the magazine of the accu

www.accu.org

{cvu}

Volume 32 • Issue 1 • March 2020 • £4.50

Features

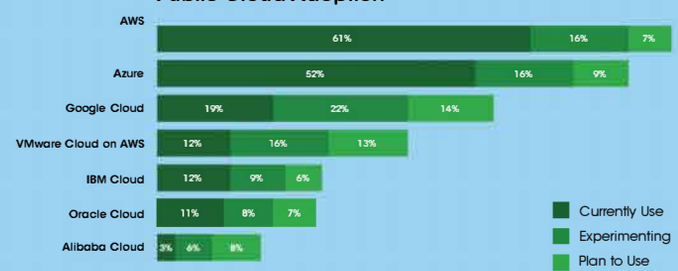
Regulars

ACCELERATING DEVOPS IN THE CLOUD

What Should You Know When Considering Moving Your Software Developement to the Cloud? Here's what SMBs and Enterprises are doing

94%
of Companies use the Cloud*

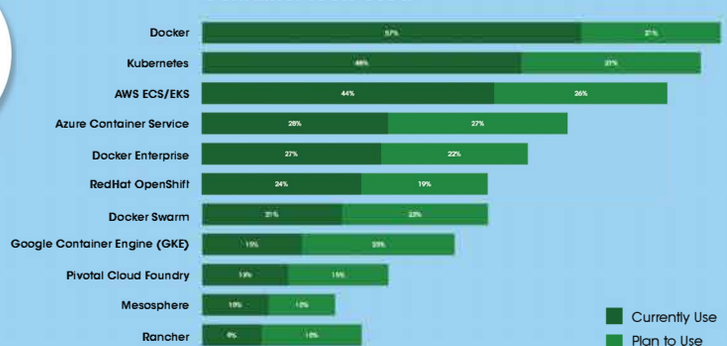
Public Cloud Adoption*



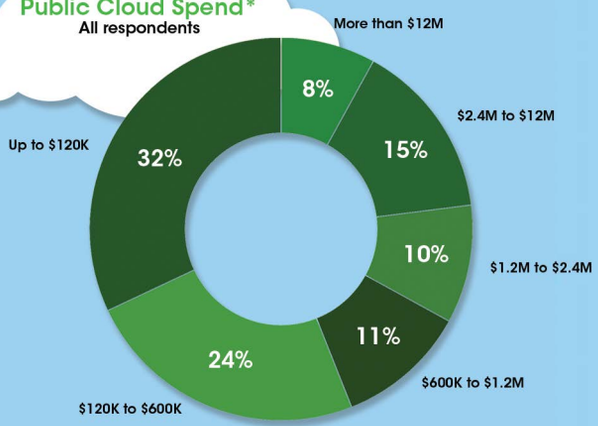
Average of Cloud Used* 5

* of Public + Private Clouds Used	Average	Median
Currently Using	3.4	3.0
Experimenting	1.5	1.0
Total	4.9	4.0

Container Tools Used*



Annual Public Cloud Spend*
All respondents



57% of companies use Docker Containers

48% use Kubernetes as their orchestration tool

JFrog cloud solutions provide businesses that develop software with universal DevOps tools designed to accelerate their CI/CD pipeline and gain trust in their releases from code to end device.

References: *RightScale 2019 State of the Cloud Report from Flexera

Key partners include:

Agisoft · ASPOSE File Format APIs · ATLASSIAN · axure · BrowserStack · DevExpress · embarcadero · Hex-Rays · intel Software · JetBrains Platinum Reseller · Microsoft Silver Partner · qbs Publishing · Sketch · SMARTBEAR · SPARX SYSTEMS · Telerik Develop experiences · think-cell · Visual Paradigm

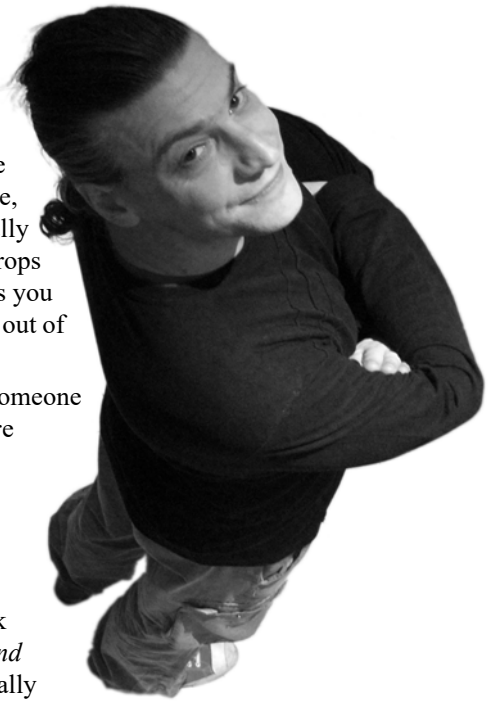
Plus many more on **www.qbssoftware.com**

sales@qbssoftware.com | 020 8733 7100

QBS SOFTWARE

{cvu}
Volume 32 Issue 1
March 2020
ISSN 1354-3164
www.accu.org

Imaginary friends



I guess it happens to all programmers at some point. You're in the office late, and alone, trying to track down some intermittent bug, or crack a gnarly problem, and the later you stay, the more tired you get, and the solution just seems to get further and further away. Sometimes you can end up making matters worse as you hack and chop your way through the code, getting ever more frustrated until you eventually give up and go home. In the morning, someone drops by your desk to ask how things are going, and as you start to explain your problem, the solution drops out of the sky, just like that. Huh.

Sometimes the act of explaining a problem to someone inspires you to discover the solution as you're talking. The process of explaining forces you to get your thoughts into a sensible order. The really interesting thing is, you don't actually need a person for this to work. It's not a new idea, and is variously known as the 'cardboard programmer' and (my favourite) 'rubber duck debugging'. The idea is you only have to *pretend* that you're explaining to someone who can actually help you out. It really can work with a rubber duck!

It can also help to write things down. When people say that writing articles can really help you deeply understand something, it's because an article forces you to get your thoughts into line, and also to explore some corners of the topic you might not ordinarily think of. The same principle applies to creating a talk or presentation. When you're writing an article or presentation, you're really about explaining things to people *who aren't actually there*! This adds a further dimension, because you find yourself having to think about what those imaginary people might want to get out of the final result.

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.

accu

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 32.2: 1st April 2020
C Vu 32.3: 1st June 2020

Overload 156: 1st May 2020
Overload 157: 1st July 2020

WRITE FOR C VU

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

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

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

The Ethical Programmer

Pete Goodliffe considers an ethical approach to programming.

Ethics is in origin the art of recommending to others the sacrifices required for cooperation with oneself.
~ Bertrand Russell

I often describe how the quality of a coder depends more on their *attitude* than their technical prowess. A recent conversation on this subject led me to consider the topic of *the ethical programmer*.

What does this mean? What does it look like? Do ethics even have an appreciable part to play in the programmer's life?

It's impossible to divorce the act of programming from any other part of the coder's human existence. So, naturally, ethical concerns govern what we, as programmers, do and how we relate to people professionally.

It stands to reason, then, that being an 'ethical programmer' is a worthwhile thing; at least as worthwhile as being an ethical *person*. You'd certainly worry about anyone who aspired to be an *unethical programmer*.

Many professions have specific ethical codes of conduct. The medical profession has the Hippocratic oath, binding doctors to work for the benefit of their patients, and to not commit harm. Lawyers and engineers have their own professional bodies conferring chartered status, which require members to abide by certain rules of conduct. These ethical codes exist to protect their clients, to safeguard the practitioners, as well as to ensure the good name of the profession.

In software 'engineering' we have no such universal rules. There are few industry standards that we can be usefully accredited against. Various organisations publish their own crafted code of ethics, for example the ACM [1] and BCS [2]. However, these have little legal standing, nor are they universally recognised.

The ethics of our work are largely guided by our own moral compass. There are certainly many great coders out there who work for the love of their craft and the advancement of the profession. There are also some shadier types who are playing the game predominantly for their own selfish gain. I've met both.

The subject of computer ethics was first coined by Walter Maner in the mid-1970s. Like other topics of ethical study, this is considered a branch of philosophy.

Working as an 'ethical' programmer has considerations in a number of areas: notably in our attitudes towards code, and towards people. There are also a bunch of legal issues that need to be understood. We'll look at these in the following sections.

Attitude to code

Do not write code that is deliberately hard to read, or designed in such a complex way that no one else can follow it.

We joke about this being a 'job security' scheme: writing code that only you can read will ensure you will never get fired! Ethical programmers know that their job security lies in their talent, integrity, and value to a company, not in their ability to engineer the company to depend on them.

Do not make yourself 'indispensable' by writing unreadable or unnecessary 'clever' code.

Do not 'fix' bugs by putting sticking-plaster workarounds or quick bodges in place, hiding one issue but leaving the door open for other variants of the problem to manifest. The ethical programmer finds the bug, understands it, and applies a proper, solid, tested fix. It's the 'professional' thing to do.

So, what happens if you're within a gnat's whisker of an unmovable deadline and you simply *have* to ship code, when you discover an awful, embarrassing, showstopping bug? Is it ethical to apply a temporary quick-fix in order to rescue the imminent release? Perhaps. In this case, it may be a perfectly pragmatic solution. But the ethical programmer does not let it rest here: he adds a new task to the work pool to track the 'technical debt' incurred, and attempts to pay it off shortly after the software ships. These kinds of Band-Aid solutions should not be left to fester any longer than necessary.

The ethical programmer aims to write the best code possible. At any point in time, work to the best of your ability. Employ the most appropriate tools and techniques that will lead to the best results – for example, use automated tests that ensure quality, pair programming, and/or code review to catch mistakes and sharpen designs.

Legal issues

An ethical, professional programmer understands pertinent legal issues and makes sure to abide by the rules. Consider, for example, the thorny field of software licensing.

Do not use copyrighted code, like source provided under the GNU GPL (General Public Licence) [3], in proprietary code when the licence does not permit this.

> Honour software licences.

When changing jobs, do not take source code or technology from an old company and transplant parts of it into a new company. Or even show parts of it in an interview with another company.

This is an interesting topic, as it leads to a large grey area: copying private intellectual property or code that has a clear copyright notice is clearly stealing. However, we hire programmers based on their prior experience – the things they have done in the past. Is rewriting the same kind of code from memory, without duplicating exact source lines, ethical? Is re-implementing another version of a proprietary algorithm that conferred competitive advantage unethical if you've hired the designer of that algorithm specifically for their experience?

Often code is published online with a very liberal licence, merely asking for attribution. The ethical programmer takes care to make sure attribution is given appropriately for such code.

> Ensure appropriate credit is given for work you reuse in your codebase.

