s that follow. The condition is completely eliminated
- The second version has a `== true` comparison that is a major blunder – later we will see that it the actual source of misbehavior
- The main code starts in weird way, we have a `WRITE_IF` followed by another check of the same condition – carefully phrased differently. The normal way would emit message on the `else`. And not even with this conditional facility.
- I accept the reading of the file into the map though it could be arranged with some algorithm without a loop.
- The stream-related error handling is missing, but on problem we just result an empty map that should be okay.
- We run ahead dereferencing `*begin` without first checking for empty map that is a no-no.

- A manual search for element is presented instead of using `std::minmax_element` that would just do the job.

Certainly the last would kill our chance to use `WRITE_IF` and notice the misbehavior too... not sure to count that as pro or a con. The way we use it in the loop, we pass the `int` with the count. Honestly it beats me why we do that, as the method of our creating the map ensures it will not be zero.

The `int` is fine as a boolean expression and would work as intended if the macro just used it that way, just having `(X)`. With the `== true` added, the promotions work 'upwards', instead of forcing the count to a `bool` and compare with `true`, it stays the `int` and `true` is promoted to integer 1. Then `==` will pass only letters with exactly one instance in the input. Which matches the behavior we see.

Beyond that the loop looks correct though massively wasting resources: for has `auto` instead of `const&` making copy of every entry in the map and then making copies into `most` and `least`. With a `pair<char, int>` it might not be noticeable, but should be avoided. And as usual, using the stock algorithm would prevent this kind of mess-up and many others.

Finally, the deferred discussion on `WRITE_IF`. The design goes half-way in many points with respect to compile and runtime. I'd prefer an all-or-nothing approach. I'd rather work with a facility like

```
TRACE_IF( bool_cond, message_string );
```

that, if arranged with a macro can compile to nothing. Getting rid of both the condition and the message instructions. Or if we prefer to always have the code, can be a simple function. No switcheroo with streams and whatnot.

The 'weak' point is the string part, that in the presented version is assembled with the `<<` method. That some people still consider a good idea, while others not so much. My practice showed that even a `Sprintf()` function that uses the traditional `sprintf` interface with all the potentially related problems (I think we can ask a warning on all of them), and returns the favorite flavor of string is much better. But there are modern ways that combine the benefits of type safety and formatting with extras. One of them is supposedly getting into the standard too.

Peter Sommerlad <peter.sommerlad@hsr.ch>

I will first comment the code directly line by line and then provide a more idiomatic solution.

`logging.h`

This header file misses an include guard. If this file is included in the same translation unit twice, the global variables with internal linkage `nullbuf` and `null` will be doubly-defined and thus the code will no longer compile.

```
#include <iostream>
```

This include is not recommended, the code in this file requires the definitions of `std::streambuf` and `std::ostream`, both are available through the `<ostream>` header. However, the macro will require `std::cout` in case of debug-level compilation. But the macro definition will not require the definition of this global variable, but the macro expansion eventually. I teach my students to only include `<iostream>` in the file where the `main()` function is defined and only use the global variables `std::cin` and `std::cout` within the `main` function and write all code that does I/O taking a parameter of `std::istream&` or `std::ostream&` respectively. That principle allows writing unit tests for these functions, which are almost impossible to formulate, when the global stream variables are used directly.

`namespace`

Defining an anonymous namespace in a header file is not recommended, because wherever that header file is included all definitions exist with internal linkage. If done within the implementation file, I could live with that.

```
{
  class null_stream_buf
  : public std::streambuf
  {
  } nullbuf;
  std::ostream null(&nullbuf);
```

This is an interesting attempt to introduce the null-object pattern for `ostream`. However, using global variables to achieve that might not be the best way. Especially since formatting operations still will be performed, just no output actually happens.

For modern code formatting, I would always use braced-initialization for initializing variables, this removes the mental gymnastics to ensure that a variable is really initialized, since the default for trivial types without braces might mean uninitialized (for automatic storage duration variables), using uninitialized variables can lead to undefined behavior. While we have global variables here, it is still a practice recommended.

On the other side, there is some problem in the macro logic below.

```
}
#ifdef NDEBUG
#define WRITE_IF(X) if (false) null
```

defining a macro taking a parameter `(X)` and not using it can be strange. Also having a statement that is incomplete and not using braces for a block is very brittle, especially when surrounding code contains other conditional statements and a following `else` might not be attached to the `if` in the surrounding code but to the `if` from the macro.

On the other hand, having that `if` statement should guarantee that the output will actually never be generated, because the code path is not taken and the optimizer will eliminate that code.

```
#else
#define WRITE_IF(X) ((X == true) ? std::cout
: null)
```

Here is the bug that leads to only outputting letters with the count 1 (that is the integral conversion value of `true`). Whenever I see code that compares a (potential `bool`) value with either `true` or `false` that code is unprofessional (regardless of the language).

If the macro parameter is meant to be a `bool` value, then it should read `(X)?..:..`

Note that a macro parameter should always be in parenthesis when expanded to avoid passing 'interesting' strings that, when expanded, change the syntax of the expression or statement.

Again, we have a partial expression of type `ostream&`

If we want to stick with a macro for the debugging, I would suggest to pass all the output to be logged as parameter and not to introduce a conditional even in the debug case.

```
#endif

--- letter.cpp ---
#include "logging.h"
```

It is good practice to include the 'own' header first, because that guarantees that the own header is self-contained.

```
#include <fstream>
#include <map>

// get the most and least popular chars in a file
int main(int argc, char **argv)
{
```

If we follow the unix pipes and filters practice, if no file is given standard input (`std::cin`) should be processed. This will make the program logic not more complex, but will actually provide a more useful program.

A big problem of this program is that almost all code resides within the `main` function and is thus not unit-testable. BAD! For very simple example programs that are obviously correct, that might be OK, but as we see from the bugs, this program is not simple enough.

```
WRITE_IF(argc < 2)
  << "No arguments provided";
if (argc > 1)
{
```

There is no check that the file named `argv[1]` actually exists. From C++17 on we have `std::filesystem` that allows us to have such a sanity check.