If you know that there are legal issues surrounding some technology you're using (e.g., encryption or decryption algorithms that are encumbered by trade restrictions), you have to make sure your work does not violate these laws.

Do not steal software, or use pirated development tools. If you are given a copy of an IDE make sure that there is a valid licence for you to use it.

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

Just as you would not pirate a movie, or share copyrighted music online, you should not use illegally copied technical books.

Do not hack or crack your way into computers or information stores for which you do not have access authority. If you realise it's possible to access such a system, let the administrators know so they can remedy the permissions.

Attitude to people

Treat others as you would want them to treat you.
~ Matthew 7:12

We've already considered some 'ethical attitudes' towards people, as we write code primarily for the audience of other programmers, not for the compiler. Programming problems are almost always people problems, even if the solutions have a technical nature.

Good attitudes towards code are *also* good attitudes to other programmers.

Imagine yourself as a heroic coder. The kind of programmer who wears your underwear atop your trousers, and not simply as a geeky fashion faux pas. Now: *do not abuse your super powers for evil*. Only write software for the good of mankind.

In practice, this means: don't write viruses or malware. Don't write software that breaks the law. Don't write software to make people's life worse, either materially, physically, emotionally, or psychologically.

Don't turn to the dark side.

> Do not write software that will make another person's life worse. This is an abuse of power.

And here we open a wonderful new can of worms: is it ethical to write software that makes some people very rich at the expense of poorer people, if it doesn't break any laws? Is it ethical to write software to distribute pornographic content, if the software itself breaks no laws? You can argue that people are exploited as a by-product of both activities. Is it ethical to work in these industries? This is a question that I can only leave for the reader to answer.

What about working on military projects? Would an ethical programmer feel comfortable working on weapons systems that could be used to take a life? Perhaps such a system will actually *save* lives by acting as a deterrent against attack. This is a great example of how the ethics of software development is a philosophical topic, not an entirely black-and-white affair. You have to reconcile yourself with the consequences that your code has on other people's lives.

Teammates

The people you encounter most frequently in your programming career are your teammates – the programmers, testers, and so on, that you work with closely day by day. The ethical programmer works conscientiously with all of them, looking to honour each team member, and to work together to achieve the best result possible.

Speak well of all people. Do not engage in gossip or back-biting. Do not encourage jokes at the expense of others.

Always believe that anyone, no matter how mature or how experienced, has something valuable to contribute. They have an opinion that is worth hearing, and should be able to put forward points of view without being shot down.

Be honest and trustworthy. Deal with everyone with integrity.

Do not pretend to agree with someone when you believe they are wrong; this is dishonest and rarely useful. Constructive disagreements and reasoned discussions can lead to genuinely better code design decisions. Understand what level of ``debate' a team member can handle. Some people thrive on intense, passionate intellectual debate; others are frightened by confrontation. The ethical programmer seeks to engage in the most productive discussion result without insulting or offending anyone. This isn't always possible, but the goal is to always treat people with respect.

Do not discriminate against anyone, on any grounds, including gender, race, physical ability, sexual orientation, mental capability, or skill.

The ethical programmer takes great care to deal with ``difficult people' in the most fair, transparent way, attempts to diffuse difficult situations, and works to avoid unnecessary conflict.

> Treat others as you would like them to treat you.

Manager

Many issues that may be seen as agreements between you and your manager can also be seen as ethical contracts between you and the other members of the team, as the manager acts as a bridge with the team.

Do not take credit for work that is not yours, even if you take someone else's idea and modify it to fit the context a bit. Give credit where it's due.

Do not give an unnecessarily high estimate for the complexity of a task, just so that you can slack off and do something more enjoyable on the pretence of working hard on a tricky problem.

If you see looming issues that will prevent the smooth running of the project, report them as soon as you notice. Do not hide bad news because you don't want to worry or offend someone, or be seen to be a pessimistic killjoy. The sooner issues are raised, planned around, and dealt with, the smoother the project will go for everyone.

If you spot a bug in the system, report it. Put a bug in the fault tracking system. Don't turn a blind eye and hope that no one else will notice it.

> The ethical programmer takes responsibility for the quality of the product at all times.

Do not pretend to have skills or technical knowledge that you do not possess. Don't put a project schedule in danger by committing to a task you cannot complete, just because you think it's interesting and you'd like to work on it.

If you realise that a task you're working on is going to take significantly longer to complete that you expected, voice that concern as soon as possible. The ethical programmer does not keep it quiet in order to save face.

When given responsibility for something, honour the trust placed in you. Work to the best of your ability to fulfil that responsibility.

Employer

Treat your employer with respect.

Do not reveal company confidential information, including source code, algorithms, or inside information. Do not break the terms of your employment contact.

Don't sell work you have done for one company to another, unless you have express permission to do so.

However, if you realise that your employer is engaged in illegal activities, it is your ethical duty to raise it with them, or report their malfeasance as appropriate. The ethical programmer does not turn a blind eye to wrongdoing just to keep her own job secure.

Do not falsely represent, or bad-mouth your employer in public.

Yourself

As an ethical programmer, you should keep yourself up-to-date on good programming practice.

Ethical programmers do not work so hard that they burn themselves out. This is not only personally disadvantageous, but also bad news for the whole team. Hours and hours of extra work, week in, week out, will lead to a tired programmer, which will inevitably lead to sloppy mistakes, and a worse final outcome. The ethical programmer understands that although

'HTTPS Everywhere' Less Harmful Now

Silas S. Brown takes another look at web security.

This is a follow-up to my article in *C Vu* 27.6 (January 2016), 'HTTPS Everywhere Considered Harmful' [1].

At the time (end-2015), I was particularly concerned about people whose only connection to the Internet is via 2G GSM, reading public articles. Since the question of 'which article are you reading' could be answered via traffic analysis (unless we make all articles the same length), in this situation HTTPS gave little benefit and had the huge drawback of being almost unworkable at 2G data speeds.

Of course, nowadays fewer people are stuck on 2G speeds, and some places (like Taiwan) have even turned off their 2G networks completely. Additionally, TLS 1.3 is making the protocol work better on high-latency connections, and browsers have made other optimisations that can actually make HTTPS even faster than HTTP. The only remaining problem is low-income users on ancient equipment that cannot keep up with the latest developments in encryption software.

SILAS S. BROWN

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

The Ethical Programmer (continued)

it's nice to look like a hero who works incredibly hard, it's not a good idea to set unrealistic expectations and to burn yourself out.

> A tired programmer is of no use to anyone. Do not overwork. Know your limits.

You have a right to expect the same ethical conduct from others you work with.

The Hippocodic Oath

What would the ideal programmers' *code of ethics* look like? The ACM and BSI ethics documents are formal, lengthy, and hard to recall.

We need something pithier; more of a mission statement for the ethical programmer.

I humbly suggest:

> I swear to cause no harm to the code, or to the business; to seek personal advancement, and the advancement of my craft. I will perform my allotted tasks to the best of my ability, working harmoniously with my team. I will deal with others with integrity, working to make the project, and the team, maximally effective and valuable.

Conclusion

How much you care about this kind of thing depends on your level of diligence, your professionalism, and your personal moral code. Are you in the programming game for fun, enjoyment, and the development of great code? Or are you in it for yourself, for career development (at the expense of others, if necessary), to make as much money as you can, and to hoist yourself above others on the professional ladder?

It's a choice. You can choose your attitude. It will shape the trajectory of your career.

I find my attitudes shaped by my desire to write good code and to participate in a community that cares about working well. I seek to work amongst excellent developers whom I can learn from. As a Christian, I have a moral framework that encourages me to prefer others over myself and to honour my employers. This shapes the way I act.

I conclude from what we've seen here that there are (at least) two levels to the ethical programming career: the mandate to 'do no harm' is the base level, to not tread on people, or be involved in work that exploits others. Beyond this is a more involved ethical mantra: to only work on projects that provide sound social benefits, to specifically *make the world better* with your talents, and to share knowledge in order to advance the programming craft. I suspect that many people ascribe to the first camp of ethics. Fewer feel the urge, or are able to devote themselves to the cause of the second.

How do your beliefs and attitudes shape the way you work? Do you think you are an ethical programmer? ■

Questions

- Do you consider yourself an 'ethical' programmer? Is there a difference between being an ethical programmer and an ethical person?
- Do you agree or disagree with any of the observations in this article? Why?
- Is it ethical to write software that makes bankers fabulously wealthy if the money they make comes at the expense of other people, who are not able to exploit the same computing power? Does it make a difference whether the trading practice is legal or not?
- If your company is using GPL code in its proprietary products, but is not fulfilling the obligations of the licence terms (by withholding its own code), what should you do? Should you lobby for the licence terms to be met by open sourcing the company's code? Or should you ask for that GPL code to be replaced with a closed source alternative? If the product has already shipped, should you be a 'whistle-blower' and expose the licence violation? What if your job security depended on keeping quiet?
- What should you do if you identify another programmer acting 'unethically'? How does this answer differ if that programmer is a co-worker, a friend, someone whom you've been asked to give a reference for, or a coder you've met but do not work directly with?
- How do software patents fit into the world of ethical programming?
- Should your passion for software development have any bearing on how much you care about ethical issues? Does a passionate programmer act more ethically than a career coder?

References

[1] ACM Code of Ethics and Professional Conduct (Association for Computer Machinery): http://www.acm.org/about/code-of-ethics

[2] BCS: http://www.bcs.org/server.php?show=nav.6030

[3] GNU GPL (General Public Licence): http://www.gnu.org/licenses/gpl.html

Faster than HTTP?

The reason why modern browsers make HTTPS faster than HTTP is that they turn on the HTTP2 protocol (a successor to SPDY) and use its 'multiplexing', which is like an improved version of the 'pipelining' feature in HTTP/1.1 that most browsers didn't actually use (they benefitted from Keep-Alive, but everything else was done by opening separate connections to the server rather than true pipelining). That means, on a page with images, scripts, etc, HTTPS can now result in less network traffic than HTTP, reversing the previous situation. The 'BREACH' attack (which relies on being able to trick somebody's browser into sending chosen strings to the server and observing the size of its compressed responses) threatens to undo this if server administrators elect to mitigate it by switching off HTTP compression, but hopefully they'll be sensible enough not to switch off compression when the Referer header confirms the request is a follow-up to the same domain.

I do feel sad about client hardware having to be replaced before it's physically broken

Integrity and Wi-Fi

My previous position was that HTTPS is needed for situations where private data is being exchanged, such as in banking or private messaging, but it's less-obviously needed when you're reading public articles, especially if the question of 'which article are you reading' can be guessed anyway from response lengths (not to mention surveillance cameras and screen-recording software) so using HTTPS doesn't actually buy you much. But what if someone tries to tamper with the article?