This can still cause a race condition when a file is deleted between the check and the actually opening, but it usually allows providing better error messages. A side effect is, that on filesystems with interesting character encodings for file names the string represented by `argv[1]` can actually be converted into `std::path` to provide platform independent (unicode!).

```
std::ifstream ifs(argv[1]);
```

no error checking, but OK, just no input here.

```
std::map<char, int> letters;
char ch;
while (ifs >> ch)
  ++letters[ch];
```

This is quite idiomatic and automatically skips whitespace by using `operator>>`.

However, I would extract filling the map into a separate function, that returns the map by value. Such a function can be independently tested with unit tests.

When there is no input, and thus the map is empty, the following code dereferencing the `begin()` iterator of the map is undefined behavior.

It might be better to use iterators instead of the pair values for keeping track of the found elements. But using iterators then requires the check, if the iterators are valid (for example not equal to the map's `end()` iterator.

```
std::pair<char, int> most =
  *letters.begin();
std::pair<char, int> least =
  *letters.begin();
```

Later on, writing your own loop for a perfectly ready-made linear search algorithm is a code smell and it can be wrong. The algorithm in the standard library to use here would be `minmax_element` taking a comparison function object (a lambda) that compares the counters in the map's `pair::second` member.

```
for (auto pair : letters)
{
  WRITE_IF(pair.second) << "Seen "
    << pair.first << std::endl;
```

Here the debug macro is used and passing the actual count that is then compared with the value of `true`, results in only those letters to be printed that have a count of one. Using `std::endl` instead of `'\n'` is a smell, since in some cases it will result in unneeded operating system interaction, because of flushing the buffer prematurely.

Also test output is bad for a filter program in a pipeline, since that extra output will taint the processing of potentially later filters.

```
if (pair.second > most.second)
  {
    most = pair;
  }
  else if (pair.second < least.second)
  {
    least = pair;
  }
}
```

we will get rid of the whole loop above, so that we do not need to have the debug output at all.

```
std::cout << "Commonest letter: "
  << most.first << " ("
  << most.second << ")\n";
if (most != least)
```

An `if` statement where the body lasts over several lines without braces is dangerous. if the single statement is split for whatever reason, the logic might be wrong. Also the

```
  std::cout << "Rarest letter: "
    << least.first << " ("
    << least.second << ")\n";
  }
}
```

In contrast to my usual practice, I provide a solution without accompanying unit tests, but I have refactored the code, so that unit tests might be added easily, by splitting the code besides `main` into a separate compilation unit that can be linked to test cases.

As we will see, `main()` contains (almost) no branching logic and thus can not be wrong. All the other functions do just one thing, that should be testable and they are not dependent on global variables.

I will address the logging macro separately, since I believe it is no longer needed.

```
#include <fstream>
#include <string>
#include <map>
#include <algorithm>
using charmap=std::map<char,int>;
charmap countChars(std::istream &in) {
  std::map<char, int> letters { };
  char ch { };
  while (in >> ch) {
    ++letters[ch];
  }
  return letters;
}
using mapiter=charmap::const_iterator;
void printresult(
  std::ostream &out,
  std::string text,
  mapiter res,
  mapiter noresult) {
  if (res != noresult) {
    out << text << res->first << " ("
      << res->second << ")\n";
  }
}
#include <iostream>
// get the most and least popular chars in a file
int main(int argc, char **argv) {
  std::ifstream ifs {
    argc > 1 ?
    std::ifstream { argv[1] }
    : std::ifstream { 0 } };
  auto const letters { countChars(ifs) };
  auto const least_most {
    minmax_element(letters.begin(),
      letters.end(),
      [](auto l, auto r) {
        return l.second < r.second;
      }) };
  printresult(std::cout,
    "Commonest letter: ",
    least_most.second, letters.end());
  printresult(std::cout, "Rarest letter: ",
    least_most.first, letters.end());
}
```

What needs to be explained?

I included `<iostream>` directly in front of `main`, to demonstrate that the other functions are independent of the global stream objects. While this is not very idiomatic, it was my test to make sure that I did not leave `std::cout` somewhere in the extracted code and thus make it untestable.

Let us start with `main()`. I extended the functionality to be able to operate the program as a filter. The first statement initializing `ifs` is using a trick. Since `iostream` objects in general are neither copy-able nor move-able it is tricky to provide a factory that either returns `std::cin` (or a reference to it) or otherwise a `std::ifstream` that opens the file given as an optional argument. File stream objects can be moved, so it is possible to initialize an `std::ifstream` variable from a temporary. Here I use the trick, that an `ifstream` can either open a file, given as a string or path argument or binds to an already open file descriptor. Since 0 is the file descriptor of the standard input of a program, I create an `ifstream` object

using that file descriptor when no file name is given. (I should have implemented a factory with my filesystem readability check first, but I refrained from it for brevity. This is left as an exercise to the student.)

The code reading the stream is extracted into a separate function and thus becomes testable.

Instead of searching for the letter with the lowest and highest count, I use the `minmax_element` algorithm available since C++11. To avoid having to spell `std::pair<char,int>` I use a generic lambda doing the comparison on the counts. Note that we get a slight deviation from the logic, because the 'commonest' (largest) letter count is the last one of potentially similar counts. The original algorithm behaved slightly different. Nevertheless, the logic is OK, because we do not have a specification (or test case) demonstrating what should happen if we have multiple 'commonest' letters.

Since the output needs to check for validity (the input might have been empty) and do so twice, I extracted the output code into the `printresult` function.

For the logging functionality, I refer to a stackoverflow entry, that demonstrates different approaches to achieve logging:
https://stackoverflow.com/questions/19415845/a-better-log-macro-using-template-metaprogramming

But my guideline, especially for beginners, when you need logging or interactive debugging to understand what your code does, your code is usually very badly structured. So, simplify it into smaller parts, write unit tests against these parts, or even better write the tests first, this will help you to grow your software according to your needs instead of guesswork and wasting time (see Pete Goodliffe's article in *CVu* that presented that problem).

That's it. I hope I did not forget anything crucial, because I was too lazy to write test cases. Shame on me.

Silas S. Brown <ssb22@cam.ac.uk>

The reason why only some of the letters are being shown as `Seen` is that your `WRITE_IF(X)` macro tests for `(X == true)` when `X` is an integer. The integer value of `true` is 1, so only letters that occurred exactly once (and no more than once) are being printed as `Seen`.

Instead of checking that `X` is equal to `true`, check that `X` is not equal to `false`, i.e. `(X != false)`.

But you really need to put extra parentheses around that `X`, i.e. `((X) != false)`, because the macro is expanded without regard to C++ syntax and you could break it if you ever wrote, for example, `WRITE_IF(A & B)` expecting to get the bitwise AND of `A` and `B`: you would instead get the bitwise AND of `A` and either a 0 or a 1 depending on whether `B` differs from `false`, since `!=` has a higher precedence than the bitwise `&` operator. Ouch. (This particular problem would have been flagged up by GCC if you had passed the `-Wall` (warnings all) parameter to the g++ command. I really don't understand why `-Wall` is not the default.)

In this particular case, we could side-step the need for extra parentheses by noting that `if ((X) != false)` is equivalent to `if (X)` (since C++, unlike languages such as Java, does let you put non-boolean types into an `if` expression, and automatically applies an `!= 0` when it sees a numeric type), but I still made the point about the extra parentheses because you'll need to know that if you do any more things with macros.

But ideally I don't want you to be doing any more with macros at all. To misquote Doc Brown from *Back To The Future*: "where we're going, we don't need macros." Try this:

```
static inline auto& WRITE_IF(bool X) {
#ifdef NDEBUG
  return null;
#else
  return X ? std::cout : null;
#endif
}
```

If your compiler is too old for `auto&` to work, then write `std::ostream&` instead, but do consider upgrading your compiler because `auto` is a really useful feature of modern C++ that saves us from having to figure out exactly what type something's going to be when the compiler can do that job.

(The usual convention is that a name written in all capitals implies a macro, so really we should be changing it to lower case now that it's a function. But I've left it as capitals for now so you can drop the above into the existing code without having to change anything else and it'll still work.)

Here are just some of the advantages of that function-based approach:

1. The compiler can check that our function makes sense, even before we try to use it. (Macros are not checked until they are used, and if there's a problem the error messages are usually harder to understand, that is if they're not missed altogether as in your case.)
2. We can specify a type for the parameter `X`. In this case, we say we want a `bool`, so anything else will be converted to a `bool`. If you had written `X==true` in *this* version, it would not have been a problem.
3. We don't have to worry about extra parentheses (and I haven't even mentioned multiple evaluations of the same variable).
4. It's easier to span multiple lines, and as the above shows, putting the `#ifdef NDEBUG` inside the function saves us from writing the first line twice (which is good, because writing something twice means there are two places to fix it if it turns out to be wrong, and if one of those is not even reached by the compiler due to an `#ifdef` then it could easily be missed).

In the old days, a counterpoint to all of this might have been 'but what about the overhead of having a function call'. But with modern optimising compilers, especially as we said 'static inline', I would be amazed if the actual machine code generated were any different from doing it with a macro, whether or not `NDEBUG` is defined.

Another improvement that is possible (now that we have a function) is to hide the `null` stream inside that function, by making it a static instead of a global. What I mean is to change the start of the function to:

```
static inline auto& WRITE_IF(bool X) {
  static std::ostream null(&nullbuf);
```

and remove the `std::ostream null(&nullbuf);` from your namespace block. This has the advantage of not creating a thing called `null` that's just 'floating around' your file's global namespace, which could be accidentally referred to in the wrong context. After all there's nothing in the name to reinforce the idea that it's a null stream, rather than a null something else. Yes, you could change the name, but I'd rather show you how to encapsulate it in the function so you no longer have to worry about a good global name.

Chris Trobridge <christrobridge@gmail.com>

Considering the output of the program:

- The program is only outputting letters that occur exactly once;
- The program counting capitals separately. There's no reason to count each case separately;
- There is more than one most common and least common (case insensitive) letter and it is misleading to just print the first one.

Considering `logging.h`...

Line 15:

```
#define WRITE_IF(X) ((X == true) ? std::cout
: null)
```

Comparison against `true` is tautology at best but dangerous as bool `true` is promoted to an integer value of 1, when compared against an `int`, while any non-zero integer converts to bool `true`.

I don't like an object being named as a (conditional) verb.

Considering `letter.cpp`…

Lines 10–11:

```
WRITE_IF(argc < 2)
<< "No arguments provided";
```

This is an error message and would typically be output unconditionally to **`std::cerr`**.

Line 18:

```
++letters[ch];
```

This is case sensitive and **`ch`** should be converted to lowercase first.

Line 26:

```
WRITE_IF(pair.second ) << "Seen "
```

As noted previously **`WRITE_IF`** compares its argument against **`true`**, which is promoted to 1 and hence only letters that have a frequency of one are displayed.

This may be fixed by removing the comparison in the definition of **`WRITE_IF`** above by replacing **`(X == true)`** with **`(X)`**.

Replacing the macro invocation with **`WRITE_IF(pair.second > 0)`** would also fix the issue and express the programmer's intention better.

The rest of the **`for`** loop looks for the most and least common letters. This most and least could be replaced with vectors of **`char`** to capture all the most and least popular letters.