I would have guessed that if the websites you read are going to be tampered with, then this dark deed would far more likely be done by malware on either the machine you're using or on the server itself, rather than the network in the middle. But then I visited London and tried to use the Underground Wi-Fi.

It's generally accepted that public Wi-Fi services will intercept your first HTTP request and redirect it to an 'accept our terms' page. Newer smartphones even issue a test request automatically, to save you the trouble of finding an HTTP-only website to do it yourself. But the London Underground Wi-Fi was stuck on intercepting all HTTP requests, even after the terms had already been accepted and HTTPS was working perfectly. Since HTTP-only websites are a dying breed, the developers are now less likely to test that use-case, and we are going to see more public Wi-Fi that inadvertently blocks HTTP-only sites, so if it's not available on HTTPS it simply won't be available. (Thankfully I wasn't stranded on the Underground: I had someone with me, plus I'd downloaded the blind-friendly descriptions of all stations from Describe Online in advance of the trip: I wasn't naive enough to rely on the Wi-Fi working for essential navigation data. But the problems I had with non-essential access may show what's coming.)

And that's not even considering the possibility of malicious Wi-Fi networks injecting malware into pages, nor the question of whether having non-sensitive websites on HTTPS actually helps the more sensitive sites out there, by cultivating a general expectation that HTTPS is 'normal'.

But what about old equipment?

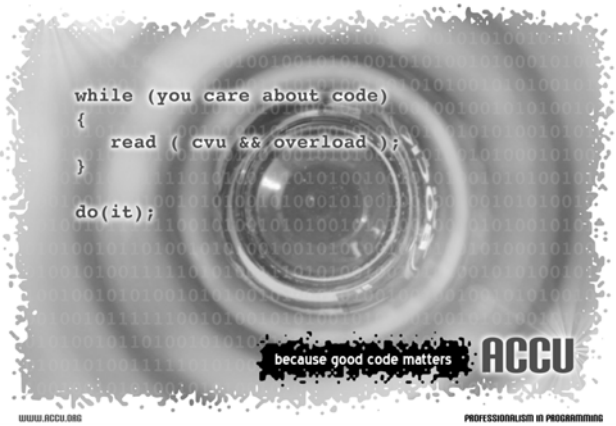
The one remaining problem with HTTPS is that of old equipment. I'm not worried about the climate-change implications of doing cryptography on the server (it's now negligible, and probably outweighed by the positive contribution of the latest protocols' reduced traffic), but I do feel sad about client hardware having to be replaced before it's physically broken (unless the new one has significant power-consumption benefits), and I'm also concerned about low-income users who can't afford to upgrade their hardware as soon as their old device is no longer supported by software updates. Many newer Certificate Authorities are not recognised by the software installed on older devices, resulting in these users having to click through a myriad of security warnings (which is undesirable but manageable), or being locked out by HSTS (which means they have to look up how to clear the browser's HSTS cache), but a bigger problem is an increasing number of servers, like Wikimedia's, are insisting on using only the latest and most secure encryption methods, thus cutting off these users altogether unless they find a transcoding proxy service (such as the one I run, but it has traffic limits so I don't advertise it prominently).

For a non-sensitive website, you could say 'I'll use HTTPS to work around Wi-Fi issues, but I'll still allow old equipment to connect with broken old cryptography, because the risk of the network running a cryptography exploit is low enough, whereas the odds of plain HTTP being intercepted are higher'. But that position requires you to have fine-grained control of which types of cryptography your HTTPS server supports. If it's a shared server then some well-meaning administrator could come along and turn off the old ciphers, but even if you run it yourself, you're still at the mercy of your software vendor when you apply security patches, unless you want to take over responsibility of maintaining your entire software stack from the ground up. It seems paradoxical that plain HTTP, whose security is even worse than that of the broken ciphers, will likely be supported by vendors for longer than old-HTTPS will, but at least administrators are more likely to know what they're getting by turning it on.

My personal website recently moved from being HTTP-only to accepting both HTTP and HTTPS. I wasn't sure which old browsers support URIs starting with '//' to mean 'use the current protocol', so I made all internal links relative and am currently considering mechanisms to select the protocols of external links when I know the site I link to supports both protocols (which few now do). At the time of writing I still declare plain-HTTP to be the preferred 'Canonical' URLs of my pages for search engines, which strongly advise you to choose a single 'Canonical' URL to accumulate all ranking points from different versions of the same page; most Western search engines now use HTTPS-only anyway, so you may be tempted to say 'if the visitor was using a search engine then they must have equipment that can handle HTTPS', but the dominant Chinese search engine 'Baidu' is still on plain HTTP, and there's no way to declare different 'Canonical' URLs to different search engines (at least not without using user-agent based 'cloaking' which is inadvisable: they do spot-checks from other browser strings and IPs), nor is there a way to tell Google you have 'moved to HTTPS' without actually 301-redirecting all requests to it, which would defeat the purpose of supporting both protocols. Also, search engines tend to exclusively own their IP addresses and so don't have to worry about old browsers not supporting SNI (Server Name Indication) when checking the site's certificate, whereas this is more likely to be an issue for a smaller site on a virtual-hosting service. So I'm still not sure when I should set 'Canonical' to HTTPS, but at least I'm now supporting both protocols and will take links/bookmarks on either. ■

Reference

[1] Available to members at: https://accu.org/index.php/journals/2192



Adding Python 3 Compatibility to Python 2 Code

Silas S. Brown explains how to cope with the differences.

When Python 3 was new, its pace of change was fairly quick, and as most of us didn't want to spend too long rewriting our code to adapt to every new release, we carried on using the far more stable Python 2. Now that Python 2 is being thrown out of GNU/Linux distributions, we're finally having to convert all our code to Python 3 (unless we want to compile Python 2 in our home directories and just hope no more security issues arise, although that approach is not possible in every situation), and Python's '2to3' tool does not help with everything (I don't use it as in my case it did more harm than good to my code). Since I have a lot of legacy Python code and I'd rather work with 'stable intermediate forms', I have been trying to convert as much as possible of it to work on both Python 2 and Python 3 from the same codebase. But this dual-compatibility has more caveats.

Byte-strings

In Python 2, the default string type is a byte-string, and Unicode strings are something else. But a Unicode string containing only ASCII will compare as equal to the same ASCII in a byte-string, and the index operator `[]` on a string will give a string of length 1 in both byte and Unicode strings. In Python 3, however, the default string type is now Unicode (and the representation for byte string-literals is not compatible with all versions of Python 2), and more subtly a Unicode string containing only ASCII will *not* be considered equal to its equivalent byte-string, and the index operator `[]` on a byte-string gives an integer: if you want a string of length 1 then you'd better convert it into slice notation i.e. `s[i:i+1]` instead of `s[i]`. Since the slice-notation version behaves identically in Python 2 and Python 3, I suggest converting all single-index operators to that, plus making sure as much as possible of your code will work regardless of whether it's given byte-strings or Unicode-strings as input, using `type` if necessary to determine the type of its input. But remember `str` means different things on the two platforms; a quick way of checking if we're on Python 3 is to check if `type("")==type(u"")`.

Code that mentions `encode('utf-8')` or `decode('utf-8')` will particularly need attention (and even more so if other character sets are in use). I also find it useful to define some small helper functions to 'make sure this thing is a byte-string' (calling `.encode` if it's Unicode) or 'make sure this thing is a Unicode-string' (calling `.decode` if it's a byte-string) – sometimes these are best done in such a way that non-string objects can be passed through unchanged. String operations like `.replace` (and the regex library) can work on both Unicode strings and byte-strings, but they'll fault if there's inconsistency between their parameters (e.g. `b.replace(x,y)` where `b` is a byte-string and `x` and `y` are Unicode strings will fail), so those 'make sure this thing is a' helper functions can be especially useful for porting regex-related code.

Another thing to be aware of is that file I/O (and `stdin`, `stdout` and `stderr`) might or might not be done in UTF-8 by default: it depends on your system's locale. When you have the luxury of a GNU/Linux system that's set to UTF-8 by default, it's easy to forget that the Microsoft Windows platform has an annoying habit of setting locale charset to something other than UTF-8, and even some Linux-based environments (such as containers) use the 'C' locale instead, in which case Python 3's I/O (when not done in binary mode) will fault on anything that isn't ASCII. To work around this from inside your script (i.e. if setting up the right environment variables before Python runs is not an option), the easiest way is probably to write code like Listing 1. Obviously you should do this only if you know for sure that the input and output really should be in UTF-8 and the system's locales are simply not set up properly (see Listing 1).

Listing 1

```
if type("")==type(u""): # Python 3+
  import codecs
  # Make sure stdin and stdout are set to UTF-8,
  # even if the system's locales don't have
  # UTF-8.
  stdin=codecs.getreader("utf-8")
    (sys.stdin.buffer)
  stdout=codecs.getwriter("utf-8")
    (sys.stdout.buffer)
  old_stdin, sys.stdin = sys.stdin, stdin
  old_stdout, sys.stdout = sys.stdout, stdout
```

Numbers

In Python 2, division of two integers is an integer operation just as it is in C. But in Python 3, division of two integers will convert it to a floating-point number, and if you wanted to have the integer then you must ask for it explicitly. This likely means many of the divisions in your code will need some attention. Also the `L` suffix for long integers has been removed; if you want compatibility with early versions of Python 2 (which required `L`) and also Python 3, you'll probably have to reach these numbers by multiplying up or similar, and may also have to detect the Python version at runtime and go down different branches as appropriate.

Standard output and error

Python 3 of course makes `print` into a function which requires parentheses (and I still don't understand why that change gets more attention than the byte-strings change, but perhaps I do more work with Unicode than most English developers do). `print` with parentheses will also work in Python 2, but if supplied more than one argument, it will make its arguments look like a tuple, which is probably not what you want. Compatible with both versions is to restrict `print` to one argument and use format strings or construct the string manually (but remember to account for Unicode string / byte string differences in Python 3); also of note is that Python 2 code containing `print` by itself for a blank line will need to be written as `print()` in Python 3, or `print("")` for compatibility with both versions.

You might prefer to use `sys.stdout`, and/or `sys.stderr` for the 'standard error' stream (which is a separate stream if your program's standard output has been redirected to a file or pipe). But another difference between Python 2 and Python 3 is that, in Python 3, `sys.stderr` is buffered in the same way as `sys.stdout` is, i.e. the output won't happen until you call `sys.stderr.flush()` or output a newline. If this matters, you might need to add some calls to `sys.stderr.flush()` that are unnecessary (but harmless) in Python 2.

SILAS S. BROWN

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

Making a Linux Desktop – Launching Applications

Alan Griffiths continues his series on the Mir desktop.

I'm working on a project (Mir [1]) that, among other things, aims to make it easy to develop graphical 'desktop environments' for Linux. There's a lot of features that are common between all designs for desktop environments and, in addition, a lot that is common between the majority of designs. For example, it is common for applications to draw 'windows' and for these to be combined onto the screen.