Finally line 40:

```
if (most != least)
  std::cout << "Rarest letter: "
    << least.first << " ("
    << least.second << ")\n";
```

This is interesting in as much as it only prints out the rarest letter if it is not equal to the most common. Both the most common and least common letters are set to the first letter in the map and are only changed if there is at least one other letter more common/rarer than the initial letter. The rarest letter will be printed unless all the letters in the file appear with equal frequency.

Making the changes I have suggested I get the following output:

```
Seen a 2
Seen e 4
Seen h 1
Seen i 3
Seen l 2
Seen m 2
Seen n 1
Seen o 1
Seen p 2
Seen r 1
Seen s 4
Seen t 4
Seen u 1
Most common letters: e s t (4)
Rarest letters: h n o r u (1)
```

Amended `letter.cpp`:

```
#include "logging.h"
#include <fstream>
#include <map>
#include <vector>

// get the most and least popular chars in a file
int main(int argc, char **argv)
{
if (argc < 2)
  {
    // Require at least one argument
    std::cerr << "No arguments provided\n";
  }
if (argc > 1)
{
  std::ifstream ifs(argv[1]);
  std::map<char, int> letters;
  char ch;
  while (ifs >> ch)
    // Ignore case of letters when counting
    ++letters[tolower(ch)];
  // The most common letter must have a
  // frequency >= 0
  int mostFrequency = 0;
  std::vector<char> most;

  // The rarest letter must have a
  // frequency <= INT_MAX
  int leastFrequency = INT_MAX;
  std::vector<char> least;

  for (auto pair : letters)
  {
    // Diagnostics: display the frequency of
    // all letters seen in the file
    WRITE_IF(pair.second > 0) << "Seen "
      << pair.first << " " << pair.second
      << std::endl;

    if (pair.second > mostFrequency)
    {
      // This letter is more common than any
      // seen before so reset
      // the most common list to this letter
      mostFrequency = pair.second;
      most = { pair.first };
    }
    else if (pair.second == mostFrequency)
    {
      // This letter is as common as the most
      // common seen so far
      // so add to the most common list
      most.push_back(pair.first);
    }

    if (pair.second < leastFrequency)
    {
      // This letter is less common than any
      // seen before so reset the most rarest
      // list to this letter
      leastFrequency = pair.second;
      least = { pair.first };
    }
    else if (pair.second == leastFrequency)
    {
      // This letter is as rare as the rarest
      // seen so far so add to the rarest list
      least.push_back(pair.first);
    }
  }

  if (!most.empty())
  {
    // There is at least one letter so print
    // out the list of most common letters and
    // their frequency
    std::cout << "Most common letters: ";
    for (auto ch : most)
      std::cout << ch << " ";
    std::cout << "(" << mostFrequency
      <<  ")\n";
  }
```

```
    if (!least.empty() &&
      mostFrequency != leastFrequency)
    {
      // There is at least one letter that is not
      // the most common so print out the list of
      // rarest letters and their frequency
      std::cout << "Rarest letters: ";
      for (auto ch : least)
        std::cout << ch << " ";
      std::cout << "(" << leastFrequency
        << ")\n";
    }
  }
}
```

Commentary

I think the six entrants between them covered pretty well all the ground, so there seems little left for me to add!

One thing that wasn't explicitly discussed was the performance downside of using a map where the input is a small discrete set of values. I suggest an **`array`** indexed by the (unsigned) **`char`** value might be simpler and faster or, in a more general case, perhaps an **`unordered_map`**.

Tying the logging to **`NDEBUG`**, as Pál suggested, may be a mistake: especially for the error message written if the argument count is wrong. While some of the logging frameworks out there do this, others use other ways to enable/disable the logging.

Note that the empty class **`null_stream_buf`** is required since the default constructor of **`std::streambuf`** is protected.

It might also be worth a little more explanation of the problem with an anonymous namespace in a header: everything inside the namespace will have a **unique** name for each translation unit that includes the header. The result can be that multiple copies of entities from the header may end up in the resultant binary, differing only in their namespace.

If this sort of thing is needed, in C++17 you can use a named namespace and **`inline`** variable declarations to ensure only one definition will be included in the built program.

While I like Peter's solution that uses **`std::cin`** if no filename is provided please note that his solution relies on a non-standard extension that allows construction of an **`ifstream`** from a file handle of 0.

Finally, in correspondence received that was not an actual *critique* as such, it was suggested you could write the below command in less time it takes to compile the C++ program:

```
cat sample.txt |fold -w1|grep -v ' ' \
 |sort|uniq -c|sort -n -k 1 \
 |gawk 'NR==1; END{print}'
```

The winner of CC 116

All the entrants correctly identified the bug in the **`WRITE_IF`** macro with the expression **`X == true`**. Explicitly checking against **`true`** or **`false`** is, as Chris said, a tautology at best. While some languages require this, C++ does not.

I liked Silas' approach of fixing the problem by replacing the macro with an **`inline`** function – in C++17 there are even more places where this may be possible.

A couple of people ensured the code was valid if the file failed to open, Peter also suggested checking the file exists using **`std::filesystem`**.

Most of the critiques addressed the 'C++' issues in the code; Hans noticed that the code incorrectly treated *punctuation* as letters and Chris recognised in addition to this that the code treated upper and lower case letters differently. (I suspect an *international* solution might need to deal with Unicode rather than the **`char`**-based code presented...).

While many of the critiques suggested using **`minmax_element`** to replace a hand-written loop (I was pleased to see this!) and some people noted that this might change *which* element was found where there was a duplication, only Chris suggested that this reflected a problem with the code itself in that it incorrectly failed to handle the case where multiple letters tied as most or least frequent.

After reading again through each critique and weighing the various slightly different approaches each took, in my opinion Chris gave the most helpful answers to the writer of the code and so I have awarded him the prize for this issue's critique. But thanks to *all* who took part; one of the things I appreciate is looking at the different ways that different programmers can view the same code, and the variety of possible approaches they suggest to improving it.