By providing the common features, and for the rest offering both sensible defaults and an easy way to customise them Mir is designed to support a range of possible designs.

Last issue we finished with a basic but functional desktop shell with some wallpaper. This time we'll extend that with the capability to select and launch applications from the desktop.

Preparation

The code in this article needs Mir 1.2 (or later). On Ubuntu 18.04 (and later) Mir 1.7 is available from the mir-team/release PPA. To build it requires a few additional packages as indicated below. It is also useful to install the Qt toolkit's Wayland support: qtwayland5. And finally, the g++ compiler and cmake.

```
$ sudo apt-add-repository ppa:mir-team/release
$ sudo apt install libmiral-dev mir-graphics-
drivers-desktop libwayland-dev
$ sudo apt install libfreetype6-dev libxkbcommon-
dev libboost-filesystem-dev
$ sudo apt install qtwayland5
$ sudo apt install g++ cmake
```


ALAN GRIFFITHS

Alan Griffiths has delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk

Adding Python 3 Compatibility to Python 2 Code (continued)

Reading and writing from files in Python 3 automatically converts to/from Unicode strings; if you want bytes, you must either open the file in binary mode (`rb` or `wb`) or else use the file's `.buffer` member (which is not present on Python 2, so you'll have to write an `if-else` branch depending on the Python version). Note that `.buffer` is only a weak reference: you must keep a reference to the file itself, not just its buffer, or you'll find it has been automatically closed.

Library changes

There are too many standard library changes between Python 2 and Python 3. In some cases it's just a matter of importing a different module, and you can have `if-else` branches in your imports to maintain compatibility with both versions. For example, `commands.getoutput` now needs to be `subprocess.getoutput`, `thread` now needs to be `_thread`, and various HTML-related and urllib-related libraries may need importing differently. But there are other libraries with more substantial changes, e.g. the `email` module works completely differently in Python 3 (my IMAP-processing code is still stuck in Python 2 for this reason); some usage of `StringIO` might need to be `BytesIO` on Python 3 (and now imported from `io`); some exceptions have been renamed and might need assigning for compatibility; and version 6 of the third-party Tornado library has completely changed the way it does callbacks and `IOLoop` (although I managed to make Web Adjuster compatible with both versions by writing some fancy decorators).

Some built-in functions are also no longer available in Python 3, so you might have to write things like:

```
try: unichr # Python 2
except: unichr,xrange = chr,range # Python 3
```

to keep your code compatible. Also, some things that used to return lists now return iterators, and if you want a list you must explicitly ask for one, so for example you can no longer say:

```
Unicode_Greek_letters = range(0x3b1,0x3ca)
+ range(0x391,0x3aa) # wrong
```

you'll have to say `list(range())` instead. Most notably, `.items()` no longer returns a list: some Python 2 code will assume that it does, and will assume that the dictionary from which it was taken may be changed without averse effect on the `.items()` list it has (this is now likely to raise an exception if used in a loop), so you may wish to wrap all use of `.items()` in `list()` to help port this.

Also the `sort()` functions and methods have changed: they no longer take comparison functions, only key functions. Python 2 `sort()` can also take `key=`, so if you can rewrite all your comparison functions as key functions, i.e. functions that return the 'equivalent value' of a single item for sorting purposes, then you can write this in a way that's compatible with both 2 and 3.

There are many other subtle changes, and you will need to test the code carefully in both versions of Python before considering it compatible with both. But the above changes were the most important ones to make in my code so far.

Summary

The most likely places that will need amending are:

1. Anywhere where Unicode is converted to/from UTF-8, or where files are written/read
2. Any `[]` index operators that might be applied to byte strings (use slices for maximum compatibility)
3. Any use of `.replace` or `re.sub` (make sure it's all the same type)
4. Any divisions (should we take the integer?)
5. `print` and `import` statements
6. Any writes to `sys.stderr` (do we need to flush?)
7. Any use of `.items()` (does it need to be put into a `list()` now?), and `sort()` with comparison function

As always, good test coverage is the most important thing, and you may have to go through several iterations before it works. ■

Mir 1.6 is available on Fedora and can be built from source for many versions of Linux.

Building the example

The full code for this example is available on github [2]:

```
$ git clone https://github.com/AlanGriffiths/
  egmde.git
$ git checkout Article-3
```

Naturally, the code is likely to evolve, so you will find other branches, but the Article-3 branch goes with this article. Assuming that you've MirAL installed as described above you can now build egmde as follows:

```
$ mkdir egmde/build
$ cd egmde/build
$ cmake ..
$ make
```

Running the example

After this you can start egmde:

```
$ ./egmde
```

This time the Mir-on-X window should have a coloured gradient with the words "Ctrl-Alt-A = app launcher | Ctrl-Alt-T = terminal | Ctrl-Alt-BkSp = quit" at the bottom.

If you press CTRL-ALT-A you will see the name of one of the programs available on your computer, pressing up or down arrows will change the named program (as will pressing the initial letter of another program), pressing SPACE or ENTER will (attempt to) launch the program.

If you have a touchscreen, you can activate the app launcher by swiping from the left edge, launch the shown app by touching the middle of the screen and change apps by touching the top or bottom area.

Normally launching the program will succeed, but if the program doesn't run using Wayland then it will fail. (A future article will explore supporting X11-only applications.)

Listing 'desktop' applications

The 'Desktop Entry Specification' [3] details how desktops identify the available programs, the icons and text to display (in potentially many languages) and the command(s) to run them. Like GNOME, KDE, Unity and other desktop environments, egmde follows this specification.

This specification covers the places to look for `.desktop` files, and the content of these files. A large chunk of the new code deals with finding these files and extracting the name and command to execute.

The remaining code is based on the **`FullscreenClient`** class we saw in the last article and deals with offering a selection from the applications found in the `.desktop` files.

The example code

I'll leave the interested reader to find and read the code of the new **`Launcher`** class, it deals with parsing files, handling the display and user input. I'll just show the changes to the main file. Listing 1 shows the keyboard and touchscreen handling code to show the launcher.

In the initial article of this series, the program simply expected to use weston-terminal as a Wayland based terminal emulator. This version actually tries to find a terminal that's already installed. (There are two mechanisms for identifying a default terminal emulator on Linux: x-terminal-emulator from Debian and xdg-terminal from XDG. Sadly neither reflects user preferences, or Wayland support.) Listing 2 shows the new code.

Listing 1

```
ExternalClientLauncher external_client_launcher;
egmde::Launcher launcher
  {external_client_launcher};
…
  switch (mir_keyboard_event_scan_code(kev))
  {
    case KEY_A:launcher.show();
      return true;
…
auto touch_shortcuts = [&, gesture =
  false](MirEvent const* event) mutable
  {
    if (mir_event_get_type(event) !=
        mir_event_type_input)
      return false;
    auto const* input_event =
        mir_event_get_input_event(event);
    if (mir_input_event_get_type(input_event) !=
        mir_input_event_type_touch)
      return false;
    auto const* tev =
    mir_input_event_get_touch_event(input_event);
    if (gesture)
    {
      if (mir_touch_event_action(tev, 0) ==
          mir_touch_action_up) gesture = false;
        return true;
    }
    if (mir_touch_event_point_count(tev) != 1)
      return false;
    if (mir_touch_event_action(tev, 0) !=
        mir_touch_action_down)
      return false;
    if (mir_touch_event_axis_value(tev, 0,
        mir_touch_axis_x) >= 5)
      return false;
    launcher.show();
    gesture = true;
    return true;
  };
…
runner.add_stop_callback([&] {launcher.stop();});
```

Listing 2

```
namespace
{
// Neither xdg-terminal nor x-terminal-emulator
// is guaranteed to exist, and neither is a good
// way to identify user preference...
  std::string const terminal_cmd
    = []() ->   std::string
  {
    auto const user_bin = "/usr/bin/";
    for (std::string name : { "weston-terminal",
        "gnome-terminal", "konsole", "qterminal",
        "lxterminal", "xdg-terminal"})
    {
      if (boost::filesystem::exists(user_bin
          + name))
        return name;
    }
    return "x-terminal-emulator";
  }();
}
…
    external_client_launcher.launch(
        {"weston-terminal"});
    case KEY_T: external_client_launcher.launch
        ({terminal_cmd});
      return true;
```

DevelopHER Overall Award 2019

Paul Grenyer reports on his experience of being a judge.

I was honoured and delighted to be asked to judge and present the overall DevelopHER award once again this year. Everyone says choosing a winner is difficult. It may be a cliché, but that doesn't change the fact that it is.

When the 13 category winners came across my desk, I read through them all and reluctantly got it down to seven. Usually on a first pass, I like to have it down to three or four and then all I need to agonise over is the order. Luckily, on the second pass I was able to be ruthless and get it down to four.

To make it even more difficult, three of my four fell into three categories I am passionate about:

- Technical excellence and diversity
- Automated Testing
- Practical, visual Agile

And the fourth achieved results for her organisation which just couldn't be ignored.

So I read and reread and ordered and re-ordered. Made more tea, changed the CD and re-read and re-ordered some more. Eventually it became clear.

Technical excellence and the ability for a software engineer to turn their hand to new technologies is vital. When I started my career, there were basically two main programming languages, C++ and Java. C# came along soon after, but most people fell into one camp or another and a few of us crossed over. Now there are many, many more to choose from and lots of young engineers decide to specialise in one and are reluctant to learn and use others. This diminishes us all as an industry. So someone who likes to learn new and different technologies is a jewel in any company's crown.

The implementation of Agile methodologies in Software Development is extremely important. Software, by its very nature is complex. Only on the most trivial projects does the solution the users need look anything like what they thought they wanted at the beginning. Traditional waterfall approaches to software development do not allow for this. The client requires flexibility and we as software engineers need the flexibility to

deliver what they need. Software development is a learning process for both the client and the software engineer. Agile gives us a framework for this. Unlike many of the traditional methods, Agile has the flexibility to be agile itself, giving continuous improvement.

When implementing Agile processes, the practices are often forgotten or neglected and in many ways they are more important. Not least of which is automated testing: the practice of writing code which tests your code and running it at least on every checkin. This gives you a safety net that code you've already written isn't broken by new code you write. And when it is, the tests tell you, they tell you what's wrong and where it's wrong. We need more of this as an industry and that is why I chose Rita Cristina Leitao, an automated software tester from Switch Studios as the overall DevelopHER winner. ■

PAUL GRENYER

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

Making a Linux Desktop – Launching Applications (continued)

Conclusion

The program has grown from the 75 lines in the initial working example, but has started to look a little more polished and is a lot more functional. It is still not a big program (see below). ■

```
$ wc -l *.cpp *.h
  616 egfullscreenclient.cpp
  639 eglauncher.cpp
  147 egmde.cpp
  176 egwallpaper.cpp
   46 egwindowmanager.cpp
   53 egworker.cpp
  201 printer.cpp
  236 egfullscreenclient.h
   53 eglauncher.h
   58 egwallpaper.h
   43 egwindowmanager.h
   52 egworker.h
   54 printer.h
 2374 total
```

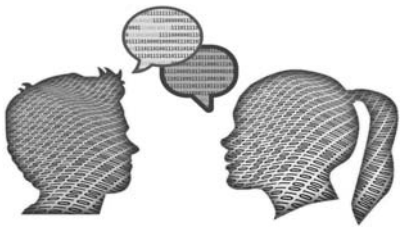
References

[1] The Mir homepage: https://mir-server.io/

[2] The egmde git repo: https://github.com/AlanGriffiths/egmde

[3] 'Desktop Entry Specification': https://specifications.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html

Code Critique Competition 122



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

```
$ queue.exe
main thread
Argument 0 = queue.exe
```

or sometimes only:

```
$ queue.exe
main thread
```

The coding is in three listings:

- Listing 1 contains `async_logger.h`
- Listing 2 contains `async_logger.cpp`
- Listing 3 contains `queue.cpp`.

Note: I inadvertently elided an underscore in transcribing the code for publication. I'm sorry for any confusion.

Listing 1

```
#pragma once

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

using std::thread;

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

  void run();

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

Listing 2

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

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

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

Listing 3

```
#include "async_logger.h"

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


ROGER ORR

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

Critiques

Paul Floyd <paulf@free.fr>

Let's start by running through 'main' to see what should be happening.

1. An instance of `async_logger` is created on the stack. The constructor creates a thread with a lambda that executes the `run()` member, sets the active flag and detaches the thread that was just created.
2. A log message is sent from `main()`.
3. A thread is created with a lambda that captures the logger instance by reference, and sends a message.
4. A `for` loop sends messages to the logger, one per command line argument.
5. A final ending message is sent to the logger.
6. The lambda thread from step 3 is joined.
7. `Logger` goes out of scope at the end of `main`. Its destructor clears the `active_` flag, causing `async_logger::run()` to terminate.

On a semi-positive note, I see that the two threads that get created are either detached or joined. Creating detached threads is not recommended, see http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-detached_thread.

Now for the problems. Firstly, a nitpick. I don't like the member variable of `async_logger` called `thread`. The other members use a trailing underscore, so I'd prefer that to be `thread thread_`.

Next, there is a problem with the initialisation of `async_logger::active_`. The `async_logger` constructor will initialise its members in order of declaration. `active_` is first, but is not initialised. Next `queue_` is initialised, via its constructor. Next `thread` is initialised, as per the default member initialiser. In the body of the constructor, `active_` is set to `true` and `thread` is detached.

There is a data race on the `active_` member. If the uninitialised value is `false` and `thread` runs to the `while (active_)` test before the constructor body sets `active_` to `true` then `run()` will terminate.

To fix this, move the initialisation of `active_` either to the initializer list or use default member initialisation. For consistency, the latter is what I would recommend.

The next problem is also a data race. Read and write to `async_logger::queue_` are not controlled, and `std::queue` is not thread safe.

To fix this, we can add a `std::mutex` to `async_logger` and then use a `std::lock_guard` (with an extra scope) in `async_logger::run()` and `async_logger::log()`.

Are we done? No, there are still two more data races. They both involve the `async_logger` destructor. Firstly, there is a data race in the `active_` member. The destructor sets `active_` to `false`, but `run` could be reading it at the same time. To solve this we can make `active_` an atomic (e.g., `std::atomic<bool> active_;`). In general this sort of error is 'mostly harmless' – if the race condition triggers it will probably just mean that `run` does one more iteration. However, I would still advise fixing the issue. The last data race concerns the destruction of `queue_`. The `run` thread could be accessing this member while the main thread destructor is destroying it. Unfortunately, there's no easy way in the destructor to know when the `run` thread has terminated (and so it is safe to destroy `queue_`). So here I suggest not detaching the `run` thread. Remove the `detach` from the constructor and add a `join` to the constructor. After the `join`, we know that the `run` thread has terminated, so it is safe to proceed with the default destruction of `queue_`.

Are we there yet? No, still a long way to go. The data races are fixed, but there are still two problems. The first is that `main()` does nothing to make sure that `async_logger::run()` has emptied `queue_` before it terminates (remember, the termination condition for run is that `active_` is `false`, not that `queue_` is empty. We can fix this by adding a loop to empty `queue_` in the `async_logger` destructor.

The last remaining issue is performance. `run()` is busy-polling. It will be using ~100% of a CPU as long as the application runs. The fix for this is to use thread communication, for instance a `std::condition_variable`. Add one as a member of `async_logger`, add a call to `notify_one` in `log` and add a `wait` in `run` (and change the `lock_guard` to `unique_guard`).

OK, now we are done. No more thread hazards or CPU hogging and no missing output.

I spotted most of the issues other than the destructor and falling off main just by looking at the code. For further testing I used Thread Sanitizer, Valgrind DRD and Valgrind Helgrind.

A bit now about tools. As an example, I ran the unmodified code with `clang++ -fsanitize=thread` and I got

```
WARNING: ThreadSanitizer: data race (pid=89319)
Read of size 1 at 0x7ffee1517890 by thread T1:
#0 async_logger::run() async_logger.cpp:8
(cc121:x86_64+0x1000015a0)

[snip]

Previous write of size 1 at 0x7ffee1517890 by
main thread:
#0 async_logger::async_logger()
async_logger.cpp:19 (cc121:x86_64+0x1000018b8)
```

Always test your thread code with a dynamic analysis tool!

I'll finish with a few comments about the design. As always, you should consider reusing code or packages where possible. I'm sure that there are many existing packages for performing asynchronous logging. The code here only logs to standard output. A more complete system would have options to log to a file, syslog, Windows Events as a few examples. I would also expect some sort if severity level like critical, serious, warning and information, perhaps with an option to filter out levels. I have just one last point. Systems like this have a hard time with logging events that occur during program shutdown. How can I log an event from the destructor of an object that is a file static? `logger`, being local to `main()`, will have already been destroyed when static destructors start getting called.

Ovidiu Parvu <accu@ovidiuparvu.com>

Let us assume throughout that two requirements need to be met by an `async_logger` object. Firstly an `async_logger` object can be used to log messages, which will be printed to the standard output in the order in which they were logged. Secondly the logged messages will be printed to the standard output on a separate thread. All messages should be printed to the standard output before the `async_logger` object is destructed.

There are four main issues with the implementation given in the listings provided.

First of all the code cannot be compiled successfully due to an undeclared identifier error. Specifically the `std::thread async_logger` data member is referred to as `thread_` at async_logger.cpp:20:3 but it is defined as `thread` in `async_logger.h`. Renaming the data member from `thread` to `thread_` as shown below, which would be consistent with the naming of other `async_logger` data members (e.g. `active_`), addresses this issue.

```
// async_logger.h
class async_logger
{
  ...
  thread thread_ { [this]() { run(); } };
  ...
};
```

Secondly, the `active_ async_logger` data member should be initialized with the value `true` when an `async_logger` object is constructed and before the `thread_` data member is initialized. Otherwise it may be possible for the `while (active_) { ... }` loop in the `run()` method to be executed before the value of `active_` is set to `true`, and

therefore the `run()` method will finish executing before the `async_logger` object is constructed. In this case no logged messages would be printed to the standard output. To avoid this issue the `active_` data member is initialized with the value `true` using a default member initializer as shown below.

```
// async_logger.h
class async_logger
{
  ...
  bool active_ = true;
  ...
};
// async_logger.cpp
async_logger::async_logger()
{
  thread_.detach();
}
```

Thirdly, read and write access from multiple threads to the `active_` and `queue_ async_logger` data members should be synchronized. Otherwise data races could occur. Access to the `active_` data member can be synchronized by changing the type of `active_` from `bool` to `std::atomic_bool` and importing the `<atomic>` header as follows.

```
// async_logger.h
#include <atomic>
...
class async_logger
{
  ...
  std::atomic_bool active_{true};
  ...
};
```

Conversely, access to the `queue_` data member can be synchronized using a `std::mutex`. The mutex needs to be used in different `async_logger` member functions and therefore will be added as a data member to the `async_logger` class. The mutex will be (unique) locked in the `run()` and `log(const std::string&)` methods before accessing the `queue_` data member as follows. In addition the `<mutex>` header must be included in `async_logger.h`.

```
// async_logger.h
...
#include <mutex>
...
class async_logger
{
  ...
  std::mutex mutex_;
  ...
};
// async_logger.cpp
// This runs in a dedicated thread
void async_logger::run()
{
  while (active_)
  {
    std::unique_lock<std::mutex> lock(mutex_);
    if (!queue_.empty())
    {
      std::cout << queue_.front() << '\n';
      queue_.pop();
    }
  }
}
// queue for processing on the other thread
void async_logger::log(const std::string& str)
{
  std::unique_lock<std::mutex> lock(mutex_);
  queue_.emplace(str);
}
```

Fourthly, the thread used to print messages to the standard output should not be `detach()`'ed in the `async_logger` constructor, and should be `join()`'ed in the `async_logger` destructor, because otherwise it will not be possible to wait for all messages to be printed on this thread before the `async_logger` object is destructed. After removing the `thread_.detach()` statement from the `async_logger` constructor, the constructor definition body is empty. Therefore both the `async_logger` constructor declaration and definition can be removed. The updated `async_logger` destructor definition is given below.

```
// async_logger.cpp
async_logger::~async_logger()
{
  active_ = false;
  thread_.join();
}
```

To ensure that all log messages are printed to the standard output before the `async_logger` is destructed, an additional loop is added to the `run()` method which prints all log messages remaining in `queue_` after `active_` is set to `false` and the `while (active_) { ... }` loop finishes executing. Both the newly added loop and the `while (active_) { ... }` loop pop the log messages from `queue_` and print the messages to the standard output. Therefore the log messages printing code can be extracted into a separate member function called `print_oldest_msg()` as follows.

```
// async_logger.h
class async_logger
{
  ...
  void print_oldest_msg();
  ...
};
// async_logger.cpp
// This runs in a dedicated thread
void async_logger::run()
{
  while (active_)
  {
    std::unique_lock<std::mutex> lock(mutex_);
    if (!queue_.empty()) print_oldest_msg();
  }
  while (!queue_.empty()) print_oldest_msg();
}
void async_logger::print_oldest_msg() {
  std::cout << queue_.front() << '\n';
  queue_.pop();
}
```

Additional minor improvements that could be made to the code given in provided listings are that the `using std::thread;` statements could be removed to avoid future potential name collisions should the project include user-defined thread type definitions in the future, and the unused `<iostream>` header include could be removed from `async_logger.h`.