Code Critique 117

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

> I'm trying to do some simple statistics for some experimental data but I seem to have got an error: I don't *quite* get the results I am expecting for the standard deviation. I can't see what's wrong: I've checked and I don't get any compiler warnings. I ran a test with a simple data set as follows:
>
> ```
> echo 1 2 3 4 5 6 7 8 9 | statistics
> mean: 5, sd: 1.73205
> ```
>
> The mean is right but I think I should be getting a standard deviation of something like 2.16 for the set (without outliers this is [2,8].)

Can you solve the problem – and then give some further advice?

Listing 3 contains `statistics.h` and Listing 4 is `statistics.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 3

```
namespace statistics
{
  // get rid of the biggest and smallest
  template <typename T>
  void remove_outliers(T& v)
  {
    using namespace std;
    auto min = min_element(v.begin(), v.end());
    auto max = max_element(v.begin(), v.end());
    v.erase(remove_if(v.begin(), v.end(),
      [min, max](auto v) {
        return v == *min || v == *max;
      }),
      v.end());
  }
  template <typename T>
  auto get_sums(T v)
  {
    typename T::value_type sum{}, sumsq{};
    for (auto i : v)
    {
      sum += i;
      sumsq += i * i;
    }
    return std::make_pair(sum, sumsq);
  }
```

Challenges

Francis Glassborow revisits old challenges and sets a new one.

I am under a good deal of time pressure as I write this column. I got back from the ACCU Conference 2019. That was a great event and I was delighted by the number of first time attendees. It was also great to see that we are beginning to achieve more diversity in attendance and participation.

If you are not already familiar with #include<C++> please take a few minutes to read about it at www.includecpp.org. I think that it is important.

Do you know what the Pac-Man Rule is? If not, look it up on the internet using your search engine of choice. (Note that we should not call that 'googling' just as we no longer call using a vacuum cleaner 'hoovering'. We should avoid referring to a generic activity by deriving a verb from a proprietary name.) When you have found out what the Pac-Man Rule is, consider using it, not just at conferences but at all gatherings (weddings, funerals, etc.).

OK, so returning from the conference left me with work to do on the allotment (surprising how quickly the grass grows) and the house, where I'm in the middle of laying new wooden floors upstairs, requiring tedious moving of furniture, piles of books, etc. (The actual process of laying engineered wooden floors is pretty straightforward – well, it is if you have all the right tools, such as a saw table, spacers, etc.) But now, just five days later, I am off to Eastercon for the whole of the Bank Holiday weekend, Friday to Monday inclusive.

Then your editor emailed me that he was using this weekend to put together the issue of *C Vu* you are reading. Well, I know what it is like to be an editor and how precious contributions are. So here I am throwing together a column without the time to polish it.

Time to get started.

Challenge 5 revisited

Don't you just love it when a spam filter blocks an email that has valuable content? (As I wrote that I reflected on English abbreviations, concatenations etc. Try replacing 'don't' with 'do not' and you will see what I mean.)

Sadly, a spam filter somewhere prevented me from seeing an excellent entry from Silas Brown. Every time I get something from Silas, I am reminded that he was the youngest ever contributor to *C Vu*. Somewhere over the years ACCU has lost touch with the youthful enthusiasts among whom are the future contributors to the developer community. I think it is time for members to try to involve themselves in 'Code Club' not just by helping in a local school but by actively encouraging the young to tell us about what they are coding. (Steve, do you think we could set aside a regular column about Code Club with space for contributions from young coders?) [Ed. Yes, I think it's a great idea. If anyone has any suggestions for starting this, please get in touch!]

Well here is Silas' excellent response to Challenge 5.

From Silas Brown

Dear Francis,

I'm sorry to read in *C Vu* 30.6 that you received only one partial submission for Challenge 5 and the submitter was silent after you gave a further hint. I'm now suspecting that spam filters are working against us. I sent a submission (copy below) on 17th September, and I did not hear back from you. So if you received one submission and sent one hint, then either my submission was lost on its way to you, or, if mine was the one you received and responded to with a hint, then your hint was lost on its way back to me.

Neither possibility is very comfortable to consider, as I begin to wonder if the filters responsible for this are also blocking other submissions. I am

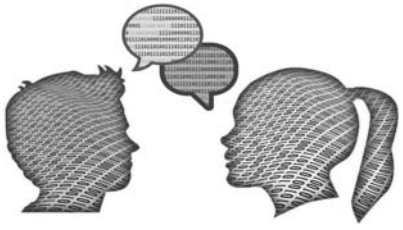
FRANCIS GLASSBOROW

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

Code Critique Competition 117 (continued)

Listing 3 (cont'd)

```
  template <typename T>
  auto calc(T v)
  {
    remove_outliers(v);
    auto sums = get_sums(v);
    auto n = v.size();
    double mean = sums.first / n;
    double sd = sqrt((sums.second -
      sums.first * sums.first / n) / n - 1);
    return std::make_pair(mean, sd);
  }
}
```



Listing 4

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <iterator>
#include <vector>
#include "statistics.h"
void read(std::vector<int>& v)
{
  using iter = std::istream_iterator<int>;
  std::copy(iter(std::cin), iter(),
    std::back_inserter(v));
}
int main()
{
  std::vector<int> v;
  read(v);
  auto result = statistics::calc(v);
  std::cout << "mean: " << result.first
    << ", sd: " << result.second << '\n';
}
```

therefore CC'ing this message to cvu@accu.org as if the worst comes to the worst, printing it may be the only way to reach you! Please check your filter settings to make sure you can receive submissions. It might also help if every episode of your Challenge series were to clearly mention where submissions should be sent. Many thanks.

[Francis: I think it is probably wise to copy all submissions to cvu@accu.org as well as to the columnist, in my case francis.glassborow@btinternet.com, so that there are at least two chances of beating the filters.]

Here is what I sent:

I am not sure we need 7 operators. If we choose 6: **= - / < ^ &** we can get what is (I think) all of the C++ operators that are capable of being overloaded in static functions, although I had to cheat a bit for 3 of them (I commented where I cheated in the code below), so if you want you can replace one of the three 'cheats' with its real operator to choose a 7th.