[Ed: The code in Listings 2–4 updated according to all the changes described above was provided, but is omitted for brevity.]

James Holland <jim.robert.holland@gmail.com>

Some minor points. `#pragma once` is not standard C++. My version of `async_logger.h` uses the more universal method of writing Include Guards. The original code is missing a trailing underscore in the declaration of `thread_` within the `async_logger` class.

When I ran the software on my Linux system, it produced the desired output. This was unfortunate because it gave no hint as to what could be wrong. When the student ran the program, some of the expected output was missing. This suggests there are some problems with the ordering of events within the executing program. Some systems may order events in a different order than others. It is very difficult just by looking at the source code to determine the relative timing of events when multiple threads are involved. In an attempt to make some headway with this problem I

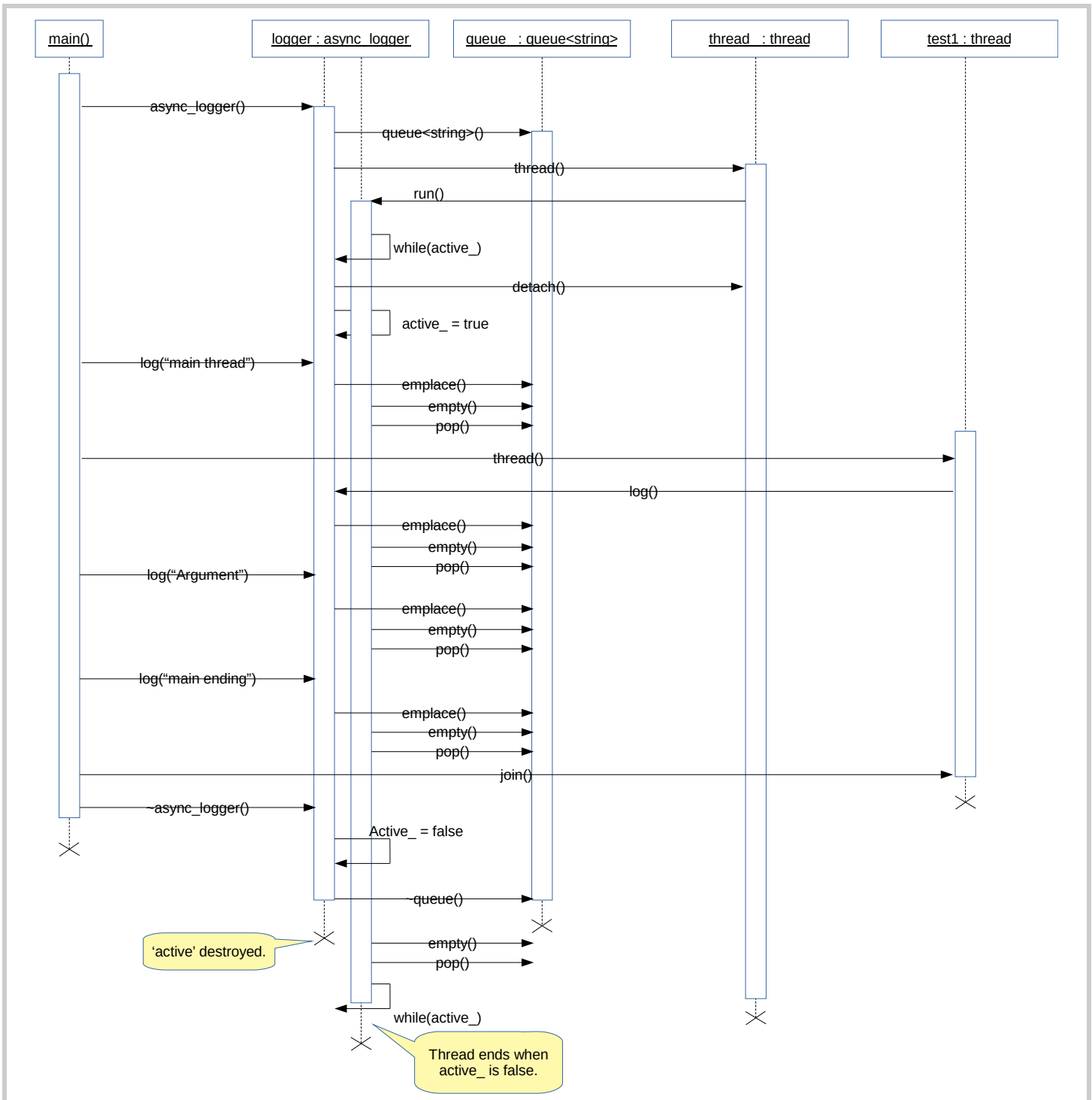
constructed a timing diagram that is loosely based on the Unified Modelling Language (UML) sequence diagram as shown in Figure 1.

At the top of the diagram are the objects involved in the exchange of messages. The time axis runs from top to bottom. One feature that is revealed is that the execution of **async_logger**'s **run** function's **while**-loop is dependent on value of the **active_** variable. Unfortunately, it is quite possible that the variable has not been assigned a value before it is read by the **while**-loop. The problem is that **active_** is not initialised as soon as it could be. It is assigned a value within the body of **async_logger()**. This gives time for the while-loop to read the uninitialised variable. Probably the simplest way of correcting this problem is to initialise **active_** to **true** within the class definition of **async_logger** (located in `async_logger.h`). Objects are initialised in the order they are declared within the class. Declaring (and initialising) **active_** before **thread_** ensures **active_** is set to the required value before it is read by **thread_**. The assignment of **active_** within the body of **async_logger**'s constructor can now be removed.

The sequence diagram also reveals a problem when logger is terminated. There is no guarantee that the calls to **empty()** and **pop()** will access a valid version of **queue_** or that **active_** exists when an attempted is made to access it by **run()**. Perhaps the simplest way to ensure these objects are still valid when accessed is for the destructor of **async_logger** to call **thread_**'s **join()** function instead of the constructor of **async_logger** calling **thread_**'s **detach()** function.

The compiler and the processor hardware are capable of optimising the program in ways that include altering the order in which instructions are executed as long as the result of a single thread of execution is the same as for the unaltered program. This can cause problems when two or more threads interact. In the program in question it is important that **active_** is assigned a value at the appropriate time relative other instructions. To

Figure 1



prevent such reordering, the type of **active_** should be changed from **bool** to **atomic_bool**.

With any luck, the program is now producing the desired result. The program relies on luck because the C++ standard makes no guarantees about the correct operation of **std::queue** when working in a multi-threaded environment. What is needed is a thread-safe queue. Such queues are a little involved and I do not intend to explore their construction here. One of the best sources of information regarding threads and concurrency in general is the book by Anthony Williams, *C++ Concurrency in Action*, second edition, ISBN 978-1617294-69-3. Here, various thread-safe queues are discussed and are worthy of study. Implementing a thread-safe queue will remove the element of luck.

There is another interesting aspect to the student's code. The function **run()** belonging to **async_logger** contains a loop that continually tests for items in the queue. This is wasteful of processor resources. It would be better for the **async_logger** thread to 'sleep' when there is nothing in the queue and to be 'woken' when something is added to the queue. Such a facility is provided by condition variables.

I have added a mutex, **m**, and a condition variable, **data_available**, to global scope. I have modified **async_logger**'s **log** function by changing the **if** statement for another **while**-loop and created a **unique_lock** object using the mutex **m**. A call is then made to **data_available**'s **wait()** function. **In log()**, I have added a call to **data_available**'s **notify_one()** function just after an item has been placed on the queue. In this way a thread can block on the **wait()** function until awoken by **notify_one()**.

Arranging for **run()** to break out of the **while**-loop is a little tricky. I have chosen to wake-up the blocked thread with another call to **notify_one()** just after setting **active_** to **false**. This requires a modification to **waits()**'s lambda so that execution can continue if **active_** is false. An **if** statement is also added after the newly added **while**-loop of **run()** that has the effect of returning from **run()** when **active_** is **false**. Another way to return from **run()** is to recognise a special message placed on the queue and to return when it is detected.

I provide my version of the code despite it not having a thread-safe queue. It may be instructive, however.

```
--- async_logger.cpp ---
#include "async_logger.h"
#include <iostream>
std::mutex m;
std::condition_variable data_available;
using std::thread;
// This runs in a dedicated thread
void async_logger::run()
{
 while (true)
  {
    std::unique_lock<std::mutex> lock(m);
    data_available.wait(lock, [this]{
      return !queue_.empty() || !active_;});
    while (!queue_.empty())
    {
      std::cout << queue_.front() << '\n';
      queue_.pop();
    }
    if (!active_)
    {
      return;
    }
  }
}
async_logger::~async_logger()
{
  active_ = false;
  data_available.notify_one();
  thread_.join();
}
// queue for processing on the other thread
void async_logger::log(const std::string &str)
{
  queue_.emplace(str);
  data_available.notify_one();
}

--- async_logger.h ---
#ifndef ASYNC_LOGGER
#define ASYNC_LOGGER
#include <iostream>
#include <queue>
#include <string>
#include <thread>
#include <atomic>
#include <mutex>
#include <condition_variable>
using std::thread;
class async_logger
{
private:
  std::atomic_bool active_ = true;
  std::queue<std::string> queue_;
  thread thread_{[this](){run();}};
  void run();
public:
  ~async_logger();
  void log(const std::string &str);
};
#endif

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

Hans Vredeveld <accu@closingbrace.nl>

In making the author's code available to us for review, the variable **thread_** lost its underscore in the declaration on line 14 of `async_logger.h`. After correcting this, the code compiled and ran with the behaviour as described by the author.

One thing that testing this program makes clear, is that testing multi-threaded programs is hard. When I run the program with four arguments, I had to run it 27 times before I got something different than the expected output. When I run it with the complete alphabet as program arguments, it usually didn't give the expected output, but stopped somewhere after outputting the letter m.

Before we delve into the real issues with the code, let's get some annoyances out of the way.

Following the mantra 'include what you use and nothing else', **#include <iostream>** should be removed from `async_logger.h`, and `queue.cpp` should have **#include**s for **<string>** and **<thread>**.

Unscoped **using** declarations in headers should be avoided at all times and used with extreme care in implementation files. They can cause conflicts with identifiers that a user defines herself. At best this will result

in a compiler error. At worst it will compile and link without any error or warning, but use the wrong declaration. We can remove the **using** declaration for **std::thread** in async_logger.h and qualify the types of **thread_** in async_logger.h and **test1** in queue.cpp with **std::**. The **#using std::thread** in async_logger.cpp serves no purpose at all and should be removed.

The last annoyance is the parameter of **async_logger::log**. Its type is **std::string const&**, and this forces the compiler to always copy the string into the queue. Especially for a log-function, that is called with a temporary object most of the time, it would be nice if we could use move semantics. Changing the function to

```
void async_logger::log(std::string str)
{
  queue_.emplace(std::move(str));
}
```

makes this possible. Of course, async_logger.cpp now must **#include <utility>**.

Now that the annoyances are out of the way, let's focus on the real problems. The first thing to notice is that in **async_logger**, **thread_** has a default member initializer and that **active_** is not initialized until inside the constructor body. Depending on thread scheduling, this could mean that **thread_** starts executing **async_logger::run** while **active_** is still not initialized and might have the value **false**. The **while**-loop in **async_logger::run** might thus never be executed. The solution to this problem is to remove the initialization of **active_** from the constructor's body and move it to the member initializer list or to a default member initializer. Also, **active_** is used from multiple threads and not only for reads. Therefore, it should be an **std::atomic_bool** instead of a **bool**.

More on **thread_**. In **async_logger**'s constructor the thread that it creates is detached. Consequently, the lifetime of the running thread is not connected to the lifetime of the **async_logger** instance that created it any more. The thread keeps on running when the **async_logger** instance that it is using, is destroyed, resulting in undefined behaviour. To solve this, remove **thread_.detach()** from the constructor and add **thread_.join()** to the destructor after setting **active_** to **false**. Note that, if **active_** was initialized by a member initializer, now there is no need to define the constructor explicitly any more, and we can leave it to the compiler to generate one.

Now, when **active_** becomes **false**, the thread will drop out of the **while**-loop in **async_logger::run** and the thread will exit. Also, each time the body of the while-loop is executed and there are messages in the queue, only one message is printed. So when the thread drops out of the while-loop, any remaining messages in the queue are not printed, which is the behaviour that the student observed. By changing the if-statement to a while-statement, we make sure that all messages in the queue are printed before **active_** is checked again.

While it now looks like we have the behaviour that the student wanted, there are still two issues with the code. First, **std::queue** is not thread safe. Second, **async_logger::run** uses a busy wait for **queue_** to be filled again. We can solve these issues with a mutex and a condition variable. First, we add them as private members to the **async_logger** class in async_logger.h:

```
std::mutex mutex_;
std::condition_variable cv_;
```

Of course, we must also **#include <mutex>** and **<condition_variable>**. Now before we add a message to the queue in **async_logger::log**, we lock the mutex, and after we add the message, we notify a thread through the condition variable:

```
void async_logger::log(std::string str)
{
  std::lock_guard<std::mutex> lock(mutex_);
  queue_.emplace(std::move(str));
  cv_.notify_one();
}
```

Now **async_logger::run** is a bit more complex:

```
void async_logger::run()
{
  while (true)
  {
    std::queue<std::string> q;
    {
      std::unique_lock<std::mutex>
        lck(mutex_);
      cv_.wait(lck,
        [this](){ return !(queue_.empty() &&
                           active_); });
      if (queue_.empty() && !active_)
          return;
      q = std::move(queue_);
      queue_ = std::queue<std::string>();
    }
    while (!q.empty())
    {
      std::cout << q.front() << '\n';
      q.pop();
    }
  }
}
```

We need to hold a lock on **mutex_** when we are accessing **queue_**, but we don't want to hold the lock when printing the messages. Therefore, we move the messages to a local queue **q**, reinitialize **queue_** as an empty queue and release the lock before we print the messages from the local queue.

Also, we let the thread sleep until there is something in **queue_** or the async logger has become inactive. To make sure that we print the last messages, we change the outer **while**-loop into a perpetual loop and in its body explicitly return from the function when **queue_** is empty and **active_** is false.

Commentary

This simple looking example has multiple problems and goes to show, in Hans's words "that testing multi-threaded programs is hard".

The wording for this critique says the user is trying to write some logging to offload work from their main thread, and thought they'd use this as a chance to "learn something about threading". I would avoid trying to learn threading *indirectly* like this as there are too many 'sharp edges' and the resultant code may well be buggy. Better to work through a tutorial or a book, such as Anthony's *C++ Concurrency in Action* that James mentioned.

As far as the code itself goes, the entrants between them did a great job of finding the problems, explaining what was wrong, and providing solutions, so I don't have a huge amount to add.

One concern I had over James' solution is that he uses *global* variables **m** and **data_available**, which could cause link-time problems (and also have a potential static initialisation order problem).

All the critiques fixed the race condition in **active_**, then made it an **atomic_bool** for thread-safety reasons, and then added a mutex to make access to the queue thread-safe. Once the mutex was added I think the change to **atomic_bool** should be reverted (and the variable accessed while the mutex is locked) as there are *two* conditions for the thread to check and using a single synchronisation mechanism makes it easier to avoid threading errors, such as deadlock.

The original code had a potential inefficiency over copying temporary strings; Hans changed the function to take a string by value which is an optimisation for temporary strings and at least no worse for others. Should measurement indicate the performance of this method is a problem, there are other possibilities to investigate such as adding overloads for common use cases.

A note on **#pragma once**. While this is non-standard, it is very widely supported. There was at least one proposal to standardise this (wg21.link/p0538), but it was rejected in Kona 2017.

The winner of CC 121

All four critiques covered the major problems with the code and produced a solution that fixes the errors. Some additionally pointed out that the reading thread is using a busy loop and corrected this using a condition variable – this is probably the correct thing to do, unless latency is critical!

Diagrams like the one that James made use of can be really useful in a threading algorithm in identifying the dependencies and possible errors in logic, such as race conditions. This can be done in conjunction with dynamic analysis tools, such as those that Paul describes.

I liked Ovidiu's introduction of a helper function **print_oldest_msg** as this made a separation between the logic of handling the queue and the work of processing the contents. While it doesn't make too much difference in such short examples, this is the sort of separation of concerns that can make code significantly easier to understand.

Paul ends his critique by raising some of the design issues with logging and recommends looking for an existing package. I consider that may be the best advice in this case. Ovidiu's opening paragraph lists two basic requirements of an asynchronous logger, which could be the basis of a check list for looking for a third-party logger.

There is a problem with 'liveness' in that the first three solutions process the message in the logging thread with the mutex locked. This blocks the main thread if it tries to log again during the I/O. Hans's solution ensures that the processing of the queue of messages happens outside of the lock. (I think rather than moving from the queue and assigning a default constructed one it might be simpler to just swap the two queues over.)

Ending gracefully remains a potential problem: both Paul's and Hans's critiques as they stand contains a possible deadlock on exit as the *destructor* also needs modifying when the condition variable and mutex are added to ensure it notifies the condition variable. Without this change the code could hang on exit if the logging thread is inside the call to **wait**. when the value of **active_** is changed.

Overall there were four good critiques here, but I thought that Paul's discussion of dynamic analysis tools and his closing comments on the design made his critique broader in coverage and I have awarded him the prize for this issue's critique.

Code Critique 121

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

> I was set a challenge to read the 'Top 100 books' (from the BBC Big Read a few years ago) so thought I could start by seeing which ones I needed to get. Obviously this meant writing a program. Unfortunately, the output from the program isn't sorted, although I was expecting it would be; and in addition, it crashed on Windows.

Can you help the author fix their program so they can begin to catch up with their reading? The program (`the_big_read.cpp`) is in Listing 4.

Listing 4