```
template<typename T> T operator+(T t1, T t2)
{ return t1 - (0 - t2); }
/* This was given in the challenge, so I assume
   T is numeric, not a string etc. */
template<typename T> T operator+(T t)
{ return t; }
template<typename T> T operator-(T t)
{ return 0 - t; }
template<typename T> T operator*(T t1, T t2)
{ return t1 / (1 / t2); }
template<typename T> T operator%(T t1, T t2)
/* cheat: this works only if T is an integer type */
{ return t1 - (t1 / t2) * t2;
template<typename T> T operator+=(T &t1, T t2)
{ t1 = t1 + t2; return t1; }
template<typename T> T operator-=(T &t1, T t2)
{ t1 = t1 - t2; return t1; }
template<typename T> T operator*=(T &t1, T t2)
{ t1 = t1 * t2; return t1; }
template<typename T> T operator/=(T &t1, T t2)
{ t1 = t1 / t2; return t1; }
template<typename T> T operator%=(T &t1, T t2)
{ t1 = t1 % t2; return t1; }
template<typename T> T operator++(T & t1)
/* prefix */
{ return (t1 = t1 + 1); }
template<typename T> T operator++(T & t1, int)
/* postfix */
{ T t0 = t1; t1 = t1 + 1; return t0; }
template<typename T> T operator--(T & t1)
{ return (t1 = t1 - 1); }
template<typename T> T operator--(T & t1, int)
{ T t0 = t1; t1 = t1 - 1; return t0; }
template<typename T> bool operator!(T t)
{ if(t) return false; else return true; }
template<typename T,typename U> bool
  operator&&(T t,U u)
/* cheat: this won't lazily-evaluate u */
 { if(t) { if(u) return true; } return false; }
template<typename T,typename U> bool
  operator||(T t,U u)
/* cheat: this won't lazily-evaluate u */
{ if(t) return true; if(u) return true;
  return false; }
template<typename T> bool operator>(T t1, T t2)
{ return t2 < t1; }
template<typename T> bool operator!=(T t1, T t2)
{ return t1 < t2 || t1 > t2; }
template<typename T> bool operator==(T t1, T t2)
{ return !(t1 == t2); }
template<typename T> bool operator<=(T t1, T t2)
{ return t1 < t2 || t1 == t2; }
template<typename T> bool operator>=(T t1, T t2)
{ return t1 > t2 || t1 == t2; }
template<typename T> bool operator~(T t)
{ return t ^ (T)-1; }
template<typename T> bool operator|(T t1, T t2)
{ return t1 ^ t2 ^ (t1 & t2); }
template<typename T> T operator&=(T &t1, T t2)
{ t1 = t1 & t2; return t1; }
template<typename T> T operator|=(T &t1, T t2)
{ t1 = t1 | t2; return t1; }
template<typename T> T operator^=(T &t1, T t2)
{ t1 = t1 ^ t2; return t1; }
template<typename T,typename U> T
  operator<<(T t, U u)
{ while(u--) t*=2; return t; }
template<typename T,typename U> T
  operator>>(T t, U u)
{ while(u--) t/=2; return t; }
template<typename T,typename U> T
  operator<<=(T & t, U u)
{ t = t << u; return t; }
template<typename T,typename U> T
  operator>>=(T & t, U u)
{ t = t >> u; return t; }
template<typename T,typename U> U
  operator,(T t, U u)
{ return u; }
```

I would have liked to add something like:

```
  template<typename T> T operator=(T &t1, T t2)
  { t1(t2); return t1; }
```

but this won't compile: an overloaded **=** must be a nonstatic member function of **T**, which means you can't just template it across all types like that. If the above were legal, it would work only for types where construction is equivalent to assignment, but I'm assuming that's the case for numeric types and it would buy us one more operator.

new and **delete** are operators too, but overriding them will require some serious low-level work. And overriding the **[]** operator must also be done as a nonstatic member function (for good reason: we cannot assume **t[u]** is always equivalent to ***(t+u)** when we're dealing with non-array containers.) And the **?:** operator cannot be overloaded so it shouldn't count.

Francis responds...

Thanks. You have surfaced a number of issues that I had not considered. The case of lazy evaluation is interesting because there is currently no way that a programmer can write functions that involve lazy evaluation. Indeed, it is worse than that because we do not (or has the language changed somewhere in the last few years?) even control the order of evaluation of arguments in a call to a function with more than one parameter.

C++ is relatively free of the core language doing things that cannot be done by libraries. The lazy evaluation case is notable because those operators can be overloaded but cannot emulate the built-in version.

Anyone like to highlight other places that the core language does things that cannot be emulated by libraries?

Challenge 6

From James Holland

For this challenge, I can do no better than to reproduce the work of the rector and mathematician Christian Zeller (1822–99) to calculate the day of the week from a given year, month and day of the month. By the time some output formatting has been added, the program is not all that short but it may be of some interest. The code for the Gregorian calendar follows.