```
#include <algorithm>
#include <iostream>
template <int A, int B>
void to_buy(const char *(&to_read)[A],
  const char *(&own)[B])
{
  const char** ptr = new const char *[A];
  const char **first = ptr;
  for (int i = 0; i != A; ++i)
  {
    *ptr++ = to_read[i];
    for (int j = 0; j != B; ++j)
    {
      if (!strcmp(to_read[i], own[j]))
      {
        ptr--; // already own it!
      }
    }
  }
  std::sort(first, ptr);
  for (const char **q = first; q != ptr; ++q)
    std::cout << *q << std::endl;
  delete ptr;
}
int main()
{
  const char* top_100 [] = {
    "The Lord of the Rings",
    "Pride and Prejudice",
    "His Dark Materials",
    "The Hitchhiker's Guide to the Galaxy",
    // 93 missing ...
    "Girls In Love",
    "The Princess Diaries",
    "Midnight's Children",
    };
  const char* my_library[] = {
    "The Hitchhiker's Guide to the Galaxy",
    "Vintage Jeeves",
    // and many more ...
    "His Dark Materials",
    "Pride and Prejudice",
    "Emma",
    "Where Eagles Dare",
    };
  to_buy(top_100, my_library);
}
```

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.

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

The Standard Report

Guy Davidson reports from the C++ Standards Committee.

In this report, I'm going to cover the final meeting of this release cycle. In Prague, on February 15th, the committee voted to send the final draft for National Body ballot and then publication, and C++20 was completed.

This is a huge release, the biggest since 2011, with new features such as ranges, modules, coroutines and concepts. The meeting itself was deeply uncontroversial: there is no appetite for putting any spanners in the works, and all the controversy was resolved at Cologne, with the purpose of not having any nasty surprises at this stage.

I'm not going to iterate through the resolved comments, but I will discuss some aspects of the meeting that have long term ramifications.

The first of these is Application Binary Interface (ABI) continuity and compatibility, and how we want to deal with it. You can get a good overview of ABI from the gcc documentation site [1] but in a nutshell, the ABI describes the implementation of the language and library. If this remains stable from version to version then new code can safely interoperate with old code. On the other hand, if an ABI break is introduced then you may need to rebuild everything from scratch if you want to add new code built with a newer toolchain.

The ABI has been stable for GCC since C++11, and for MSVC since C++14. MSVC used to break the ABI with every release, but with the introduction of near-monthly updates it is much harder for them to schedule ABI breaks in the same way.

The idea of introducing an ABI break terrifies some, but the idea of staying stable horrifies others. The cost of an ABI break can be estimated in engineer-millenia worldwide. However, being unable to make changes to things like hashing impacts catastrophically on security, among other things.

Evolution and Library Evolution held a joint meeting on Monday to discuss this problem and sadly seemed to this author to make no real progress, although there was some consensus to make incremental ABI changes with each release of the standard. There is yet more discussion to be had and P2028 [2] outlines all the problems. This is a Gordian knot, with no apparent solution that will not demand enormous effort. Please consider the problem and raise potential solutions on the mailing lists.

The prior chair of Evolution offered a paper [3] which offers a plan for C++23. There seems to be a feeling among the committee that we have delivered a lot of language functionality recently, and the library is starting to look rather threadbare. Looking at languages like Python, there is certainly more that we should be supporting.

The most obvious candidate is completing the long running Networking effort and adding it to the standard. This will also require the completion of work on Executors. An executor is a policy declaration about how to run a function, function object or lambda function (callables). The policy decisions include issues of heterogeneity, i.e. running on an internal or external processor, scheduling, and CPU/GPU/SIMD considerations. These have been developed for some years now, and there was considerable hope that they would land in C++20, along with networking, but it was not to be. These features should, in the author's opinion, take priority for C++23.

Additionally, now that we have modules, we should start work on 'modularising' the standard library; similarly, now that we have coroutines, we should start work on introducing library support for coroutines.

Finally, we should continue working towards reflection, contracts and pattern-matching. In the absence of a counter-proposal, this remains the only plan we have for C++23.

The hosts in Prague, Avast, organised a C++ meetup for Tuesday evening, which was attended by about 300 people and included talks from Bjarne Stroustrup, Tony van Eerd and Herb Sutter. They are all available on YouTube: search for Avast C++ Meetup. Bjarne spoke about the state of generic programming in C++20 which revolves around concepts, Tony delivered his Postmodern C++ talk and Herb spoke about parameter passing and how to fix it, suggesting five new potential keywords to decorate function parameters with. It was the first time he spoke publicly about this and by my estimation it should make the teaching of C++ a little easier and the practice less error-prone.

Avast also organised a social event on Wednesday to celebrate the completion of C++20. It took place in an old church, and the organiser, Hana Dusikova, had asked everyone to wear something nice, something sparkly. The committee did not disappoint. This author took a dinner suit and a top hat. It was not the only top hat present. There were sparkly dresses, hats, and shirts: we seem to scrub up well. The food was excellent, being a menu of local cuisine.

There was a piano present on the stage which attracted some excellent displays of skill from those attending. The convenor, Herb Sutter, is known for entertaining the throng with 80s pop classics wherever he goes. He did not disappoint on this occasion, with a splendid rendition of *Don't Stop Believin*'.

It is the habit of the committee to take a photograph when each standard ships, but in a break with tradition and potentially making ourselves hostages of fortune, we took the photo here, three full days ahead of delivery, in the expectation that we would ship on Saturday. The committee has never looked so good, nor so diverse.

It wasn't all National Body comments of course: there was still business to be done looking forward to landing new features post-C++20. The study groups are thriving, and features like linear algebra are hotly anticipated. If you want to attend a meeting of the C++ committee, the next meeting is in Varna, Bulgaria, at the beginning of June. You can find details online [4]. Meanwhile, in the next two reports, I will cover some of the upcoming language and library features that are hoping to land in C++23.

References

[1] gcc documentation site: https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html

[2] P2028 'What is ABI, and What Should WG21 Do About It?' (2020): https://wg21.link/P2028

[3] P0592 'To boldly suggest an overall plan for C++23' (2019): https://wg21.link/P0592

[4] 2020 Varna Meeting Information: https://wg21.link/N4837

GUY DAVIDSON

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

Reviews

The latest roundup of reviews.

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

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

C++17 In Detail

By Bartłomiej Filipek, published by Leanpub (2019) (continuous updates), ISBN: 9781798834060

Reviewed by Paul Floyd

This book is available with continuous updates: https://leanpub.com/cpp17indetail

Highly recommended

C++17 is the second iteration following a three-year cycle and at the end of last year (2019) we started getting some books specific to C++17. I felt that C++14 got fairly thin coverage in the publishing world (*Programming Principles and Practice*, *Effective Modern C++* and that's about it). C++17 seems to be doing better, with second editions of *C++ Templates*, *A Tour of C++* and *C++ Concurrency in Action* already. Perhaps authors and publishers are having a harder time keeping up with C++ versions? Perhaps also there's more to write about in C++17.

Getting back to the book itself, the book covers only the new features of C++17. You will need to have to have a good understanding of C++ to be able to get much benefit from the book. The first example on page 3 illustrates several new features. If you are not familiar with `std::map`, brace initialisation, `auto` and range based `for` then you won't appreciate how much is new.

The book is broken down into three parts: language, library and use cases. There is plenty of information in each chapter on the C++ Standards Committee papers, where the various changes were proposed, and also of which compilers and versions support each feature. The first few chapters cover some fairly basic items that clean up and simplify the language. I expect that Python programmers will appreciate the new Structured Bindings. Chapter 5 Templates (including fold expressions and 'if constexpr'). In my opinion these are the most important changes in C++17 (especially 'if constexpr') which continue the process of enabling easier and more powerful template programming. The language part ends with a chapter on attributes.

The second part covers the library. The first three items – `std::optional`, `std::variant` and `std::any` – have been with us for some time in `boost::` guise. It's good to see these in the library and I think that they will be very useful tools. I likes the coverage of performance and exceptions, and there are also summaries of changes from the `boost::` versions.

The next three chapters are dedicated to strings: `std::string_view`, string conversions and string searching. Searching isn't really up my street, but it's always good to know. `string_view` and the conversions look promising, at least from a performance perspective. I had already heard some misgivings about the risks of using `string_view`, and that is well covered. The next two chapters are big items: Filesystem and Parallel Algorithms. There is a lot to cover for filesystems, but everything is covered albeit briefly, and there are a few decent examples of using the interface. Parallel Algorithms is mostly about performance. A little niggle here, I would have liked to see some more useful example for the uses of `std::reduce`/`std::transform_reduce`, for instance the variance with an explanation of why `std::reduce` is not suitable for such calculations. There is a fair amount of performance analysis. The last two chapters of Part 2 cover small items and standard library cleanup.

The last part is a brief overview of some use cases. The first three are refactoring using `optional`/`variant`, `[[nodiscard]]` and `if constexpr`. The last example parallelises a CSV file reader.

I enjoyed reading the book. It is concise enough that I could read it all in a few days whilst at the same time giving enough coverage that I had a clear impression of the new additions to C++17.

C++17 The Complete Guide

By Nicolai M Josuttis, published by Leanpub (2019), ISBN: 978-3-96730-017-8

Reviewed by Paul Floyd

The book is available with continuous updates: https://leanpub.com/cpp17

Highly recommended

One of the most difficult aspects with this book is that it is similar to *C++17 In Detail* by Bartłomiej Filipek. If you are asking yourself "Which one should I get?", then I'm afraid that I'm not going to be much help as my usual answer to such a question is to buy both. This book is slightly longer and a bit denser.

This book only covers the 'deltas' that C++17 has added to (and to a small extent subtracted from) the prior C++ standards. This means that you will need to have a good working knowledge of C++03/11/14 in order to read and benefit from this book.

There are six sections to the book: basic language, template features, new and changed library components, expert utilities and finally general hints. Each section is divided into chapters of around 10 pages (a bit longer for features that require a lot of details like 'filesystem'). The overall style is mostly descriptive text with annotated snippets. There are some longer examples of runnable code, for instance a `mallinfo`-style allocator tracker for `new` and `delete`. I won't go into detail on all of the 35 chapters (the table of contents can be seen on the book's Leanpub web page). Suffice it to say that I felt that there is a good amount of coverage on all of the topics.

One little thing that I felt could have been added is a brief word on compiler/standard library/platform support. I'd like to play around with the parallel algorithms a bit, but only some compiler/library/platform combinations support them currently.

View from the Chair

Bob Schmidt
chair@accu.org

Attentive readers will notice something new this month. After almost four years of writing this column, I finally got around to taking a picture to include at the top of the page. (Sorry it took so long, Steve.) Hopefully this won't scare any of you away.

Other than the picture, there's not much to cover this month. Due to publishing deadlines, I am writing this in early February for an early March publishing date. Most of you will receive your copy just a few days prior to the start of the spring conference.

ACCU 2020

Archer Yates Associates and ACCU are putting the finishing touches to the plans for our spring conference, scheduled for the 25th through 28th of March, 2020, with pre-conference workshops scheduled for March 24th. The conference once again will be held at the Bristol Marriott City Centre.

Barring some unforeseen circumstance, I will be attending the conference. I hope to see you there.

2020 AGM

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

Nominations for committee positions were finalized on 28 January. Nominees are:

Name	Position
Patrick Martin	Treasurer
Matthew Jones	Membership Secretary
Roger Orr	Publications
Phil Nash	Local Groups
Guy Davidson	Standards
Ralph McArdell	At-Large
Jim Hague	Web Master
Ian Bruntlett	Reviews Editor

Currently there are no nominees for Chair or Secretary.

Online voting should start on 7 March. Members should look for an email directing you to our election page.

It's never too late to express interest in serving on the committee. Please contact our secretary, Patrick Martin (secretary@accu.org) or me (chair@accu.org) if you are interested.

Call for Volunteers

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

The positions the committee has determined are chronically vacant are: publicity, study groups, social media, and web editor. Contact Patrick or me if you are interested in one of these roles.

Letter to the Editor

Silas S. Brown responds to Ian Bruntlett's article.

Dear Editor,

I enjoyed Ian Bruntlett's article 'Static C library and GNU Make' (*C Vu* 31.6, January 2020), but I hope he won't mind my pointing out some improvements that can be made to the code, in the spirit of ACCU being a good learning environment.

Firstly, I'm afraid there is a bug in Listing 2, specifically this part:

```
for (int rhs = length-1;
  isspace(text[rhs]);
  --rhs
  )
  text[rhs] = '\0';
```

Consider what happens when **`text`** is a string consisting entirely of spaces and nothing else. In this case the loop decrementing **`rhs`** will quite happily set it to **`-1`** and try to read it for **`isspace()`**. In most cases, the byte before the string starts will (a) be readable by your program and (b) not happen to contain a space, in which case nothing will happen. But if it does happen to contain a space, then we're looking at memory corruption or, in the unlikely event that its address is off-limits to your program, you'll get a segmentation fault.

The easiest fix would be to add a condition for **`rhs >= 0`** in the loop. But if we start using **`size_t`** instead of **`int`** (which we'd need to do if we want to support strings that are so enormous their lengths cannot be represented by a signed integer on the platform) then we'd have to explicitly write **`if (rhs==0) break;`** at the end of the loop body.

Incidentally, it is not really necessary to overwrite *every* space with **`\0`** when right-trimming; only the first one needs to be overwritten (and this might make a speed difference if memory-writes are slower than memory-reads on the system). So it might be better to write the loop as:

```
size_t rhs;
for (rhs=length; rhs && isspace(text[rhs-1]);
     rhs--);
text[rhs] = '\0';
```

and while we're making such improvements, **`ltrim()`** might as well use **`memmove()`** rather than writing the loop manually (the library **`memmove`** will likely have optimisations that can move multiple bytes at a time using the CPU's extended registers), and the two **`ltrimcpy`** and **`rtrimcpy`** functions can be improved to copy only the non-whitespace portion of the string (after all, there's no point in copying the spaces only to remove them). But on the other hand, programmer time is also a resource and sometimes we shouldn't use our time optimising things that don't really need to be optimal; I'm mentioning this only for completeness.

I realise the thrust of the article is about Makefiles and not really about the code itself, but still thought the above might be worth pointing out for the sake of the learners.

Many thanks.

Silas