```
#include <iostream>
#include <string>
std::string dow(int y, unsigned int m,
  unsigned int d)
{
  if (m == 1)
  {
    m = 13;
    --y;
  }
  else
  if (m == 2)
  {
    m = 14;
    --y;
  }
  const auto h = (d + (13 * (m + 1)) / 5 + y + y /
    4 - y / 100 + y / 400) % 7;
  std::string day;
  switch (h)
  {
  case 0:
    day = "Saturday";
    break;
  case 1:
    day = "Sunday";
    break;
  case 2:
    day = "Monday";
    break;
  case 3:
    day = "Tuesday";
    break;
  case 4:
    day = "Wednesday";
    break;
  case 5:
    day = "Thursday";
    break;
  case 6:
    day = "Friday";
    break;
  }
  return day;
}
int main()
{
  std::cout << dow(2019, 3, 27) << '\n';
}
```

Francis responds...

Thanks, James. We can quickly reduce the size of that function with a look-up table of days of the week (which, has the potential for being populated from an external source and so make it transportable to other languages.

The second problem is that it only works for the Gregorian calendar and needs substantial reworking for other calendars, in particular the Julian calendar.

Let me outline the solution of my pupil.

The first thing he did was to substitute much calculation with look-up tables. In C++ terms:

1. A 2-dimensional 2×12 array of month names, the first row consisted of the first 3 letters of each month in lower case. The second row were the numbers 1 through 12 as strings.
2. An array of weekday names, fully spelt with leading letter in uppercase, starting with Sunday. You will see that this works well for zero-based array indices as 01/01/01was a Monday.
3. A 2-dimensional 2×13 array of offsets from the first day of the year for first of each month (so the entries for January were 0, for February were 31, for March were 59 and 60. And from then onwards the second row were one more than for the first row. The 13th entry was either 365 or 366.

The next process was to acquire input and validate it. To avoid problems with different date formats, he explicitly asked for

1. The year
2. The month
3. The day of the month

The first of these simply had to be a positive integer (i.e. greater than 0).

The second was read as a **`string`** that was converted to lower case and if there were more than 2 **`char`** it was truncated to 3 **`char`**. Then the first row of the month array was searched for a match. The value of the index for a match was stored in an **`int`**. If there were less than 3 **`char`** the second row was searched and a match was similarly stored as an **`int`**.

If no match was found, the user would be informed that the month was invalid and prompted again. Up to three attempts were allowed before the program politely informed the user that it was unable to help.

The day of the month was checked to be in range for the relevant month (note that the third look-up table above can be used as *month*+1 - *month* gives the number of days in the current month). February 29 was provisionally accepted for any year that was a multiple of 4.

The Julian leap day calculation was next performed as (year- year%100)

At this point, the user was prompted as to whether they wished to use the Gregorian calendar. If the answer was yes, the leap day calculation was adjusted by adding year%400.

Final preparation was to revisit the case of February 29 and reject it if the year was a multiple of 100 (Julian) or a multiple of 100 but not 400 (Gregorian).

Now the weekday array was used with an index computed as:

```
(day+days_offset[month, leap] +
   number of leap days)%7
```

As I mentioned earlier, the use of look-up tables that could be populated from a configuration file made the final program independent of English names for months and weekdays. That was a particular telling point with examiners who appreciated this attempt at internationalisation (which the moderators had not).

My apologies that this is only an outline, but I am out of time and you all know how much time it takes to write good, correct code.

Challenge 7

How many actual characters from the source code character set are actually necessary to write C programs? C++ programs? As a little hint, I once came across a Cobol programmer who feared for his continued employment and started using only 'O', '0', 'I' and '1' for identifiers in his code. The theory being that they would need to keep employing him as he would be the only person who could maintain his code. As you might guess, it was a self-fulfilling strategy as his employers quickly removed him before he had actually written much.

Appeal

Do you have a problem that could serve as a challenge? Good challenges should require a degree of lateral thinking, not too much coding and reveal some aspect of programming that we easily miss (for example, the problem of lazy evaluation mentioned above). They do not have to be C or C++ based but unless it is Forth, Basic, Old Fortran or Snobol, my commentary on responses will be minimal. I once spent a lot of time trying to master Lisp but never succeeded to the extent that made me a Lisp programmer, just a procedural programmer expressing procedural code in Lisp (horrible ☺).

Bookcase

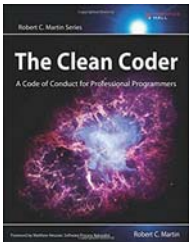
The latest roundup of book reviews.

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

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

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

The Clean Coder: A Code of Conduct for Professional Programmers



By Robert C. Martin, published by Prentice Hall (2011), ISBN: 978-0137081073

Reviewed by Ian Bruntlett

This book gave me a part crisis, part epiphany. It has given me a fresh perspective on my career and a look at the things I have got right or wrong Its subtitle, 'A Code of Conduct for Professional Programmers', is accurate but I have been showing it to friends who are office workers or engineers and just friends in general. One friend, Paul, a former Chief Pharmacist for an NHS trust, said that people in the NHS could benefit from this kind of guidance. It is structured in a way that is amenable to referring to individual chapters for specific guidance. The author started out as a programmer and then moved to consultancy and has a lot of useful tips for professional programmers.

Chapter 1, Professionalism, was a wake-up call for me. To quote the author, "Professionalism is a loaded term. Certainly it is a badge of honour and pride but is also a marker of responsibility and accountability." I've accepted responsibility and accountability in my career but this is the first time I've seen this advice written down on paper. It is the kind of information that I wish I had when I was at University.

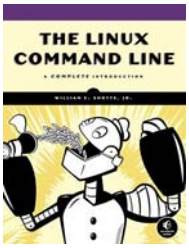
Chapters 2 ('Saying No') and 3 ('Saying Yes') are good ways to deal with Management by Objective. To quote: "Professionals speak truth to power. Professionals have the courage to say no to their managers." It covers how to deal with arguing your professional case within a company and how that can be done in a professional manner. The 'Saying Yes' chapter has an interesting guest article, 'A Language of Commitment'. Namely: 'Say. Mean. Do' – 'You say you'll do it. You mean it. You actually do it.'

Chapter 4, 'Coding', is a mini-book in itself and refers to another of the author's books: *Clean Code*. Chapter 5, 'Test-Driven Development', covers TDD well enough but the bibliography is a bit poor. Chapter 6, 'Practicing', introduces TDD concepts like Coding Dojos and the relevance to martial arts practices like Katas, Wasas and Randori and how practising is a necessary part of being a professional.

The rest of the book covers Acceptance Testing, Testing Strategies, Time Management, Estimation, Pressure, Collaboration, Teams and Projects, Mentoring, Apprenticeship and Craftsmanship and, finally, Tooling.

By now you should know if this book is of use to you. I believe it will be.

The Linux Command Line – A Complete Introduction (1st Edition)



By William E. Shott, published by No Start Press, Incorporated (2012), ISBN 978-1-59327-389-7

Reviewed by Ian Bruntlett

I am not an expert in this field. I use the command line regularly and have written a couple of shell scripts that are particularly important to me.

This book is available as a paid-for book and as a free to download PDF from http://linuxcommand.org/. Before I read this book, I read *How Linux Works*, to get a better idea about Linux.

Note from the book's author: "The second No Starch edition contains about 20 pages of additional material mostly having to do with bash version 4 and a number of additions inspired by reader feedback. The second No Starch edition is based on my 5th Internet edition which you can download from LinuxCommand.org."

This book is in 4 parts. Part 1, 'Learning the shell', is all about the command line. Part 2, 'Configuration and the Environment' is all about customising bash: environment variables, shell configuration scripts, customising the prompt and a gentle introduction to vi.

Part 3, 'Common tasks and Essential tools' explains package management for both .deb and .rpm oriented systems, and explains storage media, in particular mounting/unmounting devices, filesystems and CD-ROM images. It also explains and introduces networking, searching for files, archiving and backing up, regular expressions, the arcane arts of text processing, printing and explains how to compile C programmes from a tar ball.

Part 4 goes into detail about writing shell scripts: as well as syntax, it covers best practice for shell scripting and debugging.

I read this book to get a lot more knowledge about the bash command line and scripting. It did not disappoint.

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?

View from the Chair

Bob Schmidt
chair@accu.org

Another year has passed, and it is once again time for ACCU's Annual General Meeting (AGM) and the start of a new term. First, we need to look back at the year that is ending.

After an uptick in membership at the end of 2017, in 2018 we returned to our downward trend in membership. According to the numbers compiled by Matt Jones (Membership Secretary), in December 2017 we had 670 members compared to 618 members in December 2018. This represents an almost ten percent decline in membership in one year. 2019 year-to-date numbers remain flat.

This continues to be our biggest concern. Obviously we can't maintain 10 percent losses in membership forever. At some point, our fixed costs will exceed our income from membership, and we will have to tap our surplus to pay our bills.

It hasn't been all bad news:

- Our financial situation remains strong, with ACCU finishing 2018 with a surplus for the fiscal year. Conference revenues continue to be the main source of our surpluses.
- *Overload* and *CVu* continue to excel under the stewardship of Fran Buontempo and Steve Love (respectively).
- We published our response to the EU's General Data Protection Regulation (GDPR), thanks to the efforts of Nigel Lester.
- Our local groups continue to experience strong membership growth, as measured by the number of people signed up for the respective Meetup Groups. (As was the case last year, the large number of new Meetup members has not resulted in an increase in ACCU members.)
- We are in the beginning phases of moving our web site to a new platform, with a new look. Jim Hague (Webmaster) has created a framework with which we can move forward.
- As of this writing, registrations for the 2019 conference are a little lower than last year, but higher than for 2017. Concerns over Brexit may be contributing to the decline.

We continue to have other challenges:

- We continue to have difficulty getting people to volunteer, resulting in long-term vacancies on the committee. The Publicity, Study Groups, Social Media, Web Editor, and Book Reviews positions have all been vacant for at least a year. The committee has discussed discontinuing some of the positions that have been vacant for the longest periods of time.
- We continue to need authors and articles for our magazines. You probably have noticed that the page count of the magazines varies from month to month, which is based on the content received during the two months preceding publication.

We have some changes coming to the makeup of the ACCU Committee:

- Our Treasurer, Rob Pauer, has decided to not seek reelection for the 2019–2020 term. As of this writing, we have not had anyone volunteer to run for the position.
- After many years of service, Nigel Lester is stepping down as Local Groups Coordinator. Phil Nash is standing for the position.
- Emyr Williams is stepping down as Standards Officer. Guy Davidson is running to take over in that role.

Because Guy is planning to join the committee, he will not be available to finish his second year as Auditor. We will need to find someone to fill that role. Niall Douglas' current term as auditor is ending.

We should have some continuity on the committee, with the following people running for their current positions: Matt Jones, Membership Secretary; Patrick Martin, Secretary; Roger Orr, Publications; and Ralph McArdell, At-large. Russel Winder (ACCU Conference Chair), Seb Rose (Advertising), Jim Hague (Webmaster), and Daniel James (ePub Editor) remain as co-opted members of the committee.

As always my thanks go out to all of ACCU's volunteers: committee members; magazine editors, writers and reviewers; local group organizers; auditors; and conference committee members. Special thanks to Rob, Nigel, Emyr, and Niall for their service.

In other news, we are pleased to announce that Felix Petriconi has been selected as ACCU Conference Chair, starting after the 2020 conference. Felix will shadow Russel Winder during the planning for the 2020 conference (which will be Russel's last as Conference Chair) in order to ensure a smooth handover.

Felix finished his Electrical Engineering studies in 1993, and started his career as a freelance programmer in the telecommunications and automotive industries. In 2003, Felix joined MeVis Medical Solutions AG in Bremen, Germany, as a programmer and development manager, where he is part of a team that develops radiological medical devices. He is a regular speaker at the C++ user group in Bremen, and has been a speaker at software conferences in Bristol, Berlin, Milan, and Wroclaw. He is a blog editor of https://isocpp.org and a contributor to the https://stlab.cc open source concurrency library. Felix has been a member of the ACCU Conference committee since 2016.

Please join me in thanking Felix for volunteering for this very important and high-profile position within ACCU.

Portions of this 'View' appeared in the 2019 Annual General Meeting Information Pack as part of the Chair's Report.

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.

JOIN THE ACCU!

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

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

With full ACCU membership you get:

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

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

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

How to join

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

Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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